Module 1

Distributed computing

Syllabus

Evolution of Distributed Computing -Issues or challenges in designing a distributed system - Minicomputer model – Workstation model - Workstation-Server model–Processor - pool model - Trends in distributed systems

1.1 Introduction

A distributed system is a system whose components are located on different networked computers, which then communicate and coordinate their actions by passing messages to each other.The components interact with each other in order to achieve a common goal. Three significant characteristics of distributed systems are:

- concurrency of components The main function od distributed system is to share resources and hence distributed components must be able to handle more than one user.

- lack of a global clock Programs running on distributed systems communicate but they don't follow the same clock and with current technology it is not possible to synchronize their clocks, and

- independent failure of components A failure of a computer in a distributed system will not bring down the entire system as it is designed to handle such individual component failures

Examples of distributed systems vary from SOA-based systems to massively multiplayer online games to peer-to-peer applications.

Definitions

- "A system in which hardware or software components located at networked computers communicate and coordinate their actions only by message passing."[Text Book] or

- "A distributed system is a collection of independent computers that appear to the users of the system as a single computer." [Tanenbaum] or

- A distributed system is several computers doing something together. Thus, a distributed system has three primary characteristics: multiple computers, interconnections, and shared state. [Michael Schroeder]

1.2 Examples of distributed systems

Networks are everywhere and underpin many everyday services that we now take for granted: the Internet and the associated World Wide Web, web search, online gaming, email, social networks, eCommerce, etc.

Distributed systems encompass many of the most significant technological developments of recent years and hence an understanding of the underlying technology is absolutely central to a knowledge of modern computing.

We now look at more specific examples of distributed systems to further illustrate the diversity and indeed complexity of distributed systems provision today.

1.2.1 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, encompassing a wide range of information styles including web pages, multimedia sources and (scanned) books. This is a very complex task, as current estimates state that the Web consists of over 63 billion pages and one trillion unique web addresses. 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 (and indeed other Google applications and services such as Google Earth). This represents one of the largest and most complex distributed systems installations in the history of computing and hence demands close examination. Highlights of this infrastructure include:

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

• a distributed file system designed to support very large files and heavily optimized for the style of usage required by search and other Google applications (especially reading from files at high and sustained rates);

• 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 locking and agreement;

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

1.2.2 Massively 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 PUBG, FORTNITE etc. Such worlds have increased significantly in sophistication. The number of players is also rising, with systems able to support over 50,000 simultaneous online players (and the total number of players perhaps ten times this figure).

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. Other challenges include the real-time propagation of events to the many players and maintaining a

consistent view of the shared world. This therefore provides an excellent example of the challenges facing modern distributed systems designers.

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

• 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

•

• 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. Users are then dynamically allocated a particular server based on current usage patterns and also the network delays to the server (based on geographical proximity for example). This style of architecture, which is adopted by EverQuest, is naturally extensible by adding new servers.

• Most commercial systems adopt one of the two models presented above, but researchers are also now looking at more radical architectures that are not based on client-server principles but rather adopt completely decentralized approaches based on peer-to-peer technology where every participant contributes resources (storage and processing) to accommodate the game.

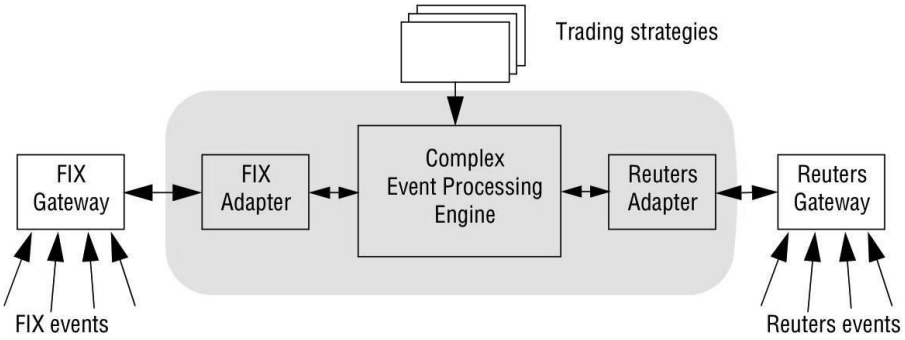
1.2.3 Financial trading

As a final example, we look at distributed systems support for financial trading markets. 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 (for example, current share prices and trends, economic and political developments). The industry employs automated monitoring and trading applications (see below).

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 (for example client-server), and such systems typically employ what are known as *distributed event-based systems*. We present an illustration of a typical use of such systems below and return to this important topic in more depth in Chapter 6.

Figure 1.2 illustrates a typical financial trading system. This shows a series of event feeds coming into a given financial institution. Such event feeds share the

Figure 1.2 An example financial trading system



following characteristics. Firstly, the sources are typically in a variety of formats, such as Reuters market data events and FIX events (events following the specific format of the Financial Information eXchange protocol), and indeed from different event technologies, thus illustrating the problem of heterogeneity as encountered in most distributed systems (see also Section 1.5.1). The figure shows the use of adapters which translate heterogeneous formats into a common internal format. Secondly, the trading system must deal with a variety of event streams, all arriving at rapid rates, and often requiring real-time processing to detect patterns that indicate trading opportunities. This used to be a manual process but competitive pressures have led to increasing automation in terms of what is known as Complex Event Processing (CEP), which offers a way of composing event occurrences together into logical, temporal or spatial patterns.

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. As an example, consider the following script:

WHEN

MSFT price moves outside 2% of MSFT Moving Average

FOLLOWED-BY (

MyBasket moves up by 0.5%

AND

HPQ's price moves up by 5%

OR

MSFT's price moves down by 2%

)

)

ALL WITHIN

any 2 minute time period

THEN

BUY MSFT

SELL HPQ

1.3 Evolution of Distributed Computing

Early computers were very expensive (they cost millions of dollars) and very large in size (they occupied a big room). There were very few computers and were available only in research laboratories of universities and industries. These computers were run from a console by an operator and were not accessible to ordinary users. The programmers would write their programs and submit them to the computer center on some media, such as punched cards, for processing. Before processing a job, the operator would set up the necessary environment (mounting tapes, loading punched cards in a card reader, etc.) for processing the job. The job was then executed and the result, in the form of printed output, was later returned to the programmer.

The job setup time was a real problem in early computers and wasted most of the valuable central processing unit (CPU) time. Several new concepts were introduced in the 1950s and 1960s to increase CPU utilization of these computers. Notable among these are batching together of jobs with similar needs before processing them, automatic sequencing of jobs, off-line processing by using the concepts of buffering and spooling, and multiprogramming.

Batching similar jobs improved CPU utilization quite a bit because now the operator had to change the execution environment only when a new batch of jobs had to be executed and not before starting the execution of every job. Automatic job sequencing with the use of control cards to define the beginning and end of a job improved CPU utilization by eliminating the need for human job sequencing. Off-line processing improved CPU utilization by allowing overlap of CPU and input/output (I/O) operations by executing those two actions on two independent machines (110 devices are normally several orders of magnitude slower than the CPU). Finally, multiprogramming improved CPU utilization by organizing jobs so that the CPU always had something to execute.

However, none of these ideas allowed multiple users to directly interact with a computer system and to share its resources simultaneously. Therefore, execution of interactive jobs that are composed of many short actions in which the next action depends on the result of a previous action was a tedious and time-consuming activity. Development and debugging of programs are examples of interactive jobs. It was not until the early 1970s that computers started to use the concept of time sharing to overcome this hurdle.

Early time-sharing systems had several dumb terminals attached to the main computer. These terminals were placed in a room different from the main computer room. Using these terminals, multiple users could now simultaneously execute interactive jobs and share the resources of the computer system. In a time-sharing system, each user is given the impression that he or she has his or her own computer because the system switches rapidly from one user's job to the next user's job, executing only a very small part of each job at a time. Although the idea of time sharing was demonstrated as early as 1960,

time- sharing computer systems were not common until the early 1970s because they were difficult and expensive to build.

Parallel advancements in hardware technology allowed reduction in the size and increase in the processing speed of computers, causing large-sized computers to be gradually replaced by smaller and cheaper ones that had more processing capability than their predecessors. These systems were called minicomputers. The advent of time-sharing systems was the first step toward distributed computing systems because it provided us with two important concepts used in distributed computing systems-the sharing of computer resources simultaneously by many users and the accessing of computers from a place different from the main computer room. Initially, the terminals of a time-sharing system were dumb terminals, and all processing was done by the main computer system.

Advancements in microprocessor technology in the 1970s allowed the dumb terminals to be replaced by intelligent terminals so that the concepts of off-line processing and time sharing could be combined to have the advantages of both concepts in a single system. Microprocessor technology continued to advance rapidly, making available in the early 1980s single-user computers called workstations that had computing power almost equal to that of minicomputers but were available for only a small fraction of the price of a minicomputer. For example,the first workstation developed at Xerox PARC (called Alto) had a high-resolution monochrome display, a mouse, 128 kilobytes of main memory, a 2.5-megabyte hard disk, and a microprogrammed CPU that executed machine-level instructions at speeds of 2-6 f.Ls. These workstations were then used as terminals in the time-sharing systems. In these time-sharing systems, most of the processing of a user's job could be done at the user's own computer, allowing the main computer to be simultaneously shared by a larger number of users. Shared resources such as files, databases, and software libraries were placed on the main computer.

Centralized time-sharing systems described above had a limitation in that the terminals could not be placed very far from the main computer room since ordinary cables were used to connect the terminals to the main computer. However, in parallel, there were advancements in computer networking technology in the late 1960s and early 1970s that emerged as two key networking technologies-LAN (local area network) and WAN (wide-area network). The LAN technology allowed several computers located within a building or a campus to be interconnected in such a way that these machines could exchange information with each other at data rates of about 10 megabits per second (Mbps). On the other hand, WAN technology allowed computers located far from each other (may be in different cities or countries or continents) to be interconnected in such a way that these machines could exchange information with each other at data rates of about 56 kilobits per second (Kbps).

The first high-speed LAN was the Ethernet developed at Xerox PARe in 1973, and the first WAN was the ARPAnet (Advanced Research Projects Agency Network) developed by the U.S. Department of Defense in 1969. The data rates of networks continued to improve gradually in the 1980s, providing data rates of up to 100 Mbps for LANs and data rates of up to 64 Kbps for WANs. Recently (early 1990s) there has been another major advancement in networking technology-the ATM (asynchronous transfer mode) technology. The Al'M technology is an emerging technology that is still not very well established. It will make very high speed networking possible, providing data transmission rates up to 1.2 gigabits per second (Gbps) in both LAN and WAN environments. The availability of such high-bandwidth networks will allow future distributed computing systems to support a completely new class of distributed applications, called multimedia applications, that deal with the handling of a mixture of information, including voice, video, and ordinary data. These applications were previously unthinkable with conventional LANs and WANs.

The merging of computer and networking technologies gave birth to distributed computing systems in the late 1970s. Although the hardware issues of building such systems were fairly well understood, the major stumbling block at that time was the availability of adequate software for making these systems

easy to use and for fully exploiting their power. Therefore, starting from the late 1970s, a significant amount of research work was carried out in both universities and industries in the area of distributed operating systems. These research activities have provided us with the basic ideas of designing distributed operating systems. Although the field is still immature, with ongoing active research activities, commercial distributed operating systems have already started to emerge. These systems are based on already established basic concepts. This book deals with these basic concepts and their use in the design and implementation of distributed operating systems. Several of these concepts are equally applicable to the design of applications for distributed computing systems, making this book also suitable for use by the designers of distributed applications.

1.4 Issues or challenges in designing a distributed system

As the scope and scale or size of distributed systems and applications is extended the challenges are likely to be encountered. In this section we describe the main challenges.

1.4.1 Heterogeneity

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

- networks;
- computer hardware;
- operating systems;
- programming languages;
- implementations by different developers.

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

Different programming languages use different representations for characters and data structures such as arrays and records. These differences must be addressed if programs written in different languages are to be able to communicate with one another.

Programs written by different developers cannot communicate with one another unless they use common standards, for example, for network communication and the representation of primitive data items and data structures in messages. For this to happen, standards need to be agreed and adopted

– as have the Internet protocols.

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. The Common Object Request Broker (CORBA) and Java Remote Method Invocation (RMI) are examples. Most middleware is implemented over the Internet protocols and deals with the differences in operating systems and .

Heterogeneity and mobile code • 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 and JavaScript codes 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.

1.4.2 Openness

The openness of a computer system is the characteristic that determines whether the system can be extended by programmers in various ways. Openness cannot be achieved unless the specification and documentation of the components of a system are made available to software developers. In a word, the key programming standards are *published*.

The designers of the Internet protocols introduced a series of documents called 'Requests For Comments', or RFCs. The specifications of the Internet communication protocols were published in this series in the early 1980s, followed by specifications for applications that run over them, such as file transfer, email and telnet by the mid-1980s. This open publication of standards have allowed companies and startups like google and Facebook to build their applications

Systems that are designed to support resource sharing in this way are termed *open distributed systems* to emphasize the fact that they are extensible. They may be extended at the hardware level by the addition of computers to the network and at the software level by the introduction of new services.

1.4.3 Security

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

Section 1.1 pointed out that although the Internet allows a program in one computer to communicate with a program in another computer irrespective of its location, security risks are associated with allowing free access to all of the resources in an intranet. Although a firewall can be used to form a barrier around an intranet, restricting the traffic that can enter and leave, this does not deal with ensuring the appropriate use of resources by users within an intranet, or with the appropriate use of resources in the Internet, that are not protected by firewalls.

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. Two security challenges can happen here:

Denial of service attacks: A 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; for example, it may seem to display an interesting picture but in reality it may access local resources, or perhaps be part of a denial of service attack.

1.4.4 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. In general, for a system with n users to be scalable, the quantity of physical resources required to support them should be at most $O(n)$ – that is, proportional to n. For example, if a single file server can support 20 users, then two such servers should be able to support 40 users.

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. Algorithms that use hierarchic structures scale better than those that use linear structures. But even with hierarchic structures an increase in size will result in some loss in performance: the time taken to access hierarchically structured data is $O(\log n)$, where n is the size of the set of data. For a system to be scalable, the maximum performance loss should be no worse than this.

Preventing software resources running out: An example of lack of scalability is shown by the numbers used as Internet (IP) addresses (computer addresses in the Internet). In the late 1970s, it was decided to use 32 bits for this purpose= the supply of available Internet addresses is running out.

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 as number of computers became larger and larger.

1.4.5 Failure handling

Computer systems sometimes fail. 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 are discussed throughout the book:

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. 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. For example, when a web browser cannot contact a web server, it does not make the user wait for ever while it keeps on trying – it informs the user about the problem, leaving them free to try again later.

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 two different 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.

1.4.6 Concurrency

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

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.

1.4.7 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.

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

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

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

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

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

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

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

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

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

As an illustration of access transparency, consider a graphical user interface with folders, which is the same whether the files inside the folder are local or remote.

As an illustration of the presence of network transparency, consider the use of an electronic mail address such as *Fred.Flintstone@stoneit.com*. The address consists of a user's name and a domain name. Sending mail to such a user does not involve knowing their physical or network location.

Failure transparency can also be illustrated in the context of electronic mail, which is eventually delivered, even when servers or communication links fail. The faults are masked by attempting to retransmit messages until they are successfully delivered, even if it takes several days.

To illustrate mobility transparency, consider the case of mobile phones. Suppose that both caller and callee are travelling by train in different parts of a country, moving

from one environment (cell) to another. We regard the caller's phone as the client and the callee's phone as a resource. The two phone users making the call are unaware of the mobility of the phones (the client and the resource) between cells.

1.4.8 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 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.

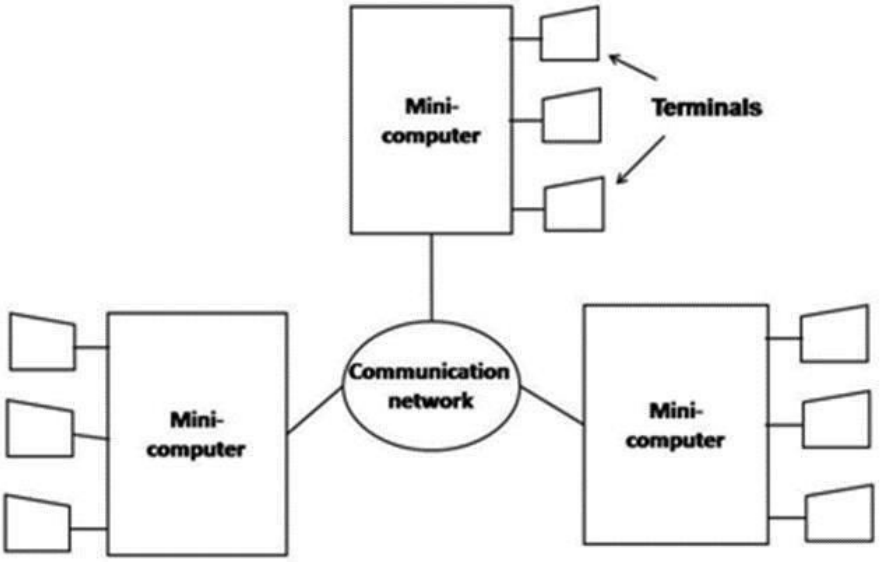
DISTRIBUTED COMPUTING SYSTEM MODELS

Various models are used for building distributed computing systems. These models can be broadly classified into five categories-minicomputer, workstation, workstation-server, processor-pool, and hybrid. They are briefly described below.

1.5 Minicomputer Model

The minicomputer model is a simple extension of the centralized time-sharing system. As shown in Figure below, a distributed computing system based on this model consists of a few minicomputers (they may be large supercomputers as well) interconnected by a communication network. Each minicomputer usually has multiple users simultaneously logged on to it. For this, several interactive terminals are connected to each minicomputer. Each user is logged on to one specific minicomputer, with remote access to other minicomputers. The network allows a user to access remote resources that are available on some machine other than the one on to which the user is currently logged. The minicomputer model may be used when resource sharing (such as sharing of information databases of different types, with each type of database located on a different machine) with remote users is desired.

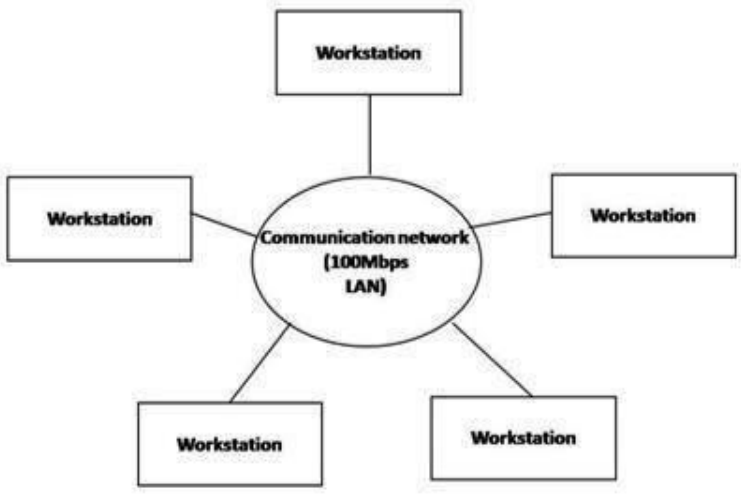
The early ARPAnet is an example of a distributed computing system based on the minicomputer model.



1.6 Workstation Model

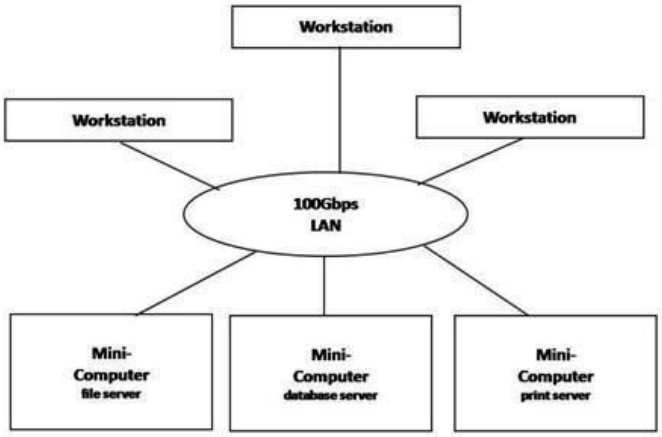
As shown in Figure below, a distributed computing system based on the workstation model consists of several workstations interconnected by a communication network. A company's office or a university department may have several workstations scattered throughout a building or campus, each workstation equipped with its own disk and serving as a single-user computer. It has been often found that in such an environment, at any one time (especially at night), a significant proportion of the workstations are idle (not being used), resulting in the waste of large amounts of CPU time. Therefore, the idea of the workstation model is to interconnect all these workstations by a high-speed LAN so that idle workstations may be used to process jobs of users who are logged onto other workstations and do not have sufficient processing power at their own workstations to get their jobs processed efficiently.

In this model, a user logs onto one of the workstations called his or her "home" workstation and submits jobs for execution. When the system finds that the user's workstation does not have sufficient processing power for executing the processes of the submitted jobs efficiently, it transfers one or more of the processes from the user's workstation to some other workstation that is currently idle and gets the process executed there, and finally the result of execution is returned to the user's workstation.



1.7 Workstation-Server Model

The workstation model is a network of personal workstations, each with its own disk and a local file system. A workstation with its own local disk is usually called a diskful workstation and a workstation without a local disk is called a diskless workstation. With the proliferation of high-speed networks, diskless workstations have become more popular in network environments than diskful workstations, making the workstation-server model more popular than the workstation model for building distributed computing systems. As shown in Figure below, a distributed computing system based on the workstation-servermodel consists of a few minicomputers and several workstations (most of which are diskless, but a few of which may be diskful) interconnected by a communication network.



Note that when diskless workstations are used on a network, the file system to be used by these workstations must be implemented either by a diskful workstation or by a minicomputer equipped with a disk for file storage. Other minicomputers may be used for providing other types of services, such as database service and print service. Therefore, each minicomputer is used as a server machine to provide one or more types of services. Hence in the workstation-server model, in addition to the workstations, there are specialized machines (may be specialized workstations) for running server processes (called servers) for managing and providing access to shared resources.

In this model, a user logs onto a workstation called his or her home workstation. Normal computation activities required by the user's processes are performed at the user's home workstation, but requests for services provided by special servers (such as a file server or a database server) are sent to a server providing that type of service that performs the user's requested activity and returns the result of request processing to the user's workstation. Therefore, in this model, the user's processes need not be migrated to the server machines for getting the work done by those machines.

For better overall system performance, the local disk of a diskful workstation is normally used for such purposes as storage of temporary files, storage of unshared files, storage of shared files that are rarely changed, paging activity in virtual-memory management, and caching of remotely accessed data.

As compared to the workstation model, the workstation-server model has several advantages:

1. In general, it is much cheaper to use a few minicomputers equipped with large, fast disks that are accessed over the network than a large number of diskful workstations, with each workstation having a small, slow disk.

2. Diskless workstations are also preferred to diskful workstations from a system maintenance point of view. Backup and hardware maintenance are easier to perform with a few large disks than with many small disks scattered all over a building or campus.

Furthermore, installing new releases of software (such as a file server with new functionalities) is easier when the software is to be installed on a few file server machines than on every workstation.

3. In the workstation-server model, since all files are managed by the file servers, users have the flexibility to use any workstation and access the files in the same manner irrespective of which workstation the user is currently logged on. Note that this is not true with the workstation model, in which each workstation has its local file system, because different mechanisms are needed to access local and remote files.

4. In the workstation-server model, the request-response protocol described above is mainly used to access the services of the server machines. Therefore, unlike the workstation model, this model does not need a process migration facility, which is difficult to implement.

The request-response protocol is known as the client-server model of communication. In this model, a client process (which in this case resides on a workstation) sends a request to a server process (which in this case resides on a minicomputer) for getting some service such as reading a block of a file. The server executes the request and sends back a reply to the client that contains the result of request processing.

A user has guaranteed response time because workstations are not used for executing remote processes. However, the model does not utilize the processing capability of idle workstations.

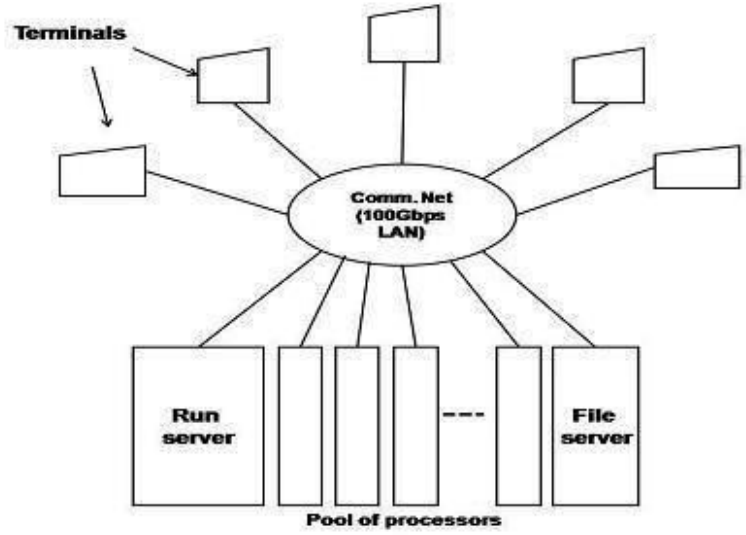
The V-System [Cheriton 1988] is an example of a distributed computing system that is based on the workstation-server model.

1.8 Processor-Pool Model

The processor-pool model is based on the observation that most of the time a user does not need any computing power but once in a while he or she may need a very large amount of computing power for a short time (e.g., when recompiling a program consisting of a large number of files after changing a basic shared declaration). Therefore, unlike the workstation-server model in which a processor is allocated to each user, in the processor-pool model the processors are pooled together to be shared by the users as needed. Each processor in the pool has its own memory to load and run a system program or an application program of the distributed computing system.

As shown in Figure below, in the pure processor-pool model, the processors in the pool have no terminals attached directly to them, and users access the system from terminals that are attached to the network via special devices. These terminals are either small diskless workstations or graphic terminals, such as X terminals. A special server (called a run server) manages and allocates the processors in the pool to different users on a demand basis. When a user submits a job for computation, an appropriate number of processors are temporarily assigned to his or her job by the run server. For example, if the user's computation job is the compilation of a program having segments, in which each of the segments can be compiled independently to produce separate relocatable object files, n processors from the pool can be allocated to this job to compile all the n segments in parallel. When the computation is completed, the processors are returned to the pool for use by other users.

In the processor-pool model there is no concept of a home machine. That is, a user does not log onto a particular machine but to the system as a whole.



As compared to the workstation-server model, the processor-pool model allows better utilization of the available processing power of a distributed computing system. This is because in the processor-pool model, the entire processing power of the system is available for use by the currently logged-on users, whereas this is not true for the workstation-server model in which several workstations may be idle at a particular time but they cannot be used for processing the jobs of other users. Furthermore, the processor-pool model provides greater flexibility than the workstation-server model in the sense that the system's services can be easily expanded without the need to install any more computers; the processors in the pool can be allocated to act as extra servers to carry any additional load arising from an increased user population or to provide new services.

However, the processor-pool model is usually considered to be unsuitable for high-performance interactive applications, especially those using graphics or window systems. This is mainly because of the slow speed of communication between the computer on which the application program of a user is being executed and the terminal via which the user is interacting with the system. The workstation-server model is generally considered to be more suitable for such applications.

Amoeba [Mullender et a1. 1990], Plan 9 [Pike et at 1990], and the Cambridge Distributed Computing System [Needham and l-lerbert 1982] are examples of distributed computing systems based on the processor-pool model.

1.9 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 user mobility in distributed systems;
- the increasing demand for multimedia services;
- the view of distributed systems as a utility.

1.9.1 Pervasive networking and the modern Internet

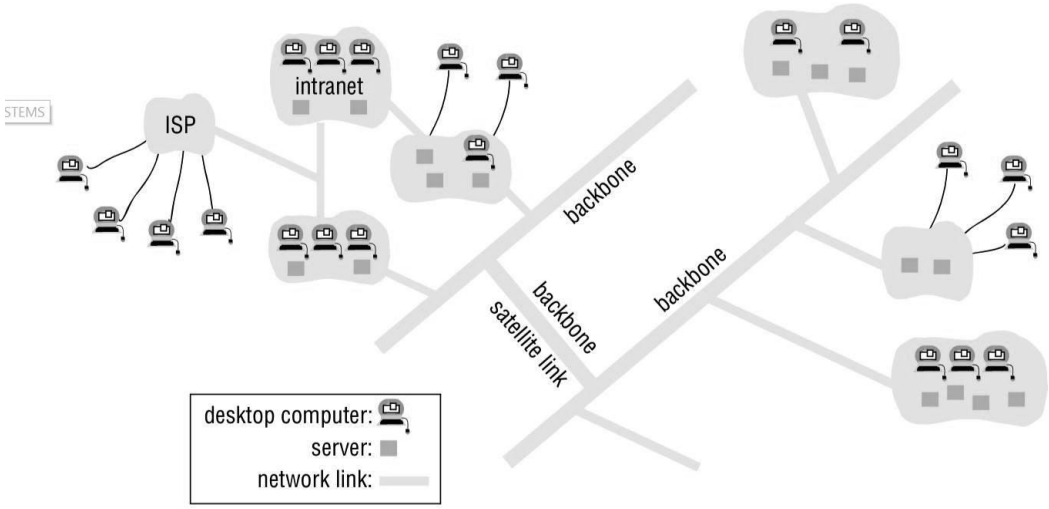
The modern Internet is a vast interconnected collection of computer networks of many different types, with the range of types increasing all the time and now including, for example, a wide range of wireless communication technologies such as WiFi, WiMAX, Bluetooth and third-generation mobile phone networks.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

Figure below illustrates a typical portion of the Internet. Programs running on the computers connected to it interact by passing messages, employing a common means of communication. The Internet communication mechanisms (the Internet protocols) is a major technical achievement, enabling a program running anywhere to address messages to programs anywhere else and abstracting over the myriad of technologies mentioned above.

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. (Indeed, the Web is sometimes incorrectly equated with the Internet.) The set of services is open-ended – it can be extended by the addition of server computers and new types of service. The figure shows a collection of intranets – subnetworks operated by companies and other organizations and typically protected by firewalls. 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. The intranets are linked together by backbones. A *backbone* is a network link with a high transmission capacity, employing satellite connections, fibre optic cables and other high-bandwidth circuits.

Note that some organizations may not wish to connect their internal networks to the Internet at all. For example, police and other security and law enforcement agencies are likely to have at least some internal intranets that are isolated from the outside world (the most effective firewall possible – the absence of any physical connections to the Internet).

The implementation of the Internet and the services that it supports has entailed the development of practical solutions to many distributed system issues.



1.9.2 Mobile and ubiquitous(everywhere) 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.
- Handheld devices, including mobile phones, smart phones, GPS-enabled devices, pagers, personal digital assistants (PDAs), video cameras and digital cameras.
- Wearable devices, such as smart watches with functionality similar to a PDA.
- Devices embedded in appliances such as washing machines, hi-fi systems, cars and refrigerators.

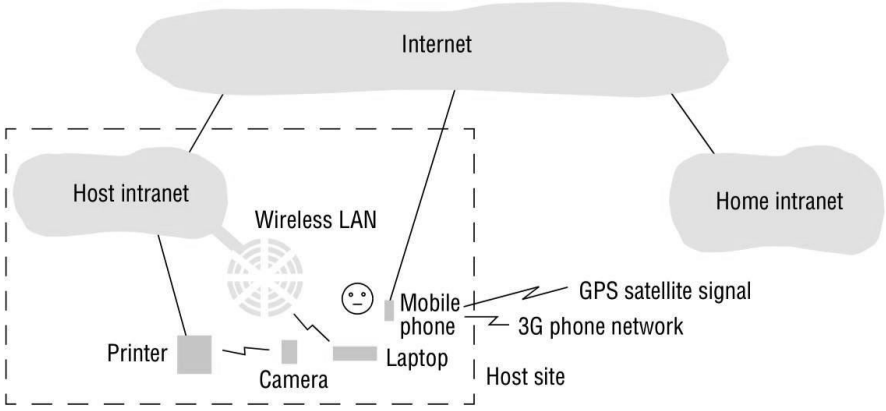
The portability of many of these devices, together with their ability to connect conveniently to networks in different places, makes *mobile computing* possible. Mobile computing is the performance of computing tasks while the user is on the move, or visiting places other than their usual environment. 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. The term 'ubiquitous' is intended to suggest that small computing devices will eventually become so pervasive in everyday objects that they are scarcely noticed. That is, their computational behaviour will be transparently and intimately tied up with their physical function.

The presence of computers everywhere only becomes useful when they can communicate with one another. For example, it may be convenient for users to control their washing machine or their entertainment system from their phone or a 'universal remote control' device in the homee.

Figure below shows a user who is visiting a host organization. The figure shows the user's home intranet and the host intranet at the site that the user is visiting. Both intranets are connected to the rest of the Internet.

The user has access to three forms of wireless connection. Their laptop has a means of connecting to the host's wireless LAN. This network provides coverage of a



few hundred metres (a floor of a building, say). It connects to the rest of the host intranet via a gateway or access point. The user also has a mobile (cellular) telephone, which is connected to the Internet. The phone gives access to the Web and other Internet services, constrained only by what can be presented on its small display, and may also provide location information via built-in GPS functionality. Finally, the user carries a digital camera, which can communicate over a personal area wireless network (with range up to about 10m) with a device such as a printer.

With a suitable system infrastructure, the user can perform some simple tasks in the host site using the devices they carry. While journeying to the host site, the user can fetch the latest stock prices from a web server using the mobile phone and can also use the built-in GPS and route finding software to get directions to the site location.

This scenario demonstrates the need to support *spontaneous interoperation,* whereby associations between devices are routinely created and destroyed – for example by locating and using the host's devices, such as printers. The main challenge applying to such situations is to make interoperation fast and convenient (that is, spontaneous) even though the user is in an environment they may never have visited before. That means enabling the visitor's device to communicate on the host network, and associating the device with suitable local services – a process called *service discovery*.

1.9.3 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 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 playback of the media to the user and optionally also to share the media across a group of users.

The crucial characteristic of media types is that they are time critical, and indeed, the integrity of the media type is fundamentally dependent on preserving real-time relationships between audio and video of a video stream.

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(e.g. Hotstar), access to music libraries, the provision of audio and video conferencing facilities and integrated telephony features including IP telephony or related technologies such as Skype, a peer-to-peer alternative to IP.

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

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

3. providing support for an (extensible) range of video 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;

4. providing a range of mechanisms to ensure that the desired quality of service can be met; i.e, no frame drops or buffering

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

6. providing adaptation strategies to deal with the inevitable situation in open systems where quality of service cannot be met or sustained for example when speed of a user goes from 4g to 2g due to lack of connectivity.

1.9.4 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. With this model, resources are provided by appropriate service suppliers and effectively rented rather than owned by the end user. 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 (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.

- Software services can also be made available across the global Internet using this approach. Example google docs, which provides the office suite over the cloud.

Cloud Computing

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, thus enabling them to largely or totally dispense with local data storage and application software. The term also promotes a view of everything as a service, from physical or virtual infrastructure through to software, often paid for on a per-usage basis rather than purchased. Note that cloud computing reduces requirements on users' devices, allowing very simple desktop or portable devices to access a potentially wide range of resources and services.

Clouds are generally implemented on cluster computers to provide the necessary scale 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. Building on projects such as the NOW (Network of Workstations) Project at Berkeley [Anderson *et al*. 1995, now.cs.berkeley.edu] and Beowulf at NASA [www.beowulf.org], the trend is towards utilizing commodity hardware both for the computers and for the interconnecting networks. Most clusters consist of commodity PCs running a standard (sometimes cut-down) version of an operating system such as Linux, interconnected by a local area network. Companies such as HP, Sun and IBM offer blade solutions.

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 (Google, for example relies on a massive cluster computer architecture to implement its search engine and other services).

MODULE 2
SYSTEM MODELS
System models: Physical models - Architectural models -Fundamental models

Introduction

Distributed Systems that are intended for use in real-world environments should be designed to function correctly in the widest possible range of circumstances and in the face of many possible difficulties and threats We show how the properties and issues of distributed systems can be understood through the use of descriptive models. Each type of model is intended to provide an abstract(high level) and simplified description of a relevant aspect of distributed system design:

Physical models consider the types of computers and devices that constitute a distributed system and how they are connected.

Architectural models describe a system in terms of the computational elements(computers) and communications performed by its computational elements

Fundamental models examine three important aspects of distributed systems: interaction models, which consider the structure and order or sequencing of the communication between the elements of the system; failure models, which consider the ways in which a system may fail to operate correctly and; security models, which consider how the system is protected against attempts to attack it or to steal its data.

1. PHYSICAL MODEL

Physical models consider the types of computers and devices that constitute a distributed system and how they are connected.

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

we can identify physical model of three generations of distributed systems they are,

1.2 Early distributed systems: Such systems emerged in the late 1970s and early 1980s in response to the emergence of local area networking technology, usually Ethernet. These systems typically consisted of between 10 and 100 nodes interconnected by a local area network, with limited Internet connectivity and supported a small range of services such as shared local printers and file servers as well as email and file transfer across the Internet.

1.3 Internet-scale distributed systems: Building on this foundation, larger-scale distributed systems started to emerge in the 1990s in response to the dramatic growth of the Internet during this time. Such systems utilise the infrastructure offered by the Internet to become truly global. They incorporate large numbers of nodes and provide distributed system services for very large companies that span the entire globe. Heterogeneity is high that is distributed system consist of computers with different hardware and software. Hence openness or standardisation was need and middleware's software like RMI was developed to reduce overall software development work.

1.4 Contemporary(current) distributed systems:

In modern distributed systems the physical model has high heterogenity such that each node can be a tiny internet enabled embedded device or desktop and even a super computer. These systems deploy an increasingly varied set of networking technologies and offer a wide variety of applications and services. Such systems potentially involve up to hundreds of thousands of nodes.

1.5 Distributed systems of systems

A future trend is the emergence of ultra-large-scale (ULS) distributed systems . Like internet is a network of networks A system of systems can be defined as a complex system consisting of a series subsystem which are complete distributed systems.

As an example of a system of systems, one system can be a distributed system consisting of group of sensor to monitor weather conditions and another system a distributed system of powerful computers that process data from sensor networks.

2 Architectural Models

Architectural models describe a system in terms of the computational elements(computers) and communications performed by its computational elements

Architectural model consists of

- **core architectural elements or building blocks of distributed systems like communicating entities, communication techniques and architectural styles to build distributed systems**
- **architectural patterns or techniques that can be used in developing more sophisticated distributed systems solutions;**
- **middleware platforms or software to reduce complexity of programming distributed system software using these architectural styles.**

2.1 Architectural elements

To understand the fundamental building blocks of a distributed system, it is necessary to understand

- **the entities that are communicating in the distributed system**
- **communication paradigm or technique used by these entities to communicate**
- **roles and responsibilities of these entities**
- **Placement of these entities**

2.1.1 Communicating entities

The entities that communicate in a distributed system are typically processes which run on different computers like a client process and server process in client server systems. In systems that support threads communication can be done by threads.

From a programming perspective these communicating entities are represented using a programming language feature like an object in Object oriented languages.

Objects: In distributed object-oriented approach, communicating entities are represented as objects. RMI in Java can be used to develop such objects.

Components: Components are like objects and can be used to represent a communication entity. A component usually mentions other components it requires to work correctly and complex distributed systems can be build combining different components. The advantage of a component compared to objects is that we can replace a component with a new one very easily. Java Beans is an example of components based feature.

Web services: Web services based communicating entities used web based technologies to interact. Such communicating entities can be called using a URI(like an URL) and uses XML-based message format to communicate.

Whereas objects and components are often used within an organization, web services are used across organisations where communicating entities belong to different companies and they need to hide their internal hardware and software details.

Communication paradigms

There are three types of communication paradigm or techniques:

- **interprocess communication;**
- **remote invocation;**
- **indirect communication.**

Interprocess communication refers to the relatively low-level communication between processes in distributed systems.

Remote Invocation

Remote invocation represents the most common communication paradigm in distributed systems in which one communicating entities in a distributed system calls a remote operation, procedure or method of another entity.

Request-reply protocols: such protocols typically involve a pairwise exchange of messages from client to server and then from server back to client, with the first message containing the operation to be executed at the server the second message containing any results of the operation. An example is the HTTP protocol.

Remote procedure calls(RPC): In RPC technique, a programmer can invoke procedures(functions) in processes on remote computers can be called as if they are procedures in the local system memory. All the support for making this happen will be done by the RPC module of the corresponding language(rpc.h header file in C language)

Remote method invocation: Remote method invocation (RMI) object oriented version of RPC. With this approach, a calling object can invoke a method in a remote object. RMI package in Java provides support for such communication.

Indirect communication

Key techniques for indirect communication include:

In indirect communication the two communicating entities need not be present at the same time. A third party will store the message and send to the second entity when it is active. Also this communication allows to send messages to a group of receiving entity. The various techniques are,

Group communication: Group communication is concerned with the delivery of messages to

a **set of recipients and hence is a one-to-many communication. Group communication relies on the group identifier.**

Recipients elect to receive messages sent to a group by joining the group. Senders then send messages to the group via the group identifier, and hence do not need to know the recipients of the message. Groups typically also maintain group membership and include mechanisms to deal with failure of group members.

Publish-subscribe systems: In Publish-subscribe systems an intermediary service that efficiently ensures information generated by producers is routed to consumers who has subscribed for this information. Like a Facebook follow.

Message queues: Queues therefore offer an indirection between the producer and consumer processes.

Tuple spaces: Tuple spaces offer a further indirect communication service by supporting a model whereby processes can place structured data, called tuples, in a persistent tuple space(memory space) and other processes can either read or remove such tuples. This is used when readers and writers are not active at the same time.

Distributed shared memory: Distributed shared memory (DSM) systems allows two process in different physical systems to share data as if they had shared physical memory.

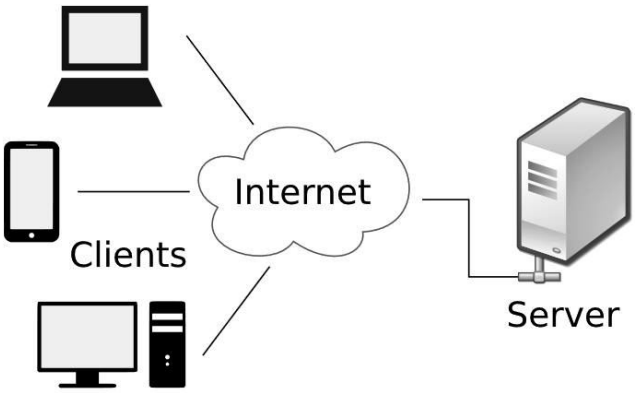
Architectural Styles
We classify architectural styles based on
Roles of communicating entities

Placement of communicating entities in the distributed system

2.1.2 Roles and responsibilities

Client-server:

In such a distributed system there are two roles for communicating entities client and server. client processes interact with individual server processes in separate computers in order to share resources. Web servers usually use client server architecture where browsers are client processes and webservers are the server process. The client process request webpages from servers using HTTP protocol.



Peer-to-peer: In this architecture all of the processes has the same role as peers without any distinction between client and server processes. While the client-server model offers a direct and relatively simple approach to the sharing of data and other resources, it scales poorly. As the number of clients increases the load of server too increases. Also if the server fails entire distributed system fails.

The hardware capacity and operating system functionality of today's desktop computers exceeds that of yesterday's servers, and the majority are equipped with always-on broadband network connections. The aim of the peer-to-peer architecture is to utilise the resources (both data and hardware) in a large number of participating computers. Peer-to-peer applications and systems have been successfully constructed that enable tens or hundreds of thousands of computers to provide access to data and other resources that they collectively store and manage. One of the earliest instances was the Napster application for sharing digital music files.

Figure 2.4a Peer-to-peer arch

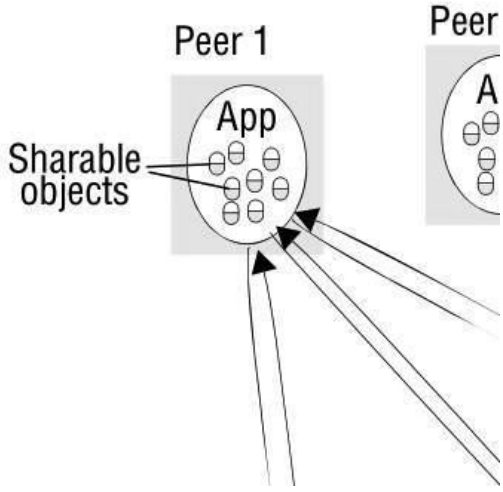


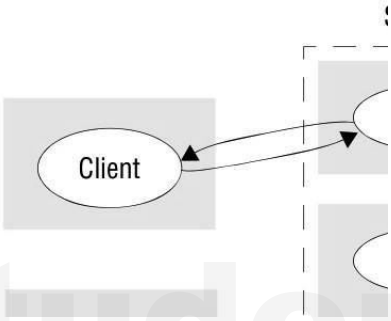
Figure 2.4a illustrates the form of a peer-to-peer application. Applications are composed of large numbers of peer processes running on separate computers. A large number of data objects are shared, an individual computer holds only a small part of the application database, and the storage, processing and communication loads for access to objects are distributed across many computers and network links.

2.1.3 Placement

Mapping of services to multiple servers:

A distributed service may be implemented as a combination of several server processes in separate computers interacting as necessary to provide a service to client processes (Figure 2.4b). The servers may share the set of objects required for the service and distribute those objects between themselves, or they may maintain replicated copies of them on several hosts.

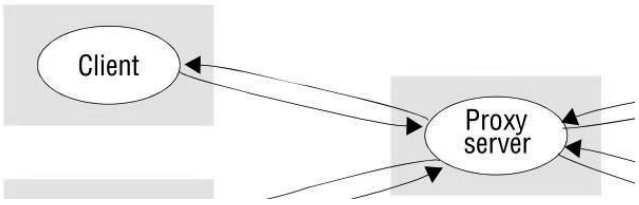
The Web provides a common example of partitioned data in which each web server manages its own set of resources. A user can employ a browser to access a resource at any one of the servers.



Caching: A cache is a storage of recently used data objects(Like in processors). When an object is needed by a client process, the caching service first checks the cache and supplies the object from there if an up-to-date copy is available. If not, an up-to-date copy is fetched. Caches may be co-located with each client or they may be located in a proxy server that can be shared by several clients.

Caches are used extensively in practice. Web proxy servers (Figure 2.5) provide a cache of web pages for the client machines in the local network. Since the proxy server is located in the clients network the copy of webpage can be downloaded from it rather than going to the internet.

Figure 2.5 Web proxy server



Mobile code: **Applets are a well-known and widely used example of mobile code – the user running a browser selects a link to an applet whose code is stored on a web server; the code is downloaded to the browser and runs there. An advantage of running the downloaded code locally is that it is faster and isnot affected by change in the network.**

Mobile agents: **A mobile agent is a running program (including both code and data) that travels from one computer to another in a network carrying out a task on someone's behalf, such as collecting information, and eventually returning with the results.**

Mobile agents might be used to install and maintain software on the computers within an organization or to compare the prices of products from a number of vendors by visiting each vendor's site and performing a series of database operations.

2.2 Architectural patterns

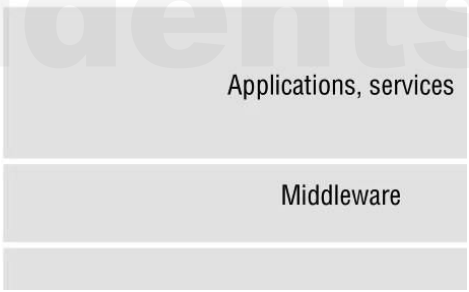
Architectural patterns are techniques or solutions that can be used to build distributed systems.

2.2.1 Layering

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.

Below figure shows the layers of a distributed system,

Figure 2.7 Software and hardware service layers in distributed s



Given the complexity of distributed systems, it is often helpful to organize such services into layers.

- **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 providing support for communication between computers.**

- **Middleware is as a layer of software whose purpose is to mask heterogeneity(differences in hardware and software) and to provide a convenient programming tool to application programmers. Middleware layer implement communication**

and resource-sharing support for distributed applications. It is concerned with providing useful building blocks for the construction of software components that can work with one another in a distributed system. Middleware allows programmers to easily implement remote method invocation; communication between a group of processes; notification of events; the partitioning, placement and retrieval of shared data objects amongst cooperating computers; the replication of shared data objects; and the transmission of multimedia data in real time.

2.2.2 Tiered architecture

There are two types, two-tiered and three-tiered architecture. The tiered architecture consist of,

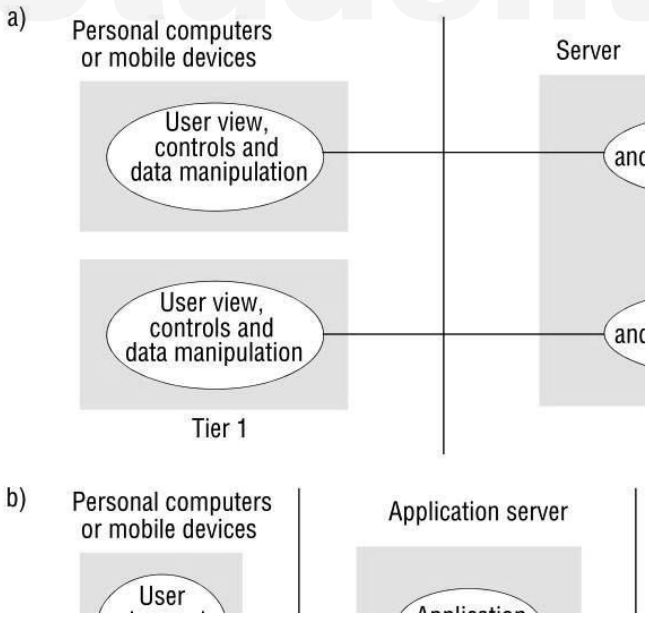
- **the presentation logic, which is concerned with handling user input and displaying the data to the user;**
- **the application logic, which is concerned with the detailed application-specific processing on data associated with the application;**
- **the data logic, which is concerned with the persistent storage of the application data, typically in a database management system.**

This is mainly used in client server architecture.

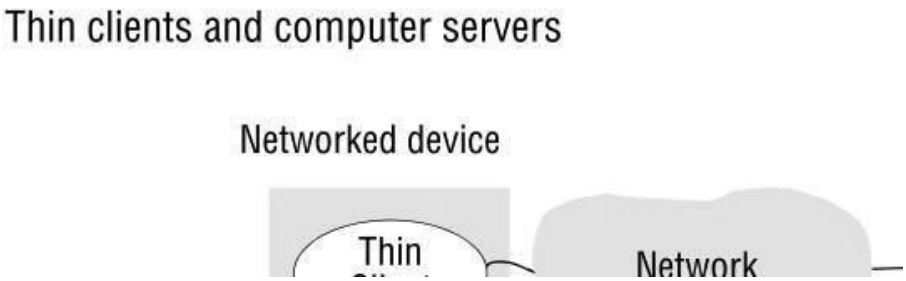
In the two-tier solution, the three logic mentioned above must be placed in two processes, the client and the server. This is most commonly done by placing presentation and part of the application logic in client and the other part of application logic and data logic in the server.

In the three-tier solution, presentation, application and data logic are placed in separate processes.

Two-tier and three-tier architectures



2.2.3 Thin clients • **The trend in distributed computing is towards moving complexity away from the client device towards services in the Internet or the cloud. This trend has given rise to interest in the concept of a thin client which is a dumb device with no processing capacity and only a display, Input-output device and networking support. Thin client has a user interface allowing the user invoke services at the server. For example, a simple phone can access webpages with all the processing to display the web pages is done at the server and needs only a thin web client running on it.**



This concept has led to the emergence of virtual network computing (VNC). VNC allows a user to have access to graphical user interfaces of a remote system. In this solution, a VNC client (or viewer) interacts with a VNC server through a VNC protocol.

2.2.4 Middleware solutions

The task of middleware is to provide a higher-level programming abstraction for the development of distributed systems and, through layering, to abstract over heterogeneity in the underlying infrastructure to promote interoperability and portability.

Figure 2.12 Categories of middleware

Major categories:	Subcategory	Example systems
Distributed objects (Chapters 5, 8)	Standard	RM-ODP
	Platform	CORBA
	Platform	Java RMI
Distributed components (Chapter 8)	Lightweight components	Fractal
	Lightweight components	OpenCOM
	Application servers	SUN EJB
	Application servers	CORBA Component Model
	Application servers	JBoss
Publish-subscribe systems (Chapter 6)	-	CORBA Event Service
	-	Scribe
	-	JMS
Message queues (Chapter 6)	-	Websphere MQ
	-	JMS
Web services (Chapter 9)	Web services	Apache Axis

	Grid services	The Globus Toolkit
Peer-to-peer (Chapter 10)	Routing overlays	Pastry
	Routing overlays	Tapestry
	Application-specific	Squirrel
	Application-specific	OceanStore
	Application-specific	Ivy
	Application-specific	Gnutella

Categories of middleware • **Remote procedure calling packages such as Sun RPC and group communication systems such as ISIS were amongst the earliest instances of middleware**

The top-level categorization of middleware in Figure 2.12 is driven by the choice of communicating entities and associated communication paradigms, and follows five of the main architectural models: distributed objects, distributed components, publish-subscribe systems, message queues and web services. These are supplemented by peer-to-peer systems, a rather separate branch of middleware based on the cooperative approach. The subcategory of distributed components shown as application servers also provides direct support for three-tier architectures. In particular, application servers provide structure to support a separation between application logic and data storage, along with support for other properties such as security and reliability.

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

Limitations of middleware • **Many distributed applications rely entirely on the services provided by middleware to support their needs for communication and data sharing. For example, an application that is suited to the client-server model such as a database of names and addresses, can rely on middleware that provides only remote method invocation.**

When using middleware for communication the programmer implement additional layers for error correction and security of communication which may make the final software more complex compared to if the software didn't use middleware software.

3 Fundamental models

Fundemental model describes the fundamental properties of a distributed system.

Interaction: In distributed system the processes interact and work together by passing messages, resulting in communication and coordination (synchronization) between processes. The interaction model shows that communication takes place with delays and that the accuracy with which independent processes can be coordinated is limited.

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

Security: Distributed systems have threats from both external and internal agents. Our security model defines and classifies such attacks and will help for the design of systems that are able to resist them.

3.1 Interaction model

Fundamentally distributed systems are composed of many processes, interacting together

to perform different operations.
For example:

- **Multiple server processes may cooperate with one another to provide a service; DNS is an example where multiple severs work together to retrieve the IP address of an URL.**

- **A set of peer processes may cooperate with one another to achieve a common goal: for example, a video conferencing system consist of a number of Skype processes in different computers working together**

Most programmers will be familiar with the concept of an algorithm – a sequence of steps to be taken in order to perform a desired computation. Algorithms is executed by a single process. Distributed systems runs algorithms called distributed algorithm where different steps of the algorithm is executed by different processes. To make sure that steps of distributed algorithm are executed in an order messages are transmitted between the processes.

3.1.1 Factors affecting Interaction of processes

Two main factors affect interaction of a process in a distributed system,

- **Communication channel performance.**

- **It is impossible to maintain a single global time for all processes.**

Performance of communication channels

The communication channels in distributed systems can be stream based(e.g TCP) or by simple message passing over a computer network(e.g, UDP).

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

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

- **Bandwidth - The bandwidth of a computer network is the total amount of information that can be transmitted over it in a given time.**

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

Computer clocks and timing events

Each computer in a distributed system has its own internal clock, which can be used by local processes to obtain the value of the current time. Therefore two processes running on different computers can each associate timestamps with their events. However, even if the two processes read their clocks at the same time, their local clocks may supply different time values. This is because computer clocks drift from perfect time and, more importantly, their drift rates differ from one another. The term clock drift rate refers to the rate at which a computer clock deviates from a perfect reference clock. Even if the clocks on all the computers in a distributed system are set to the same time initially, their clocks will eventually vary quite significantly unless corrections are applied.

3.1.2 Two types(variants) of the interaction model

Synchronous distributed systems:

synchronous distributed system is defined as one in which the following bounds(limits) are defined:

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

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

Synchronous distributed systems can be built. What is required is for the processes to perform tasks with known resource requirements for which they can be guaranteed sufficient processor cycles and network capacity, and for processes to be supplied with clocks with bounded drift rates.

Asynchronous distributed systems:

Many distributed systems, such as the Internet, are very useful without being a synchronous system.

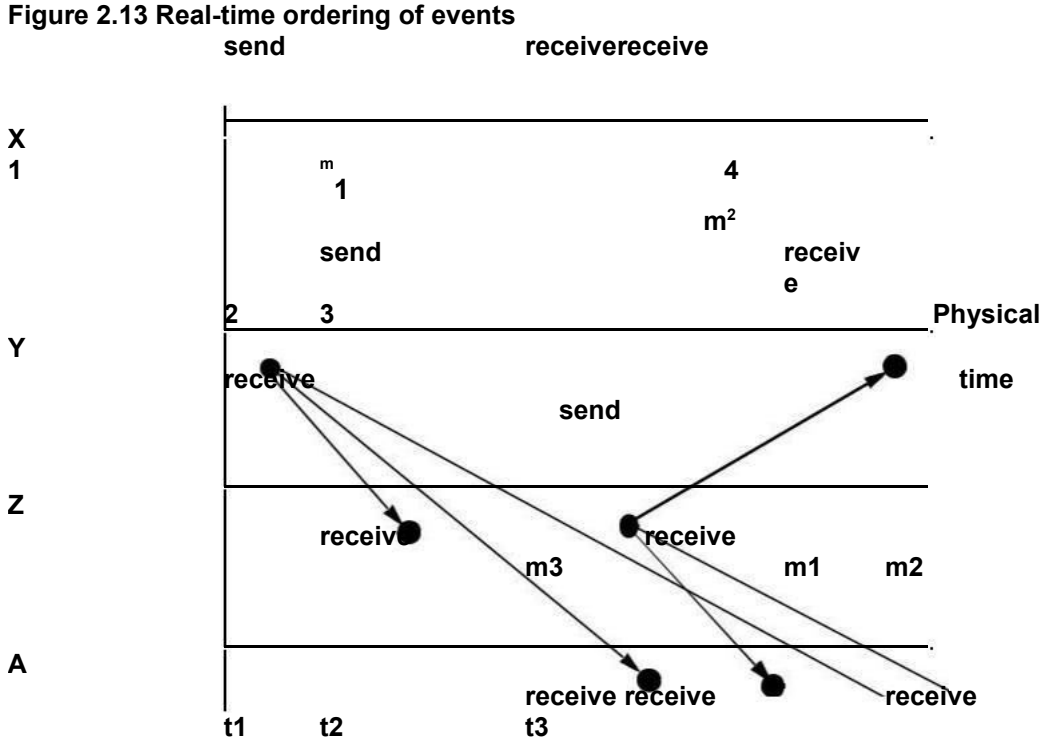
An asynchronous distributed system is one in which there are no bounds(limits) on:

- **Process execution speeds – for example, one process step may take only a picosecond and another step may take a century.**
- **Message transmission delays – for example, one message from process A to process B may be delivered in negligible time and another may take several years.**
- **Clock drift rates are not bounded**

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

Actual distributed systems are very often asynchronous because of the need for processes to share the processors and the network is also shared by multiple processes.

Figure 2.13 Real-time ordering of events



For multimedia distributed systems audio and video streaming they need a synchronous distributed model.

3.1.3 Event ordering

During an interaction between processes, we are interested in knowing whether an event (sending or receiving a message) at one process occurred before, after or concurrently with another event at another process.

If we have a mechanism to order the events then even without a common clock we can coordinate the activities of processes working together.

For example, consider the following set of exchanges between a group of email users,

X, Y, Z and A, on a mailing list:

1. **User X sends a message with the subject Meeting.**

2. **Users Y and Z reply by sending a message with the subject Re: Meeting.**

In real time, X's message is sent first, and Y reads it and replies; Z then reads both X's message and Y's reply and sends another reply, which references both X's and Y's messages. But due to the delays in message delivery, the messages may be delivered as shown in Figure 2.13, and some users may view these two messages in the wrong order. For example, user A might see:

Item	From	Inbox: Subject
23	Z	Re: Meeting
24	X	Meeting
25	Y	Re: Meeting

If the clocks on X's, Y's and Z's computers could be synchronized, then each message could carry the time on the local computer's clock when it was sent. For example, messages m^1, m^2 and m^3 would carry times t^1, t^2 and t^3 where $t^1<t^2<t^3$. The messages received will be displayed to users according to their time ordering. If the clocks are roughly synchronized, then these timestamps will often be in the correct order.

Since clocks cannot be synchronized perfectly across a distributed system, Lamport proposed a model of logical time that can be used to provide an ordering among the events at processes running in different computers in a distributed system.

Logically, we know that a message is received after it was sent. Also only after receiving a message can we sent a reply to it. Using this logic A understand that message from X is the first message as message from Y and Z are replies to the first message and hence the order must be

Item	From	Inbox: Subject
24	X	Meeting
23	Z	Re: Meeting
25	Y	Re: Meeting

Hence without using time we arrived at an ordering of events(order of messages received)

Logical time takes this idea further by assigning a number to each event corresponding to its logical ordering, so that later events have higher numbers than earlier ones. For example, Figure 2.13 shows the numbers 1 to 4 on the events at X and Y.

3.2 Failure model

In a distributed system both processes and communication channels may fail The failure model defines the ways in which failure may occur to understand the effects of failures. We study three main failures,
omission failures,

arbitrary failures and

timing failures.

3.2.1 Omission failures

The faults classified as omission failures refer to cases when a process or communication channel fails.

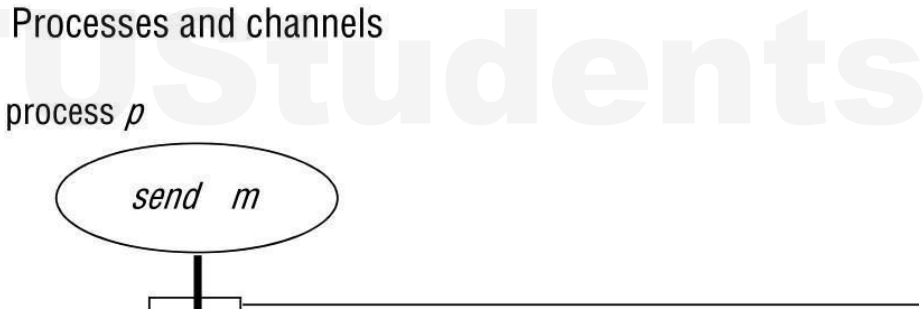
Process omission failures: The omission failure of a process usually means that a process has crashed. When we say that a process has crashed we mean that it has halted and will not execute any further steps of its program ever.

In a synchronous systems other processes can detect a process crash by using timeout that is if a process doesn't reply by a fixed time we can assume that its has stopped working or has crashed.

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

In an asynchronous system a timeout cannot confirm a process has crashed, it may have crashed or may have become very slow.

Communication omission failures: Consider the communication primitives(commands) 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 or at an intervening gateway, or by a network transmission error, detected by a checksum carried with the message data. The loss of messages between the sending process and the outgoing message buffer as send-omission failures, and loss of messages between the incoming message buffer and the receiving process as receive-omission failures, and and loss of messages in between as channel omission failures.

3.2.2 Arbitrary failures

The term arbitrary or Byzantine failure is used to describe the worst possible failure semantics, in which process has not crashed but it is now working correctly and is producing incorrect results. For example, a process may set wrong values in its data items, or it may return a wrong value in response to an invocation.

An arbitrary failure of a process is one in which it arbitrarily omits intended processing steps or takes unintended processing steps. Arbitrary failures in processes cannot be detected as process may reply but it may not have completed the requested action for example deposit an amount to account.

Communication channels can suffer from arbitrary failures; for example, message contents may be corrupted, nonexistent messages may be delivered or real messages may be delivered more than once. Arbitrary failures of communication channels are rare.

Figure 2.15 **Omission and arbitrary failures**

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

because the communication software is able to recognize them and reject the faulty messages. For example, checksums are used to detect corrupted messages, and message sequence numbers can be used to detect non-existent and duplicated messages.

3.2.3 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. Timing failures are listed in Figure 2.16. 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.

Real-time operating systems are designed with a view to providing timing guarantees, but they are more complex to design and may require redundant hardware. Most general-purpose operating systems such as UNIX do not have to meet real-time constraints.

Timing is particularly relevant to multimedia computers with audio and video channels. Video information can require a very large amount of data to be transferred. Delivering such information without timing failures can make very special demands on both the operating system and the communication system.

3.2.4 Masking failures • Each component in a distributed system is generally constructed from a collection of other components. A knowledge of the failure characteristics(how it fails) of a component can allow us to design a distributed system that masks the

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.
Performance	Channel	A message's transmission takes longer than the stated bound.

failure of the components on which it depends.

A service masks a failure either by hiding it altogether or by converting it into a more acceptable type of failure. For an example of the latter, checksums are used to mask corrupted messages(arbitrary communication failure), effectively converting an arbitrary failure into an omission failure. We can mask failures using replication. Even process crashes may be masked, by replacing the process with a backup process and restoring its memory from information stored on disk by its predecessor.

Reliability of one-to-one communication

Although a basic communication channel can exhibit the omission failures described above, it is possible to use it to build a communication service that masks some of those failures.

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

Validity: Any message in the outgoing message buffer is eventually delivered to the incoming message buffer.

Integrity: The message received is identical to one sent, and no messages are delivered twice.

The threats to integrity come from two independent sources:

- **Any protocol that retransmits messages but does not reject a message that arrives twice. Protocols can attach sequence numbers to messages so as to detect those that are delivered twice.**

• **Malicious users that may inject spurious messages, replay old messages or tamper with messages. Security measures can be taken to maintain the integrity property in the face of such attacks.**

3.3 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.

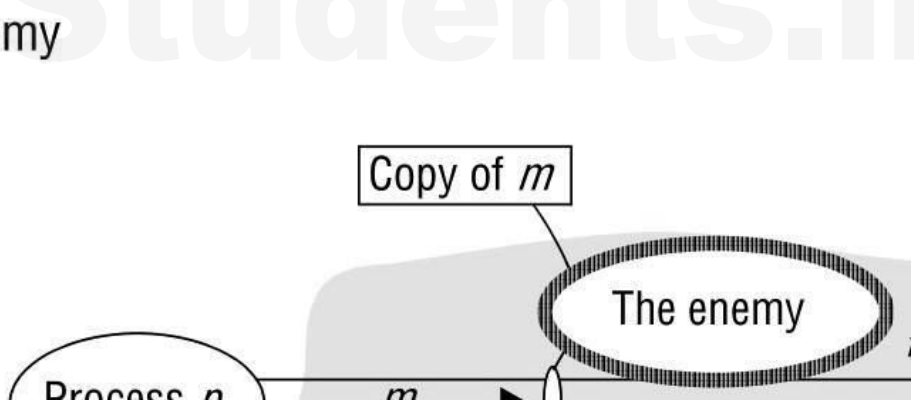
Protection is described in terms of objects, although the concepts apply equally well to resources of all types.

3.3.1 Protecting objects

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(requests) 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 email mailbox, and other objects may hold shared data such as web pages. To support this, access rights specify who is allowed to perform the operations of an object – for example, who is allowed to read or to write its state.

Thus, we must include users in our model as the beneficiaries of access rights. We do so by associating with each invocation and each result the authority on which it is issued. Such an authority is called a principal. A principal may be a user or a process. In our illustration, the invocation comes from a user and the result from a server.



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

3.3.2 Securing processes and their interactions

Processes interact by sending messages. The messages are exposed to attack because the network and the communication service that they use are open, to enable any pair of

processes to interact. Servers and peer processes expose their interfaces, enabling invocations to be sent to them by any other process.

Distributed systems are often deployed and used in tasks that are likely to be subject to external attacks by hostile users. This is especially true for applications that handle financial transactions, confidential or classified information or any other information whose secrecy or integrity is crucial.

The enemy To model security threats, we postulate an enemy (sometimes also known as the adversary) that is capable of sending any message to any process and reading or copying any message sent between a pair of processes, as shown in Figure 2.18. The attack may come from a computer that is legitimately connected to the network or from one that is connected in an unauthorized manner.

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

Threats to processes:

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

Servers: Since a server can receive invocations from many different clients, it cannot necessarily determine the identity of the principal(client) behind any particular invocation. Even if a server requires the inclusion of the principal's identity in each invocation, an enemy might generate an invocation with a false identity. Without reliable knowledge of the sender's identity, a server cannot tell whether to perform the operation or to reject it. For example, a mail server would not know whether the user behind an invocation that requests a mail item from a particular mailbox is allowed to do so or whether it was a request from an enemy.

Clients: When a client receives the result of an invocation from a server, it cannot necessarily tell whether the source of the result message is from the intended server or from an enemy, perhaps 'spoofing' the mail server. Thus the client could receive a result that was unrelated to the original invocation, such as a false mail item (one that is not in the user's mailbox).

Threats to communication channels: An enemy can copy, alter or inject messages as they travel across the network. Such attacks present a threat to the privacy and integrity of information as it travels over the network and to the integrity of the system. For example, a message containing a user's mail item might be revealed to another user or it might be altered to say something quite different.

Another form of attack is the attempt to save copies of messages and to replay them at a later time, making it possible to reuse the same message over and over again. For example, someone could benefit by resending an message requesting a transfer of a sum of money from one bank account to another.

All these threats can be defeated by the use of secure channels or encrypted communication channels.

Other possible threats from an enemy • **Section 1.5.3 introduced very briefly two further security threats – denial of service attacks and the deployment of mobile code. We reiterate these as possible opportunities for the enemy to disrupt the activities of processes:**

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 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. For example, the operation of electronic door locks in a building might be disabled by an attack that saturates the computer controlling the electronic locks with invalid requests.

Mobile code: Mobile code raises new and interesting security problems for any process that receives and executes program code from elsewhere, such as the email attachment. Such code may easily play a Trojan horse role, purporting to 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. The methods by which such attacks might be carried out are many and varied, and the host environment must be very carefully constructed in order to avoid them. Many of these issues have been addressed in Java and other mobile code systems, but the recent history of this topic has included the exposure of some embarrassing weaknesses. This illustrates well the need for rigorous analysis in the design of all secure systems.

3.3.3 Defeating security threats

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. Then if a message exchanged by that pair of processes includes information that proves the sender's knowledge of the shared secret, the recipient knows for sure that the sender was the other process in the pair. Of course, care must be taken to ensure that the shared secret is not revealed to an enemy.

Cryptography is the science of keeping messages secure, and encryption is the process of scrambling a message in such a way as to hide its contents. Modern cryptography is based on encryption algorithms that use secret keys – large numbers that are difficult to guess – to transform data in a manner that can only be reversed with knowledge of the corresponding decryption key.

Authentication: The use of shared secrets and encryption provides the basis for the authentication of messages – proving the identities supplied by their senders. The basic authentication technique is to include with message an encrypted portion. The portion may include requesting principal's identity and the date and time of the request, all encrypted with a secret key shared between the two processes. The server would decrypt this and check that it corresponds to the unencrypted details specified in the request. Here we are trying to find whether the message is fake or not and if portion of message is encrypted by the shared key then we can confirm that sender is true.

Secure channels: Encryption and authentication are used to build secure channels as a layer on top of existing TCP/IP layers. 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. Therefore if a client and server communicate via a secure channel, the server knows the identity of the principal behind the invocations and can check their access rights before performing an operation. This enables the server to protect its**

objects correctly and allows the client to be sure that it is receiving results from a bona fide server.

- **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.**

The uses of security models • **The security model outlined above provides the basis for the analysis and design of secure systems in which these costs are kept to a minimum, but threats to a distributed system arise at many points, and a careful analysis of the threats that might arise from all possible sources in the system's network environment, physical environment and human environment is needed. This analysis involves the construction of a threat model listing all the forms of attack to which the system is exposed and an evaluation of the risks and consequences of each. The effectiveness and the cost of the security techniques needed can then be balanced against the threats.**

DISTRIBUTED COMPUTING

MODULE 3
COMMUNICATION PARADIGMS:
IPC, REMOTE INVOCATION AND INDIRECT COMMUNICATION PARADIGMS

Module 3 - Overview

Communication Paradigms:

- *Inter Process Communication:*
  - IPC Characteristics
  - Multicast Communication
  - Network Virtualization
  - Case study: Skype
- *Indirect Communication:*
  - Group communication
- *Remote Invocation:*
  - Remote Procedure call

Objectives and Outcome

- Examine the different **communication paradigms – IPC, Remote Invocation and Indirect Communication.**
  - To understand the characteristics of inter process communication.
  - To summarize multicast communication based on IP.
  - To explain Network virtualization.
  - To present a case study on 'Skype'.
  - To explain the group communication process.
  - To explain Remote Procedure call (RPC).

Course Outcome:

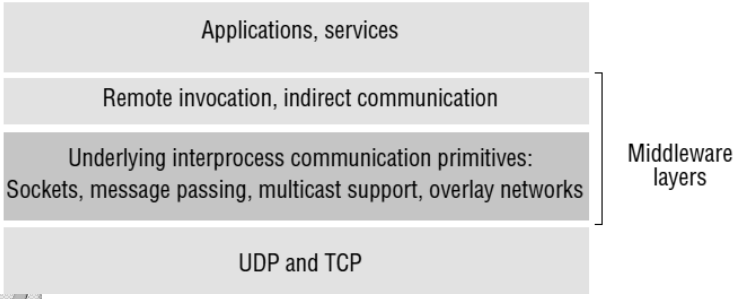
- **CO3: Summarize the mechanisms for inter process communication in a distributed computing system. L2**

CHARACTERISTICS OF IPC

- ✓ General **IPC characteristics.**
- ✓ **Datagram** and **Stream** communication using **UDP** and **TCP.**

Middleware Layers

- Concerned with the communication aspects of middleware:



The Characteristics of IPC

- Message passing between a pair of processes is done by **two** message communication operations:
- **send** and **receive**
  - **A process sends a message** *(a sequence of bytes)* to a destination and **another process at the destination receives** the message.
- This involves
  - the **communication of data** from the sending process to the receiving process and
  - the **synchronization** of the two processes.

Sync. and Async. Communication

- A **queue** is associated with each message destination.
- Sending processes **add messages to remote queues** at receiver.
- Receiving processes **remove messages from local queues.**
- Communication between the sending and receiving processes may be either **synchronous** or **asynchronous.**

Synchronous Communication

- The sending and receiving processes **synchronize at every message.**
- Both **send** and **receive** are **blocking** operations.
- Whenever a send is issued the sending process (or thread) is blocked until the corresponding receive is issued at receiver.
- Whenever a receive is issued by a process (or thread), it blocks until a message arrives.

Asynchronous Communication

- The **send** operation is **non-blocking.**
  - Sending process **proceeds** as soon as the message is copied to a local buffer.
  - The transmission of the message proceeds in parallel with the sending process.
- The **receive** operation can be **blocking/non-blocking.**
  - **Non-blocking** - the receiving process **proceeds** with its program after issuing a receive operation which provides the **buffer to be filled in the background**.
    - » *It must separately **receive notification** by **polling or interrupt** when the buffer has been filled up.*

Asynchronous Communication…

- **In multithreading environment like Java, blocking receive has no disadvantages.**
  - It can be issued by one thread while other threads in the process remain active.
  - It is **simple to synchronize** the receiving threads with the incoming message.
- Non blocking communication seems more efficient, but involves **extra complexity** in the receiving process.
  - The need to acquire the incoming message out of its flow of control.
- So current systems generally do not provide non-blocking receive.

Message Destinations – IP+Port No.

- In the Internet protocols, messages are sent to **(Internet address, local port)** pairs.
  - **IP address** points to the destination computer.
  - **Port no.** is an **integer** that specifies a **message destination within a computer** *(identifies a particular process)*.
    - » Each computer has a large number (**65536** or 2^{16}) of port numbers.
- Any process that know the port can send message to it.
- **Servers** generally **publicize** their port numbers for use by the clients.
- A port has **exactly one receiver,** but can have many senders.
- Processes may use multiple ports to receive messages.

Location Transparency

- If the client uses a **fixed IP** to refer to a service, then that service must **always run on the same computer** for its address to remain valid.
- This can be avoided by using the following approach to providing **location transparency**:
- Client programs **refer to services by names** and use a **name server or binder to translate** their names into server locations *(IP address)* at runtime.
  - *Eg. DNS that translate domain names like 'google.com' to its corresponding IP addresses.*
- This **allows services to be relocated**.
  - but not to migrate (to move while the system is running).

Reliability

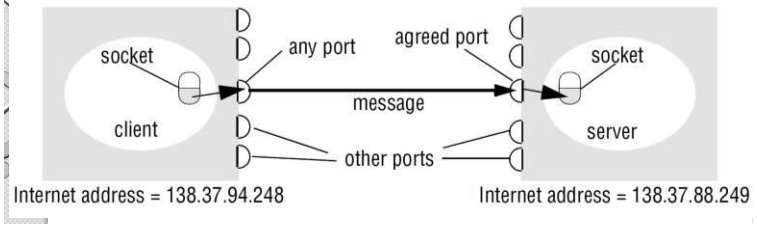
- A reliable communication ensures **validity** and **integrity**:
- **Validity**: A point to point message service is considered to be **reliable** if messages are **guaranteed to be delivered** despite a 'reasonable' no of packets being dropped or lost.
- Considered **unreliable** if messages are **not guaranteed to be delivered** when even a single packet is dropped or lost.
- **Integrity**: messages must arrive **uncorrupted and without duplication.**

Ordering

- **Sender order** delivery:
- Some applications require that messages must be **delivered** in the order in which they were transmitted by the sender.
- The delivery of messages **out of sender order** is regarded as a **failure** by such applications.

Sockets

- Both UDP and TCP uses the socket abstraction.
- A **socket** provides an **endpoint for communication** between processes.
- IPC consists of transmitting a message **between a socket in one process** and **a socket in another process** (*figure*).
- To receive messages, a socket must be **bound to a local port** and to its **IP address**.



Sockets…

- A **socket pair** ***(local IP address, local port, foreign IP address, foreign port)*** uniquely identifies a communication.
- Messages sent to a socket can be received only by a process associated with that IP and port no.
- Processes may use the **same socket for sending and receiving** messages.
- Any process may use **multiple ports** to receive messages.
- But a process **cannot share ports with other processes** on the same computer.
  - ***Processes using IP multicast do share ports.***
- Any number of processes may send messages to the same port.
- Each socket is associated with a particular protocol – either UDP or TCP.

Java API for Internet Addresses

- Java provides a class, ***InetAddress*** that represents Internet Addresses.
- Users of this class **refer to computers by DNS host names** as argument.
- The method uses the domain name to get the correspondingly mapped Internet address.
- Eg: ***InetAddress myIPaddr = InetAddress.getByName("kirk.cs.twsu.edu");***

  Note: kirk.cs.twsu.edu – DNS host name
- ***getByName()*** – get the IP address using hostname.
- This method can throw *UnknownHostException*

UDP DATADRAM COMMUNICATION

Illustrate the communication process using UDP.

API to UDP - Overview

- The application program interface to UDP provides a message passing abstraction – the simplest form of interprocess communication.
- This enables a sending process to transmit a **single message** to a receiving process.
- The **independent packets** containing these messages are called **datagrams**.
- In the Java and UNIX APIs, the sender specifies the destination using a **socket** -
  - an indirect reference to a particular port used by the destination process at a destination computer.

UDP Communication

- A datagram sent by UDP is transmitted **without acknowledgement** or **retries**.
- If **failure** occurs, the **message may not arrive**.
- Datagram is transmitted when one process *sends* it and other process *receives* it.
- To send or receive, a process must first create a socket bound to a local host IP and a local port.
  - A **server** will bind its socket to a **known server port**.
  - A **client** binds its socket to **any free local port**.
- The ***receive()*** method **returns the message** along with the IP address and **port** of the sender.

Issues in UDP Communication

Message size:

- The receiving process needs to specify an **array of bytes** of a particular size in which to receive a message.
- If the array is smaller, the message will be **truncated** on arrival.
- IP protocol allows packet lengths upto **2^{16} bytes** including header and the message.
- The most used size restriction is 8 KB (kilobytes).
- Messages larger then this must be **fragmented** to multiple datagrams.

Issues in UDP Communication…

Blocking:

- Sockets normally provide **non-blocking sends** and **blocking receives** for datagram comm.
- The *send* operation returns once the message is handed over to the underlying UDP and IP protocol.
  - They take care of transmitting to destination.
- On arrival, the message is placed in the queue for the socket that is bound to the destination port.
- The message can be collected from the queue by an outstanding or future invocation of *receive* in that socket.
- Messages are **discarded** if no process already has a socket bound to the destination port.

Issues in UDP Communication…

Blocking:

- The method *receive* blocks until a datagram is received, unless a *timeout* has been set on the socket.
- If the process that invokes the receive method has other work to do while waiting for the message, it should arrange to use a separate thread.
- The receiving thread handle over the work to other threads and wait for the next message from other clients.

Issues in UDP Communication…

Timeouts:

- Receive process that blocks for ever is suitable for use by a server that is waiting to receive requests from its clients.
- But is may not be appropriate in some cases as it may wait indefinitely in case of **crash** or **message loss**.
- For that, in some programs, **timeouts** can be set on sockets.
- It should be fairly **large** in comparison with the time required to transmit a message.

Issues in UDP Communication…

Receive from Any:

- Receive method does not specify an origin for messages, it gets a message addressed to its socket from any origin.
- Receive method **returns the IP address and local port of the sender** to allow the recipient to know where it came fro.
- It is possible to connect a datagram socket to a particular remote port and internet address so that the socket is able to send and receive messages from that address only.

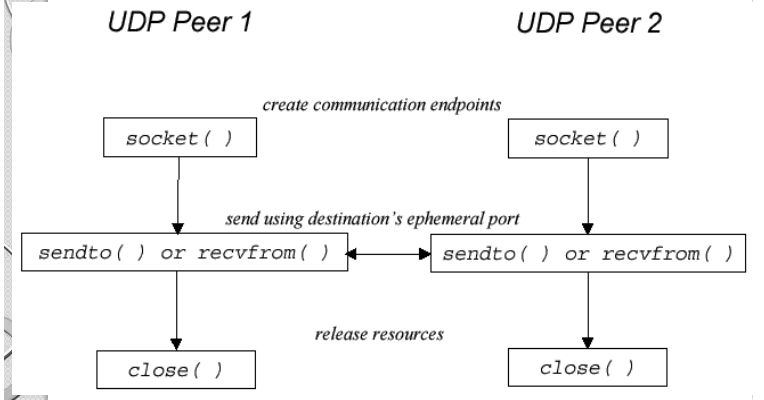
Failure Model for UDP Datagrams

- Failure model defines reliable communication in terms of **integrity** and **validity.** *(not corrupted or duplicated)*
  - Ensured by **checksum** and **sequence no**.
- *UDP datagram suffer from the following failures:*
  - **Omission failures**: Message may be dropped occasionally. Due to checksum error or no buffer space at source or destination.
    - » send-omission and receive-omission failures.
  - **Ordering**: Messages can sometimes be delivered out of sender order.
- A reliable delivery service may be constructed by the use of **acknowledgements**.

Uses of UDP

- UDP can be used in applications which accept occasional omission failures.
- Eg.: **DNS, RIP, traceroute, DHCP, NTP, SNMP, RPC, VoIP**
- UDP is preferred because they do not suffer from the overheads associated with guaranteed message delivery like:
  - the need to store state information at source and destination.
  - the transmission of extra messages.
  - latency for the sender.

Java API for UDP Datagrams



Java API for UDP Datagrams…

- The Java API provides datagram communication by means of **two classes**:
  - ***DatagramPacket*** - used to implement a connectionless packet delivery service.
  - ***DatagramSocket*** - specifies the sending or receiving point for a packet delivery service.
- **Methods of DatagramPacket**:
  - ***getData***() - Returns the data buffer.
  - ***getPort***() - Returns the port number on the remote host.
  - ***getAddress***() - Returns the IP address.

Java UDP API… Datagram Packet

- 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 *IP and local port no* of the destination socket.

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

- Instances of this may be transmitted between processes when *send* and *receive* happens.
- Also provides another constructor for message reception, whose arguments specify an *array of bytes* in which to receive the message and the *length of the array*.
- A received **message** is put in the *DatagramPacket* together with its **length** and the **IP address** and **port** of the sending socket. *(using the methods mentioned before)*

Java UDP API… Datagram Socket

- Support sockets for sending and receiving UDP datagrams.
- Provides a constructor that takes **port no** as argument.
- Also provides a **no argument** constructor that allows the system to **choose a free local port**.
- Throws *SocketException* if the port is already in use or a reserved port (below 1024)
- Provides the following methods:
  - **send** - Sends a datagram packet from this socket.
    - » Arguments include an instance of the datagram packet and its destination

Java UDP API… Datagram Socket…

- **receive** - Receives a datagram packet from this socket.
  - Argument is an empty datagram packet in which to put the message, its length and origin
- **setSoTimeout** - Enable/disable the specified timeout, in milliseconds.
  - The receive method will block for this time and then throw an InterruptedIO Exception
- **connect** - Connects the socket to a remote address for this socket.
  - Then the socket can send and receive messages only from that address

UDP Communication Example:

- ***Client*** *sends a message to server at a port 6789 and then wait for the reply.*
- *Arguments of main() supply a message and DNS hostname of the server.*
- *Message is converted into an array of bytes.*
- *DNS hostname is converted into an Internet address.*
- ***Server*** *creates a socket bound to its server port (6789).*
- *Then repeatedly waits to receive a request message from a client to which it replies by sending back the same message.*

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,  args[0].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

Illustrate the communication process using TCP.

TCP Communication

- The API to the TCP provides the abstraction of a **two way stream of bytes**.
- Data is **written to** and **read from this stream** of bytes, with **no message boundaries**.
- The following network characteristics are hidden by streams:
- **1. Message Sizes** – The application can choose how much (small/large) data it writes to a stream or reads from it.
  - Underlying TCP implementation decides how much data to collect before transmitting it as one or more IP packets.
- **2. Lost Messages** – TCP uses an **acknowledgement** scheme.
  - If the ack is not received within a timeout, it will be resent.
  - Sliding window scheme reduces the no. of acks.

TCP Communication…

- **3. Flow control** – The TCP protocol attempts to match the speeds of the sending and receiving process.
  - If the writer is too fast for the reader, then it will be blocked until the reader has consumed sufficient data.
- **4. Message duplication and ordering** – Message identifiers are associated with each IP packet.
  - It enables the recipient to detect and reject duplicates, or to reorder messages.
- **5. Message destinations -** A pair of processes establish a connection before they communicate over a stream.
  - Then the processes simply read from and write to the stream without using IP address or port nos.

TCP… Establishing a Connection:

- During conn. est., one process act as **client** and the other as **server**, but thereafter they could be **peers**.
- A **connect** request is send from client to server followed by an **accept** message back from server to client, before any communication can take place.
  - *Considerable overhead for a simple client-server req/reply.*
- Client **creates a stream socket bound to any local port** and then makes a **connect request** asking for a connection to a server at its server port.
- The server **creates a listening socket bound to a server port and wait for clients** to request connections.
  - The listening socket maintains a **queue of incoming connection requests**.

TCP… Establishing a Connection: …

- When the server **accepts** a connection, **a new stream socket is created** for the server to communicate with a client.
- Meanwhile, it **retains its socket at the server port for listening for connect requests from other clients.**
- The pair of sockets in the client and server are connected by **a pair of streams**, one in each direction.
- Thus each socket has **an input stream** and **an output stream.**
  - One of the pair of processes can send information to the other by **writing to its output stream**, and the other process obtains the information by **reading from its input stream.**

TCP... Establishing a Connection: ...

- An application **closes** a socket once it is done with the transmission, it **will not write any more data** to its output stream.
- Any data in the output buffer is sent to the other end of the stream and **put in the queue at the destination socket,** with an indication that the stream is broken.
- The destination process can read the data in the queue, but **any further reads** after the queue is empty will result in an indication of **end of stream**.
- When a process **exits or fails,** all of its sockets are eventually closed.
- Any process attempting to communicate with it will discover that its connection has been broken.

TCP... Outstanding Issues:

- The issues related to stream communication are:
- **1. Matching of data items**: Two communicating processes need to **agree on the contents** of the transmitted data.
  - *Eg: If a process writes an int followed by a double to the stream, then the reading process must **read in the same order**, else error in interpreting data occurs, or blocks due to insufficient data.*
- **2. Blocking**: The data written to a stream is kept in a queue at the destination socket.
  - ***Read***: When a process tries to read data from an input stream, it will get the data or block until the data is available.
  - ***Write***: The process that writes data to the stream may be blocked if the socket at the other end is queuing as much data as the protocol allows.

TCP... Outstanding Issues:

- **3. Thread**: When a sever accepts a connection, it generally creates a new thread for the new client.
  - The **advantage** of using a separate thread for each client is that the server can block when waiting for input without delaying other clients.
  - If threads are not provided, an alternative is to test whether input is available from a stream before attempting to read it.

TCP Failure Model

- *TCP create a reliable communication as far as possible:*
- **Checksums and sequence no.** for **integrity**: checksums to detect and reject corrupt packets and seq. nos to detect and reject duplicate packets.
- **Timeouts and retransmission** for **validity**: timeouts and retransmissions to deal with lost packets.
- *ie., messages are guaranteed to be delivered even when some of the underlying packets are lost.*
- But, ***sometimes it is not reliable***, as it does not guarantee to deliver messages in the face of all difficulties.
  - If the network becomes congested and packet loss is too much, no ack is received and then the connection is broken.

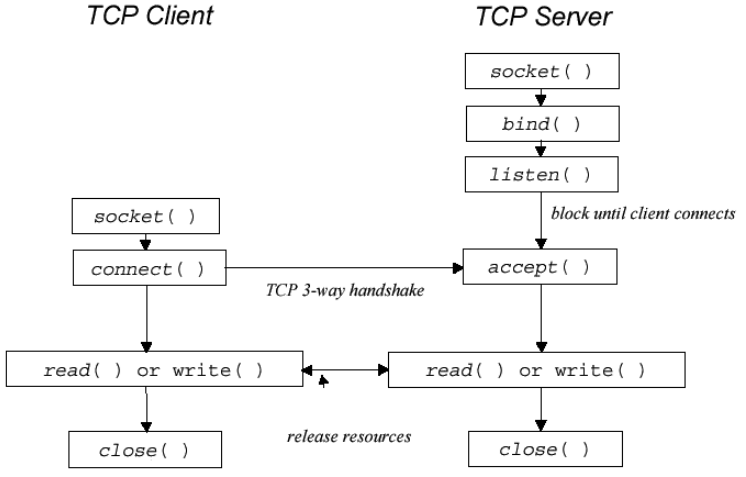
TCP Failure Model…

- When a **connection is broken, a process using it will be notified** if it attempts to read or write.
- It will has the following effects:
  - The processes using the connection cannot distinguish between network failure and failure of the process at the other end of communication.
  - The communication process cannot tell whether their recent messages have been received or not.

Uses of TCP

- Many services use TCP connections with reserved port numbers
  - **HTTP** (Hyper Text Transfer Protocol) is used for communication between web browsers and web servers.
  - **FTP** (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** (Terminal Network) provides access by means of a terminal session to a remote computer.
  - **SMTP** (Simple Mail Transfer Protocol) is used send mail between computers.

Java API for TCP Streams



Java API for TCP Streams…

- *The Java API provides TCP streams by two classes:*
- ***ServerSocket*** – Intended for use by a **server** to create a socket at the server port for listening to ***connect*** requests from clients.

  ServerSocket ss = new ServerSocket(serverPort);
  - 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* class – a socket to use for communicating with the client.

    Socket clientSoc = ss.accept();

Java API for TCP Streams…

- ***Socket*** - This class **represents a connection between a pair of processes**.
- Uses a constructor with host name and port of a server.

  Socket soc = new Socket(serverIP, serverPort);
  - It creates a socket associated with a local port and also connect it to the specified remote computer and port no.
  - Throws *UnknownHostException* if the host name is wrong and *IOException* in case of IO error.
- This class provides two methods for accessing two streams associated with a socket for reading and writing bytes.
  - ***getInputStream*** - Returns an *InputStream* for this socket.
  - ***getOutputStream*** - Returns an *OutputStream* for this socket.

TCP Communication Example:

- **Client Program**:
  - The arguments of main method supply a message and a DNS host name of the server.
  - Client creates a socket bound to the host name and server port 7896.
  - Makes *DataInputStream* and *DataOutputStream* to read and write data.
- **Server Program**:
  - Opens a server socket on its server port 7896 and listens for connect requests.
  - When one arrives, it makes a new thread to communicate with the client.
  - The new thread creates a *DataInputStream* and *DataOutputStream* to read and write data.

TCP Communication Example: …

- ***writeUTF*** / ***readUTF***: as the message is a string, the client and server uses this method to write/read data.
- UTF-8 encoding represents the string in a particular format.
- When a process closes its socket, it will no longer be able to use its input and output streams.
- Existing data can be read from the queue.
- Any further reads after the queue is empty will result in *EOFException*.
- Attempts to use a closed socket to write a broken stream result in an *IOException*.

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
                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){
                System.out.println("close:"+e.getMessage());}}
        }
}
```

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

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


TCP Server …

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


UDP vs TCP

- **Datagram** Socket: uses *datagrams* that may take any route to destination.
- Efficient but suffer from failures – **unreliable**.
- No ack, no retransmission, no seq. no. – out of order delivery.
- **Faster,** as there is no conn. establishment
- No flow control and error checking.

- **Stream** Socket: communication using a *stream* of bytes.
- **Reliable,** but complex.
- Use seq. no., ack, retransmission etc for reliability.
- Uses three way handshake to establish connection - **slower.**
- Flow and error control using sliding window and checksums etc.

MULTICAST COMMUNICATION

A form of inter process communication in which one process in a group of processes **transmits the same message to all members of the group.**

Why Multicast?

- Multicast is an important requirement for distributed applications.
- In distributed systems, a service may implemented as **a number of different processes in different computers,** perhaps to provide fault tolerance or to enhance availability.
- This **distribution is transparent,** a sender may not be knowing the identity of all the receivers.
- The pair-wise exchange of messages is not the best model for communication from one process to a group of other processes in such a scenario.

Multicasting

- A multicast 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.**
  - *Its use in DC: Some services are implemented as a number of different processes in different computers, to provide fault tolerance or to enhance availability. Multicasting suits to pass messages to such services.*
- The simplest multicast protocol provides **no guarantees about message delivery** or **ordering**.

Multicast Characteristics

- Multicast messages provide a useful infrastructure for constructing distributed systems with characteristics like:
  - **Fault tolerance based on replicated services:**
    - » A replicated service consists of a group of servers. Client requests are multicast to a group of servers. Even when some members fail, clients can still be served.
  - **Finding the discovery servers 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.

Multicast Characteristics… contd

- … characteristics:
  - **Better performance through replicated data:**
    - » Data are replicated to increase the performance of a service. Sometimes, 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.
  - **Propagation of event notifications:**
    - » Multicast to a group may be used to **notify** processes when something happens.
    - » *Eg. in Facebook, when someone changes their status, all their friends receive notifications.*
    - » *Similarly, publish-subscribe protocols may make use of group multicast to disseminate events to subscribers.*

IP multicast - An implementation of multicast communication

- IP multicast is built on top of the Internet Protocol (IP).
  - *IP packets are addressed to computers – ports belong to the TCP and UDP levels.*
- IP multicast allows the sender to **transmit a single IP packet to a set of computers** that form a multicast group.
- The sender is **unaware of the identities of the individual recipients** and of **the size of the group**.
- A multicast group is specified by a **class D Internet address**
  - an address whose first 4 bits are 1110 in IPv4.

IP multicast...

- A member of a multicast group can receive IP packets sent to the group.
- The membership of multicast groups is **dynamic,**
  - computers can **join or leave at any time** and
  - can **join an arbitrary number of groups**.
- Can send datagrams to a multicast group **without being a member**.
- At the API level, IP multicast is available only via **UDP**.
  - *A program performs multicast by sending UDP datagrams with multicast addresses and ordinary port numbers.*

IP multicast...

- A program can **join a multicast group** by **making its socket join the group,** enabling it to receive messages to the group.
  - *At the IP level, a computer belongs to a multicast group when one or more of its processes has sockets that belong to that group.*
- When a multicast message arrives at a computer,
  - copies are forwarded to all of the local sockets that have joined the specified multicast address and are bound to the specified port number.

IP multicast... Multicast support in IPv4

- **Multicast Routers** – routers enabled with multicast support.
- IP packets can be multicast both on a local network and on the Internet.
  - **Local multicasts** use the multicast capability of the local network such as an Ethernet *(**hardware multicast** using **MAC address**)*.
  - **Internet Multicast** make use of **multicast routers,** which forward single datagrams to routers on other networks, where they are again multicast to local members.
- The distance of propagation is specified in the Time To Live or **TTL**.

IP multicast... Multicast support in IPv4

Multicast Address Allocation:

- **Class D addresses** (224.0.0.0 to 239.255.255.255) are reserved for multicast traffic and managed globally.
  - *The management of this address space is reviewed annually.*
- Address Space partitioned into a number of blocks:
  - **Local Network Control Block** (224.0.0.0 to 224.0.0.225), for multicast traffic within a given local network.
  - **Internet Control Block** (224.0.1.0 to 224.0.1.225).
  - **Ad Hoc Control Block** (224.0.2.0 to 224.0.255.0), for traffic that does not fit any other block.
  - **Administratively Scoped Block** (239.0.0.0 to 239.255.255.255), which is used to implement a scoping mechanism for multicast traffic (to constrain propagation).

IP multicast... Multicast support in IPv4

Multicast Address Allocation:

- Multicast addresses may be **permanent** or **temporary**.
- **Permanent** groups exist even when there are no members.
  - Addresses are assigned by IANA and span the various blocks.
  - Addresses are **reserved** for a variety of purposes:
    - » for specific Internet protocols or
    - » for organizations that make heavy use of multicast traffic, including multimedia broadcasters and financial institutions.
  - **224.0.1.1 is reserved for the Network Time Protocol (NTP).**
  - Range 224.0.6.000 to 224.0.6.127 in the ad hoc block is reserved for the ISIS project.

IP multicast... Multicast support in IPv4

Multicast Address Allocation:

- The remaining multicast addresses are available for use by **temporary** groups.
- Must be **created before use** and **cease to exist when all the members have left.**
- A temporary multicast group requires a free multicast address to **avoid accidental participation** in an existing group before use and cease after use. The IP multicast protocol does not directly address this issue.
  - For local multicast, **TTL** can be set to a small value, making collisions with other groups unlikely.
  - However, programs using IP multicast throughout the Internet require a more sophisticated solutions for **unique allocations.**

Failure model for multicast datagrams

- IP multicast datagrams suffer from **omission failures.**
  - have the same failure characteristics as UDP datagrams.
- This is called **Unreliable Multicast:**
  - It does not guarantee that a message will be delivered to any member of a group.
  - **Some, but not all** of the members of the group may receive it.

Java API to IP multicast

- The Java API provides a datagram interface to IP multicast through the class *MulticastSocket*.
- **MulticastSocket** - Create a multicast socket.
  - A subclass of UDP *DatagramSocket*, with **additional capabilities for joining multicast groups**.
  - Either bound to a port or any free local port.
- **joinGroup** – method to join a multicast group.
  - it will receive datagrams sent by processes on other computers to that group at that port.
- **leaveGroup** - method to leave a multicast group.
- **setTimeToLive** - Set the default TTL for multicast packets sent out on this MulticastSocket in order to control the scope of the multicasts. **Default is 1**, allowing the multicast to propagate only on the local network.

Java API to IP multicast...

- An application implemented over IP multicast **may use more than one port**.
  - Eg. the **MultiTalk** [mbone] application
    - » allows groups of users to hold textbased conversations, has **one port for sending and receiving data** and **another for exchanging control data**.
- *A multicast peer program shown in next slide specify the message and the multicast address in the arguments to the main method.*
- *When several instances of this program are run simultaneously on different computers, all of them join the same group, and each of them should receive its own message and the messages from those that joined after it.*

Multicast peer joins a group and sends and receives datagrams

```
import java.net.*;
import java.io.*;
public class MulticastPeer{
    public static void main(String args[]){
      // args give message contents & destination multicast group (e.g. "228.5.6.7")
      MulticastSocket s =null;
      try {
        InetAddress group = InetAddress.getByName(args[1]);
        s = new MulticastSocket(6789);
        s.joinGroup(group);
        byte [] m = args[0].getBytes();
        DatagramPacket messageOut =
            new DatagramPacket(m, m.length, group, 6789);
        s.send(messageOut);
        byte[] buffer = new byte[1000];
        for(int i=0; i< 3; i++) {    // get messages from others in group
            DatagramPacket messageIn =
                new DatagramPacket(buffer, buffer.length);
            s.receive(messageIn);
            System.out.println("Received:" + new String(messageIn.getData()));
        }
        s.leaveGroup(group);
      } catch (SocketException e){System.out.println("Socket: " + e.getMessage());
      } catch (IOException e){System.out.println("IO: " + e.getMessage());
      } finally { if(s != null) s.close();}
    }
}
```

Sending message to group

Receiving message from group

Multicast peer joins a group and sends and receives datagrams...

- In the example, the arguments to the main method specify a message to be multicast and the multicast address of a group (for example, "228.5.6.7").
- After joining that multicast group, the process makes an instance of *DatagramPacket* containing the message and sends it through its multicast socket to the multicast group address at port 6789.
- After that, it attempts to receive three multicast messages from its peers via its socket, which also belongs to the group on the same port.
- When several instances of this program are run simultaneously on different computers, all of them join the same group, and each of them should receive its own message and the messages from those that joined after it.

Reliability and Ordering Issues

- The following situations could occur:
  - **Omission failure:**
    - » A datagram sent **from one multicast router to another may be lost,** thus preventing all recipients beyond that router from receiving the message.
    - » Sometimes, **any one of the recipients may drop the message** because its buffer is full.
  - **Process failure** – If a multicast router fails, the group members beyond that router won't receive the message.
  - **Ordering issue** – The packets could arrive in a different order.
    - » some group members receive datagrams from a single sender in a different order from other group members.
    - » messages sent by two different processes will not necessarily arrive in the same order at all the members of the group.

Effects of reliability and ordering issues

- Examples of the effects of reliability and ordering:
- **Fault tolerance based on replicated services**
  - A replicated service consists of the members of a group of servers that start in the same initial state and **always perform the same operations in the same order,** so as to **remain consistent** with one another.
  - In replicated servers, multicast requires that either all or none should receive the request in the same order to perform an operation to remain consistent.
- **Discovering services in spontaneous networking**
  - A process can discover services by multicasting requests at periodic intervals, and the available services will listen for those multicasts and respond.
  - An occasional lost request is not an issue when discovering services.

Effects of reliability and ordering issues...

- Examples of the effects of reliability and ordering:
- **Better performance through replicated data**
  - In cases where the replicated data itself, rather than operations on the data, are distributed by means of multicast messages, the effect of lost messages and inconsistent ordering would depend on the method of replication and the importance of all replicas being totally up-to-date.
- **Propagation of event notifications**
  - The particular application determines the qualities required of multicast.
  - Eg. Jini lookup services use IP multicast to announce their existence.

Reliable IP multicast

- The examples show that some applications require a multicast protocol that is more **reliable than IP multicast.**
- **Reliable multicast:**
  - must ensure that any message transmitted is **either received by all** members of a group **or by none** of them.
- **Totally Ordered multicast:**
  - Some applications have very strict and strong requirements for ordering.
  - **All of the messages** transmitted to a group **reach all of the members** in the **same order.**

Multicast using Overlays

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

NETWORK VIRTUALIZATION

Overlay networks

Network Virtualization: Why?

- IP protocols, through their API, provides a very effective set of building blocks for the construction of distributed software.
- But, Internet is growing with **different classes of applications**, like peer-to-peer file sharing, Skype, etc., which **coexist** in the Internet.
- It is **impractical to alter the Internet Protocols to suit each** of the many applications running over them.
- In addition, the **IP transport service is implemented over a large number of network technologies**.
- These two factors led to the **network virtualization**.

Network Virtualization

- Network virtualization deals with the **construction of many different virtual networks over an existing network** such as the Internet.
- Each virtual network can be designed to **support a particular distributed application**.
  - Eg. One VN might support **multimedia streaming**, as in BBC iPlayer, BoxeeTV [boxee.tv] or Hulu [hulu.com], and
  - **coexist** with another that supports a **multiplayer online game**, both **running over the same underlying network.**
- An application-specific virtual network can **be built above an existing network** and **optimized for that particular application, without changing the characteristics of the underlying network.**

Network Virtualization

- **Network virtualization** (NV) is the ability to create logical, virtual networks that are **decoupled from the underlying network hardware** to ensure that the network can better integrate with and support increasingly virtual environments.
- Computer networks have addressing schemes, protocols and routing algorithms.
- Similarly, **each virtual network has its own particular addressing scheme, protocols and routing algorithms,**
- But these are **redefined** to meet the needs of particular application classes.

Overlay Networks

- 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)* and **offers something that is not otherwise provided.**
  - A service that is **tailored towards the needs of a class of application** or **a particular higher-level service**
    - » *eg. multimedia content distribution.*
  - **More efficient operation** in a given networked environment
    - » *Eg. routing in an ad hoc network*
  - An **additional** feature – like **multicast or secure comm.**
- This leads to a wide variety of **types of overlays.**

Types of Overlay

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,

Advantages of Overlay networks

- **New network services** can be defined without changing the underlying network.
- Encourage **experimentation with network services** and the **customization** of services to particular classes of application.
- **Multiple overlays** can be defined and can **coexist,** resulting in **a more open and extensible network architecture.**

Disadvantages of Overlay networks

- Overlays introduce an **extra level of indirection** and hence may incur a *performance pen*alty.
- They **add to the complexity of network services** when compared to the relatively simple architecture of TCP/IP networks.

Overlay Networks…. contd

- Overlays can be related to the familiar concept of **layers.**
- **Overlays are layers** that **exist outside the standard architecture** *(such as the TCP/IP stack)* and exploit the resultant degrees of freedom.
- Overlay developers are **free to redefine the core elements** of a network, like the *mode of addressing*, the *protocols employed* and the *approach to routing.*
- Introduces radically different approaches more **tailored towards particular application classes** of operating environments.

Overlay Networks…. contd

- Eg., **Distributed Hash Tables:**
  - Introduce a style of **addressing based on a keyspace.**
  - Build a topology in such a way that
    - » **a node** in the topology either **owns the key** or
    - » **has a link to a node that is closer to the *owner***
  - This is a style of routing known as **key-based routing.**
  - Most commonly **ring** topology.
- **Skype** is an example of an overlay network.

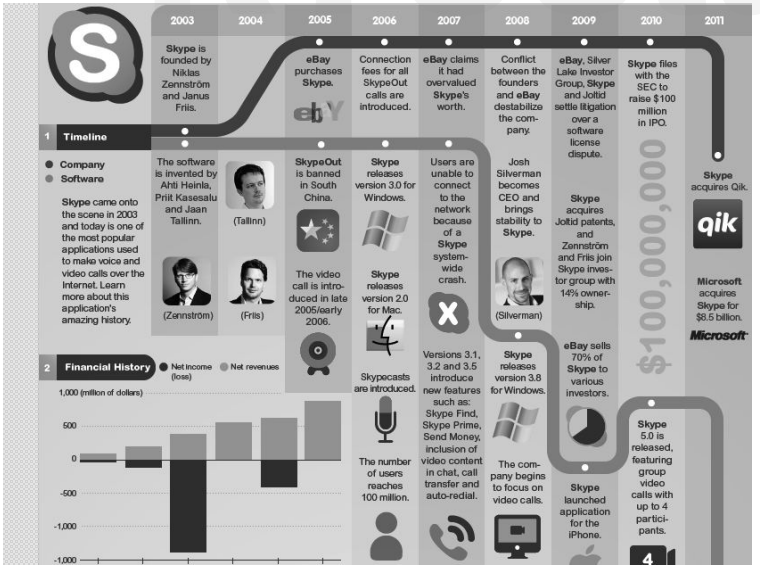
CASE STUDY: skype™

Skype Peer-to-Peer Internet Telephony Protocol

An example of Overlay Network

Skype

- Skype is a **peer-to-peer** application offering Voice over IP (**VoIP**).
- It includes **calling, instant messaging, video conferencing, file sharing** and **interfaces** to the **standard telephony services** through *SkypeIn* and *SkypeOut*.
  - ***SkypeOut***: the ability to **call landline or mobile phones** from Skype; but this term has been dropped later.
  - ***Skype Number*** (until 2010 named ***SkypeIn***) allows a Skype user **to receive calls** to their Skype client *(on whatever device)* dialed from mobiles or landlines to a Skype-provided phone number.
  - Much of the service is **free,** but **Skype Credit** or a subscription is required to call a landline or a mobile phone number.



Skype...

- Skype company was created by Niklas Zennström and the Dane Janus Friis in **2003.**, acquired by **Microsoft** in **2011**.
- Ahti Heinla, Priit Kasesalu and Jaan Tallinn developed the **software**, it was same as of **Kaaza**, a *peer-to-peer music file-sharing application*.
- Skype is widely deployed, with **millions** of users.
- Registered users of Skype are identified by a **unique Skype Name** and may be listed in the **Skype directory**.
- Skype supports conference calls, video chats and screen sharing between **25 people at a time for free**.
- In December 2017, Microsoft added '**Skype Interviews**',
  - a shared code editing system for those wishing to run job interviews for programming roles.

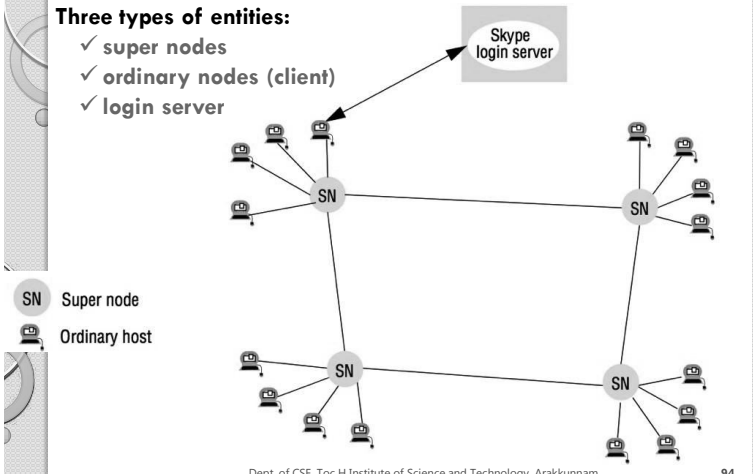
Skype as an Overlay Network

- Skype is an excellent case study of the use of overlay networks in **real-world** and **large-scale** systems.
  - It indicates how advanced functionality can be provided in an **application-specific** manner and **without modifiying the core architecture** of the Internet.
- Skype is a **virtual network** in that it **establishes connections between Skype subscribers** who are currently **active**.
- **No IP address or port** is required to establish a call.
- *The architecture of the Skype virtual network supporting Skype is not widely publicized but researchers have studied Skype through a variety of methods, including **traffic analysis**.*

Skype Overlay Architecture

Three types of entities:

- ✓ **super nodes**
- ✓ **ordinary nodes (client)**
- ✓ **login server**



Skype Architecture

- Skype is based on a **peer-to-peer infrastructure** consisting of **ordinary users' machines** (**hosts**) and **super nodes.**
- **Super Nodes (SN)** are ordinary Skype hosts that happen to have *sufficient capabilities to carry out their enhanced role*.
- They are selected on demand based on a range of criteria:
  - **bandwidth** available,
  - **reachability** (the machine must have a **global IP address** and not be hidden behind a NAT-enabled router) and
  - **availability** (based on the length of time that Skype has been running continuously on that node).

Architecture… Super Nodes

- A super node is an ordinary host's end-point on the Skype network.
- Typically, **super nodes maintain an overlay network among themselves,** while ordinary nodes pick one (or a small number of) super nodes to associate with.
- Super nodes also function as ordinary nodes and are elected from amongst them based on some criteria.
- Ordinary nodes **issue queries through the super nodes** they are associated with.

Architecture… User Connection

- Users are **authenticated** via a well-known **Login Server**.
  - **User names** and **passwords** are stored at the login server to maintain **uniqueness** across Skype namespace.
  - The **buddy list** is also saved here.
  - This is the only **central component** in Skype network.
  - Online and offline user information is stored and propagated in a decentralized fashion.
- Users then make **contact with a selected super node**.
  - To achieve this, each client builds and maintains a **cache of reachable super nodes.**
  - *(that is, IP address and port number pairs).*
  - *This is called **Host Cache**, stored as an **XML file.***

Architecture… User Connection

- At first login, the Skype cache is hardcoded with the address/port pairs of around **seven** super nodes.
- Over time the client **builds and maintains** a much larger set *(perhaps several hundreds)* of online Skype nodes.
- Skype uses **Global indexing,** which is guaranteed to find a user if that user has logged in the Skype network in the last 72 hours.

Architecture… Search for Users

- The main goal of super nodes is to perform the efficient **search of the global index of users,** which is **distributed across the super nodes**.
- The **search** is done by the **client's chosen super node** and involves an expanding search of other super nodes until the specified user is found.
- On average, **eight super nodes** are contacted.
  - A user search typically takes between **3 - 4 seconds** for hosts that have a global IP and slightly longer, 5 - 6 seconds, if behind a NAT-enabled router.
- It appears that **intermediary nodes** involved in the search **cache the results** to improve performance.

Architecture… Voice Connection

- Once the required user is discovered, Skype **establishes a voice connection** between the two parties.
- It uses **TCP** for **signalling call requests and terminations**.
- It uses either **UDP or TCP** for the **streaming audio**.
  - *UDP is preferred but TCP, along with an intermediary node, is used in certain circumstances to circumvent firewalls.*
- The **software** used for **encoding** and **decoding audio** plays a key part in providing **the excellent call quality.**
- The **associated algorithms are carefully tailored** to operate in Internet environments at 32 kbps and above.

Skype Security Policy

- Each caller provides the other with **proof of identity and privileges** whenever a session is established.
- Each verifies the other's proof before the session.
- Uses **256 bit AES** to **encrypt** communication between users.
- Skype's encryption is inherent in the Skype Protocol and is transparent to callers.
- When calling a telephone or mobile**, the part of the call over the PSTN is not encrypted.**
- Skype is **not considered to be a secure VoIP system** as
  - the calls made over the network do not make use of end-to-end encryption, allowing for routine monitoring by Microsoft and by government agencies.

GROUP COMMUNICATION

✓ Examine group communication as an **indirect** communication paradigm.

Indirect Communication

- It is defined as communication between entities in a distributed system **through an intermediary** with no direct coupling between the sender and the receiver(s).
  - ***Space uncoupling*** *– senders may know about the identity of the receivers or vice versa.*
    - » *Participants can be replaced, updated, replicated or migrated.*
  - ***Time uncoupling*** *- the sender and receiver(s) can have* ***independent lifetimes.***
    - » *They need not exist at the same time to communicate.*
    - » *Can have more volatile environments where senders and receivers may come and go.*

Indirect Communication…

- It is often used in distributed systems where **change** is anticipated, like
  - **Mobile environments** where users may rapidly connect to and disconnect from the global network.
  - **Event dissemination** where the receivers may be unknown and liable to change.
- **Group Communication** is an **indirect communication paradigm**.
  - Communication is via a group abstraction with the **sender unaware of the identity of the recipients.**

Space and Time coupling

	Time-coupled	*Time-uncoupled*
Space coupling	*Properties*: Communication directed towards a given receiver or receivers; receiver(s) must exist at that moment in time *Examples*: Message passing, remote invocation (see Chapters 4 and 5)	*Properties*: Communication directed towards a given receiver or receivers; sender(s) and receiver(s) can have independent lifetimes *Examples*: See Exercise 6.3
Space uncoupling	*Properties*: Sender does not need to know the identity of the receiver(s); receiver(s) must exist at that moment in time *Examples*: IP multicast (see Chapter 4)	*Properties*: Sender does not need to know the identity of the receiver(s); sender(s) and receiver(s) can have independent lifetimes *Examples*: Most indirect communication paradigms covered in this chapter

- **Disadvantages**:
  - Inevitable **performance overhead** introduced by the added level of indirection.
  - Such systems are more **difficult to manage** because of the lack of any direct (space or time) coupling.

Space and Time coupling in IP multicast

- **IP multicast** is ***space-uncoupled*** but ***time-coupled***.
  - *space-uncoupled*, because messages are directed towards the multicast group, not any particular receiver.
  - *time-coupled*, though, as all receivers must exist at the time of the message send to receive the multicast.
- Publish-subscribe systems, also fall into this category.
- **Persistency in communication channel** is important to achieve time uncoupling.
  - the communication paradigm must **store messages** so that they can be delivered when the receiver(s) is ready to receive.
- *IP multicast does not support this level of persistency.*

Group Communication

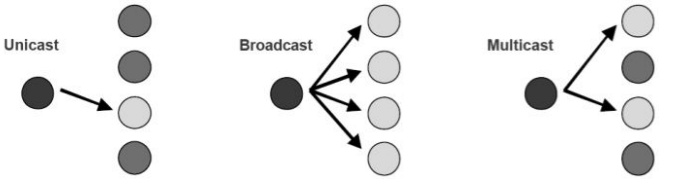
- Group communication offers a service whereby a message is sent to a group and then this message is delivered to all members of the group.
- The sender is not aware of the identities of the receivers.
- It represents an abstraction over multicast comm. and may be implemented over **IP multicast** or an equivalent **overlay network**.
  - adds significant extra value in terms of **managing group membership, detecting failures** and providing **reliability and ordering** guarantees.
- With the added guarantees, group communication is to IP multicast what TCP is to the point-to-point service in IP.

Importance of Group Communication

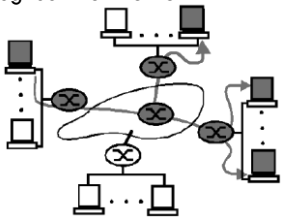
- It is an important **building block** for reliable distributed systems, with key areas of application including:
- **the reliable dissemination of information to potentially large numbers of clients**, including in the financial industry, where institutions require accurate and up-to-date access to a wide variety of information sources.
- **support for collaborative applications, where events must be disseminated** to multiple users to preserve a common user view – eg. in multiuser games.
- **a range of fault-tolerance strategies,** including the consistent update of replicated data or the implementation of highly available (replicated) servers.
- **support for system monitoring and management,** including load balancing strategies.

Group Comm. - Programming Model

- A **group** with associated **group membership,** whereby **processes may join** or **leave the group.**
- Processes can then send a message to this group and have it propagated to all members of the group with certain guarantees in terms of **reliability** and **ordering.**
- Thus, group comm. implements *multicast communication*, in which a message is **sent to all the members** of the group by **a single operation.**
  - A process issues only **one multicast operation** to send a message to each in a group of processes **instead of issuing multiple send operations** to individual processes.



- Communication to a **single process** is known as **unicast.**
- Communication to **all processes** in the system, *(not a subgroup of them)*, is known as **broadcast.**
- **Multicast** allows transmission to a **subset of hosts,** spreading across arbitrary physical networks throughout the internet.
- Subset is known as a **multicast group.**
- A single send operation results in **copies** of the data being delivered to many receivers.
- Copies created at every branching.



Advantages of Multicasting

- **Single multicast operation** instead of multiple send offers much more than a **convenience** for the programmer.
  - *In Java, this operation is aGroup.send(aMessage))*
- **Efficient** in its **utilization of bandwidth.**
- It can take steps to send the message over any communication link, by sending it over a **distribution tree;**
- and it can use network **hardware support** for multicast where this is available.
- Can **minimize the total time taken** to deliver the message to all destinations, as compared with transmitting it separately and **serially.**

Advantages… Eg. Multicasting vs Unicast

- Compare the **bandwidth utilization** and the **total transmission time** when sending the same message from a computer in London to two computers on the same Ethernet in Palo Alto.
- **(a) by two separate UDP sends**
  - **Two copies** of the message are sent **independently,** the second message is **delayed** by the first.
- **(b) by a single IP multicast operation**
  - Here, a set of multicast aware routers forward a **single copy** of the message from London to a router on the destination LAN in California.
  - That router then uses hardware multicast *(provided by the Ethernet)* to deliver the message to both destinations at once, instead of sending it twice.
  - This saves bandwidth and time.

Advantages of Multicasting ...

- **Reliability and Ordering**: The use of a single multicast operation is also important in terms of **delivery guarantees**.
- If a process issues multiple independent send operations to individual processes, there is no way to provide guarantees that affect the group of processes as a whole.
  - If the sender fails halfway through sending, then some members of the group may receive the message while others do not.
  - In addition, the relative ordering of two messages delivered to any two group members is undefined.
- Group communication has the potential to offer a range of guarantees in terms of **reliability** and **ordering**.

Process groups and Object groups

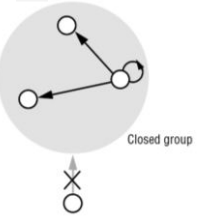
- Group services mostly focuses on **process groups** concept - groups where the communicating **entities are processes**.
- Such services are relatively *low-level* in that:
  - Messages are **delivered to processes** and **no further support for dispatching** is provided.
  - Messages are typically **unstructured** byte arrays with **no support for marshalling** of complex data types.
- The level of service provided is therefore similar to sockets.
- In contrast, **object groups** provide a *higher-level* approach to group computing.

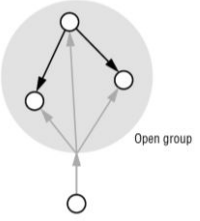
Process groups and Object groups ...

- An **object group** is a collection of objects *(instances of the same class)* that process the same set of invocations concurrently, with each returning responses.
- Client objects need not be aware of the replication.
  - They invoke operations on a single, local object, which acts as a **proxy** for the group.
  - The **proxy uses a group communication** to send the invocations to the members of the object group.
  - Object parameters and results are **marshalled** and the associated calls are dispatched automatically to the right destination objects/methods.
- *However, process groups still dominate in terms of usage.*

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.
  - Eg. for cooperating servers to send messages to one another that only they should receive.





- A group is **open** if processes outside the group may send to it.
  - Eg. for delivering events to groups of interested processes.

Overlapping and Non-overlapping groups

- In **overlapping** groups, entities *(processes or objects)* may be **members of multiple groups**.
- **Non-overlapping** groups imply that **membership does not overlap** (any process belongs to at most one group).
  - *Note that in real-life systems, it is realistic to expect that group membership will overlap.*
- **Synchronous** and **asynchronous** systems:
  - There is a requirement to consider group communication in both environments.
- *The above distinctions can have a significant impact on the underlying multicast algorithms.*

Implementation Issues

- This topic discusses the **properties of the underlying multicast service** in terms of **reliability** and **ordering** and
- also the key role of **group membership management** in **dynamic environments,** where processes can join and leave or fail at any time.

Reliability and ordering in multicast

- In group communication, all members of a group must receive copies of the messages sent to the group, with delivery guarantees.
- The guarantees include agreement on the
  - **set of messages** that every process in the group should **receive** and
  - on the **delivery ordering** across the group members.
- Group communication systems are extremely sophisticated.
- Even IP multicast, which provides minimal delivery guarantees, requires a major engineering effort.

Reliability and ordering in multicast ...

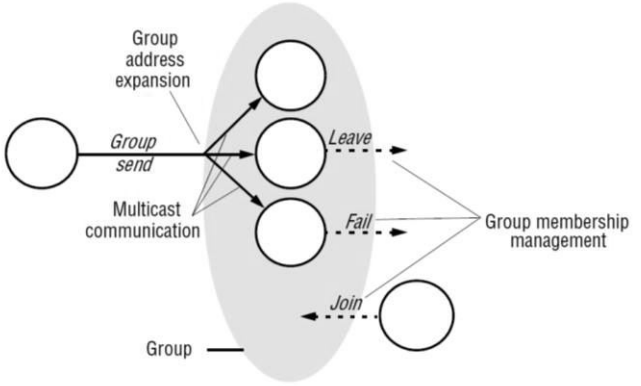
- **Reliability** in group communication - **three properties**:
  - **Integrity** - delivering the message correctly at most once
    - » (the message received is the same as the one sent, and no messages are delivered twice)
  - **Validity** - any outgoing message is eventually delivered.
- To extend the semantics to cover delivery to multiple receivers, a third property is added.
  - **Agreement** - if the message is delivered to one process, then it is **delivered to all processes** in the group.

Reliability and ordering in multicast …

- Group communication demands extra guarantees in terms of the **relative ordering of messages** delivered to multiple destinations.
- Messages may be subject to **arbitrary delays,** to counter this, group comm. services offer **ordered multicast.**
  - **FIFO ordering**: or **source ordering** preserves the order from the perspective of a sender process, if it sends one message before another, it will be delivered in this order at all processes in the group.
  - **Causal ordering**: It considers **causal relationships** between messages, 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 (‘**happens before**’).
  - **Total ordering**: In total ordering, if a message is delivered before another message at one process, then the same order will be preserved at all processes.

Group membership management

- The role of group membership management



Group membership management…

- Fig. has an open group, which shows the key elements of group communication management.
- This illustrates the important role of group membership management in **maintaining an accurate view of the current membership,** given that **entities may join, leave or** indeed **fail.**
- A group membership service has **four main tasks**:
- ***1. Providing an interface for group membership changes:***
  - The membership service provides operations to **create** and **destroy process groups** and to **add** or **withdraw a process** to or from a group.
  - In most systems like IP multicast, a single process may belong to several groups at the same time (**overlapping** groups).

Group membership management…

- ***2. Failure detection:***
  - The service **monitors the group members** in case of **crash,** or they become **unreachable** because of a communication failure.
  - The detector marks processes as **Suspected** or **Unsuspected.**
  - The service **uses the failure detector** to reach a decision about the group’s membership: it **excludes** a process from membership if **it is suspected** to have failed or became unreachable.
- ***3. Notifying members of group membership changes:***
  - The service **notifies** the group’s members when a process is **added,** or when a process is **excluded** *(through failure or when the process is deliberately withdrawn from the group).*

Group membership management…

- ***4. Performing group address expansion:***
  - When a process multicasts a message, it supplies the **group identifier** rather than a list of processes in the group.
  - The membership management service expands the identifier into the current group membership for delivery.
  - The service can **coordinate multicast delivery with membership changes by controlling address expansion**.
  - That is, it can decide consistently where to deliver any given message, even though the membership may be changing during delivery.

Implementation Issues …

- **IP multicast** is a **weak case** of a group membership service, with some but not all of these properties.
- It **does allow processes to join or leave groups dynamically** and
- **it performs address expansion,** so that senders need only provide a single IP multicast address as the destination for a multicast message.
- But IP multicast does not itself provide group members with information about current membership, and multicast delivery is not coordinated with membership changes.
- Achieving these properties is complex and requires what is known as ***view-synchronous group communication***.

Implementation Issues …

- In general, the need to maintain group membership has a significant impact on the utility of group-based approaches.
- Group communication is most effective in small-scale and static systems and does not operate as well in larger-scale or volatile environments.
- This can be traced to the need for a form of synchrony assumption.
- A more probabilistic approach to group membership designed to operate in more large-scale and dynamic environments is using an underlying **gossip protocol**.
- Researchers have also developed group membership protocols specifically for **ad hoc networks** and **mobile environments**.

REMOTE PROCEDURE CALL

✓Illustrate the Remote Procedure Call mechanism
✓Illustrate Sun RPC

Introduction

- Remote Procedure Call (RPC) is a **remote invocation** paradigm.
  - *This concept was first introduced by Birrell and Nelson [1984] and paved the way for many of the developments in distributed systems programming used today.*
- RPC helps achieving a high level of **distribution transparency** in a simple manner,
  - by extending the abstraction of a procedure call to distributed environments,
  - allowing a calling process to **call a procedure in a remote node as if it is local**.

RPC

- Allows client programs to **call procedures transparently in server programs,** running in **separate processes** and generally in **different computers** other than the client.
- The underlying RPC system **hides** important aspects of **distribution,** including
  - the **encoding** and **decoding** of parameters and results,
  - the **passing of messages** and
  - the **preserving** of the required **semantics** for the procedure call.

Design issues for RPC

- Three design issues:
  - 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 RPC

Programming with Interfaces

- Modern programs can be organized as a **set of modules** that can **communicate with one another.**
- Communication can be by **procedure calls** between modules or by **direct access to the variables** in another module.
- To control the possible interactions between modules, an explicit **interface is defined** for each module.
- The interface **specifies the procedures and the variables that can be accessed** from other modules.
- It **hides the module implementation details** from the client.
- So long as its interface remains the same, the **implementation may be changed without affecting the users** of the module.

Interfaces in distributed systems

- In a distr. program, the **modules run in separate processes**.
- A **service interface** is the **specification of the procedures offered by a server**, defining the types of the arguments of each of the procedures.
- **Benefits** of separating interface and implementation:
  - Programmers are concerned only with the interface abstraction and need not be aware of implementation details.
  - Need not know the programming language or platform used to implement the service (managing **heterogeneity**).
  - Implementations can change as long as long as the interface (the external view) remains the same.
    - » The interface can also change as long as it remains compatible with the original.

Interfaces in distributed systems ...

- A client module in one process cannot access the variables of a module in another process. Therefore the service interface **cannot specify direct access to variables**, instead use **getter/setter methods**.
- Local parameter-passing mechanisms like ***call by value* and *call by reference* are not suitable** when the caller and procedure are in different processes. Rather, the procedure description in the interface **describes the parameters as input or output, or sometimes both**.
  - *Input parameters are passed to the remote server by sending the values of the arguments in the request message and then supplying them as arguments to the operation to be executed in the server.*

Interfaces in distributed systems ...

- *Output parameters are returned in the reply message and are used as the result of the call or to replace the values of the corresponding variables in the calling environment.*
- *When a parameter is used for both input and output, the value must be transmitted in both the request and reply messages.*
- **Call by reference is not possible, why?**:
- Another difference between local and remote modules is that **addresses in one process are not valid in a remote one**. Therefore, addresses cannot be passed as arguments or returned as results of calls to remote modules.
- Interface definition languages takes care of these constraints.

Interface Definition Languages (IDL)

- Can **use any programming language** to implement RPC, if it includes an **adequate notation for defining interfaces**.
  - This allows input and output parameters to be mapped onto the language's normal use of parameters.
- Convenient as it allows the programmer to use a single language, eg. Java, for local and remote invocation.
- Many existing useful services are written in C++, etc.
- It would be beneficial to allow programs written in a variety of languages, including Java, to access them remotely.

Interface Definition Languages (IDL)…

- 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 **parameter** is described as for **input or output,** and also its **type.**
- Eg. of a **CORBA IDL**:
  - ***Person*** *structure: the interface* ***PersonList*** *specifies the methods available for RMI in a remote object that implements that interface. The method* ***addPerson*** *specifies its argument as* ***in, an input argument****, and the method* ***getPerson*** *that retrieves an instance of Person by name specifies its second argument as* ***out, an output argument****.*

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();
};
```

CORBA has a struct

remote interface

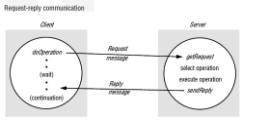
remote interface defines methods for RMI

parameters are *in*, *out* or *inout*

Interface Definition Languages (IDL)…

- **Remote interface:** specifies the methods of an object available for remote invocation
- IDL concept was initially developed for RPC systems but applies to RMI and also web services.
- Some examples of IDL include:
  - **Sun XDR** as an IDL for RPC
  - **CORBA IDL** as an IDL for RMI
  - Web Services Description Language (**WSDL**), which is designed for an Internet-wide RPC supporting web services.
  - **Protocol Buffers** used at Google for storing and interchanging many kinds of structured information.

RPC call semantics



- In Request-reply protocols, the ***doOperation*** can be implemented in different ways to provide different delivery guarantees.
  - **Retry request message:** Controls whether to retransmit the request message until either a reply is received or the server is assumed to have failed (time outs).
  - **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.

RPC call semantics…

- Combinations of these choices lead to different **semantics** for the **reliability** of remote invocations.

Fault tolerance measures			*Call semantics*
Retransmit request message	*Duplicate filtering*	*Re-execute procedure or retransmit reply*	
No	Not applicable	Not applicable	*Maybe*
Yes	No	Re-execute procedure	*At-least-once*
Yes	Yes	Retransmit reply	*At-most-once*

- Fig. shows the choices with the corresponding semantics.
- ***Note**: For **local procedure calls**, the semantics are **exactly once**, meaning that every procedure is executed exactly once (except in the case of process failure).*

RPC call semantics… MayBe Semantics

- Here, the RPC may be executed **once** or **not at all**.
- This happens when no fault-tolerance measures are applied.
- Suffer from the following types of failures:
  - **Omission failures** if the request or result message is lost.
    - » If the request was lost, then the procedure will not have been executed.
    - » If the result was not received after a timeout and there are no retries, it is uncertain whether the procedure has been executed.
    - » Or, the procedure may have been executed and the result was lost.
  - **Crash failures** when the server fails.
    - » A crash may occur either before or after the procedure is executed.
    - » In an asynchronous system, the result of executing the procedure may arrive after the timeout.
- Maybe semantics is useful only for applications in which occasional failed calls are acceptable.

RPC call… At-least-once semantics

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

RPC call… At-least-once semantics…

- An **idempotent** operation is one that can be performed repeatedly with the **same effect** as if it had been performed exactly once.
- **Non-idempotent** operations **can have the wrong effect** if they are performed more than once.
  - Eg. an operation to increase a bank balance by $10 should be performed only once; if it were to be repeated, the balance would grow and grow!
- If the operations can be designed so that all of the procedures in their service interfaces are **idempotent** operations, then at-least-once call semantics may be acceptable.

RPC call... At-most-once semantics...

- Here, the caller **receives either a result**, in which case 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.
- At-most-once semantics can be achieved by using all of the fault-tolerance measures.
- The use of **retries masks any omission failures** of the request or result messages.
- This set of fault tolerance measures **prevents arbitrary failures** by ensuring that a procedure is never executed more than once.
- Sun RPC provides at-least-once call semantics.

Transparency

- Aim was to make RPC as much like local procedure calls as possible, with no distinction in syntax.
- All the necessary calls to marshalling and message-passing procedures were hidden from the caller.
- Retransmission of request messages are transparent to the caller to make the semantics similar.
- RPC strives to offer at least **location and access transparency**,
  - hiding the physical location of the procedure and
  - accessing local and remote procedures in the same way.
- Middleware can also offer additional levels of transparency to RPC.

Transparency...

- RPC calls are **more vulnerable to failure** than local ones, since they involve a **network**, **another computer** and **another process**.
- Sometimes, no result will be received, it is impossible to distinguish between network failure and remote server process failure.
- This requires that clients making remote calls are able to **recover** from such situations.
- The **latency** is several orders of magnitude greater than that of a local one, so minimize remote interactions.
- A caller should be able to abort a remote call that is taking too long in such a way that it has no effect on the server.
- To allow this, the server would need to be able to restore things to how they were before the procedure was called.

Transparency...

- Remote calls require a **different style of parameter passing**.
  - RPC does not offer call by reference.
- Some others claim that the difference between local and remote operations should be expressed at the service interface, to allow participants to react in a consistent way to possible partial failures.
- Some arguments are that the syntax of a remote call should be different from that of a local call: in the case of Argus, the language was extended to make remote operations explicit to the programmer.
- The choice as to whether RPC should be transparent is also available to the designers of IDLs.

Transparency...

- Eg., in some IDLs, a remote invocation may **throw an exception** when the client is unable to communicate with a remote procedure.
  - The client program must be able to handle such exceptions, allowing it to deal with such failures.
- An IDL can provide a facility for **specifying the call semantics** of a procedure, so that the designer can choose the required semantics.
- The current consensus is that remote calls should be made transparent in the sense that
  - the syntax of a remote call is the same as that of a local invocation,
  - but that the difference between local and remote calls should be expressed in their interfaces.

External Data Representation

- The information **stored** in running programs is represented as **data structures**, whereas the information in **messages** consists of **sequences of bytes**.
- The data structures must be **flattened** *(converted to a sequence of bytes)* before transmission and **rebuilt** on arrival.
- Not all computers store primitive values such as integers in the same order. There are two variants:
  - **Big-endian** – the most significant byte comes first
  - **Little-endian** – the most significant byte comes last
- The set of codes used to represent characters:
  - **ASCII** – one byte per character
  - **Unicode** – two bytes per character to present texts in different languages

External Data Representation...

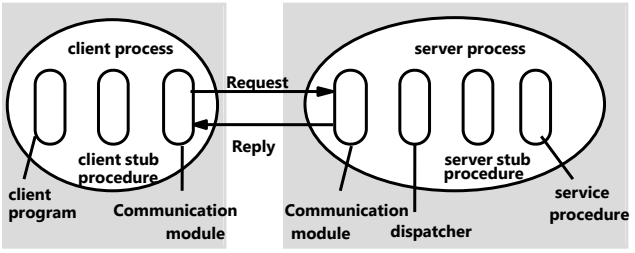
- **Two methods** can be used to exchange binary data values between two systems:
- The values are **converted to a agreed external format** before transmission and back to the local form on receipt.
  - *No conversion needed if the two systems are of same type.*
- The values are transmitted in the **sender's format**, together with **an indication of the format used**, and the recipient converts the values if necessary.
- **An agreed standard for the representation of data structures and primitive values is called an external data representation.**

Marshalling and Unmarshalling

- **Marshalling:**
  - It is the process of taking a collection of data items and assembling them into a form suitable for transmission in a message.
  - It involves data translation to an external data representation.
- **Unmarshalling:**
  - It is the process of disassembling them on arrival to produce an equivalent data items at the destination.
  - It involves generation of primitive values from their external data representation and the rebuilding of the data structures.

Implementation of RPC

- **Building blocks – Client** process and **Server** process
  - **Communication module**
  - **Client stub** procedure: marshalling, sending, unmarshalling
  - **Dispatcher**: will select one of the server stub procedures
  - **Server stub** procedure: unmarshalling, calling, marshalling



Implementation of RPC...

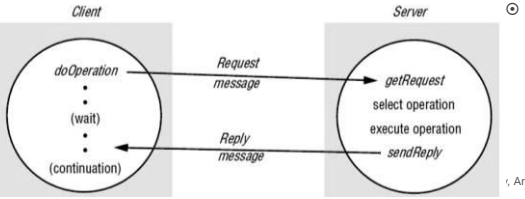
- ***The software components required to implement RPC:***
- The **client** that accesses a service includes **one client stub procedure** for each procedure in the service interface.
- The stub behaves like a local procedure to the client, but instead of executing the call,
  - it **marshals** the **procedure identifier and the arguments** into a **request** message, and
  - it sends via its communication module to the server.
- When the reply message arrives,
  - it **unmarshals** the results.

Implementation of RPC...

- The **server** process contains a **dispatcher**, 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.

Implementation of RPC...

- RPC is generally implemented over a **request-reply protocol**.
- Figure shows this based on three communication primitives :
- ***doOperation, getRequest***, and ***sendReply***
  - The *doOperation* method is used by clients to invoke remote operations.
  - After sending the request, *doOperation* invokes receive to wait for a reply message.
  - *getRequest* is used by a server process to acquire service requests.
  - When the server has invoked the method in the specified object, it then uses *sendReply* to send the reply to the client.



- When the reply is received by the client, the *doOperation* is unblocked.

Implementation of RPC...

- The contents of request and reply messages are:

messageType	*int (0=Request, 1= Reply)*	
requestId	*int*	✓ message identifier: (unique sequential req id + sender id)
remoteReference	*RemoteRef*	✓ remote object reference
operationId	*int or Operation*	✓ the id of the method to be invoked, specified in interface
arguments	*// array of bytes*	✓ parameters to be transmitted.

- Server copies these req ID into the corresponding reply messages.
- RPC can have any one of the invocation **semantics:** *at-least-once* or *at-most-once* is generally chosen.
- *To achieve this, the **communication module** will implement the desired design choices in terms of retransmission of requests, dealing with duplicates and retransmission of results.*

Sun RPC Case Study (RFC 1831)

- Designed for **client-server communication** in the **Sun Network File System (NFS).**
- Also called **ONC** (**Open Network Computing**) RPC.
- Supplied as a part of Sun and other UNIX OS and is also available with NFS installations.
- Can be implemented over either TCP or UDP.
  - With UDP, messages are restricted in length – theoretically to 64 kilobytes, but in practice to 8 or 9 kilobytes.
- It uses **atleast- once** call semantics.
- Broadcast RPC is an option.

Sun RPC Case Study... Sun XDR

- RPC uses an interface definition language called **XDR.**
- Has an **interface compiler** called **rpcgen,** for C lang.
- **Sun XDR** is used to define a service interface for Sun RPC
  - by specifying a set of procedure definitions together with supporting type definitions.
- The notation is rather primitive than CORBA IDL or Java.
- **No Interface name**: Instead, uses a **program number** and **version number,** passed with request message, *so that the client and server can check that they are using the same version.*
  - The program nos can be obtained from a central authority to allow every program to have its own unique number.
  - The version no is intended to be changed when a procedure signature changes.

Sun RPC Case Study... Sun XDR

- A **procedure definition** specifies a procedure signature and a procedure number *(a procedure identifier in request)*
- Only a **single input parameter** is allowed.
  - multiple parameters must be used as components of a single structure.
- The **output parameters** of a procedure are returned via a single result.
- The procedure signature consists of the **result type, procedure name** and the **type of the input parameter.**
  - The type of both the result and the input parameter may specify either a single value or a structure containing several values.

Files Interface in Sun XDR

```
const MAX = 1000;
typedef int FileIdentifier;
typedef int FilePointer;
typedef int Length;

struct Data {
    int length;
    char buffer[MAX];
};
struct writeargs {
    FileIdentifier f;
    FilePointer position;
    Data data;
};
```

```
struct readargs {
    FileIdentifier f;
    FilePointer position;
    Length length;
};

program FILEREADWRITE {
   version VERSION {
     void WRITE(writeargs)=1;   1
     Data READ(readargs)=2;     2
   }=2;
} = 9999;
```


Files Interface in Sun XDR...

- Shows an interface with two procedures - to read and write files.
- Program no: 9999, version no: 2
- Read and write are given numbers 1 and 2
- **READ**: line 2
  - Input parameters: a structure with three components - *file identifier, position in the file, no of bytes required*
  - Result: structure with *no of bytes returned* and *file data*
- **WRITE**: line 1, No result.
- *The number 0 reserved for a null procedure to test whether server is available.*

Files Interface in Sun XDR...

- This IDL provides a notation for defining constants, typedefs, structures, enums, unions and programs.
  - Typedefs, structures and enumerated types use the C language syntax.
- Uses the **interface compiler** **rpcgen** to **generate the RPC components** from an interface definition:
  - client stub procedures
  - server main procedure, dispatcher and server stub procedures
  - XDR marshalling and unmarshalling procedures for use by the dispatcher and client / server stub procedures

Sun RPC Case Study... Binding

- Sun RPC runs a local binding service called the **port mapper** at a well-known port no. on each computer.
  - *Port mapper records the program no., version no. and port no. in use by each service running locally.*
- When a server starts up it registers its program no., version no. and port no. with the local port mapper.
- When a client starts up, it finds out the server's port by making a remote request to the port mapper at the server's host, specifying the program no. and version no.
- Multiple instances of a service may use different port nos for receiving client requests.

Sun RPC Case Study... Binding...

- A client cannot use a direct IP multicast message to multicast a request to all the instances of a service that are using different port numbers.
- The solution is that clients make **multicast RPC** by **multicasting them to all the port mappers,** specifying the program and version number.
- Each port mapper forwards all such calls to the appropriate local service program, if there is one.

Sun RPC Case Study... Authentication

- RPC request and reply messages has fields for authentication between client and server.
- The request contains the **credentials of the client.**
  - Eg. the uid and gid of the user.
- **Access control** mechanisms can be made available to the server procedures via a second argument.
- The server is enforces access control by deciding whether to execute each procedure call according to the authentication information.
  - Eg., if the server is an NFS file server, it can check whether the user has sufficient rights to do a file operation.

Sun RPC Case Study... Authentication...

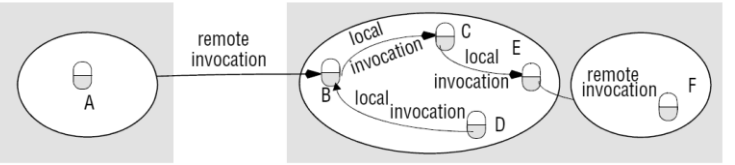
- Several different authentication protocols can be supported.
  - No authentication.
  - UNIX style.
  - A style in which a shared key is established for signing the RPC messages.
  - Kerberos - using tokens.
- A field in the RPC header indicates which style is being used.

RPC vs RMI

- **Remote procedure call (RPC)**
  - Allows client programs to **call procedures** in server programs running on different processes or machines.
- RPC's Service interface:
  - **Specification of the procedures** of the server, defining the types of the input and output arguments of each procedure.
- **Remote method invocation (RMI)**
  - Allows an object living in one process to **invoke the methods of an object** living in other process.
- RMI's Remote interface:
  - **Specification of the methods** of an object that are available for objects in other processes, defining the types of them.
  - May pass objects or remote object references as arguments or returned result.

Local and Remote Method Invocations

- Object oriented program consists of a collection of interacting **objects, encapsulating** of ***data*** *and* ***a set of methods***.
- Object communicates with other objects by invoking their methods by passing arguments and receiving results.
- In a distributed environment, the objects may be physically distributed in **different processes** or **computers**.
- For a distributed system, the objects data should be accessed only via its methods.
- Method invocations between objects in **different processes** are known as ***remote method invocations***.
- Otherwise ***local method invocations*** (objects in **same process**).
  - Has exactly once semantics.



- Objects that can receive remote invocations are called ***remote objects***, *eg. B & F*
- All objects can receive local invocations from other objects that hold references to them. *(C must have reference to E)*.
- **Remote Object Reference:**
  - Other objects can invoke the methods of a remote object if they have access to its remote object reference
  - *Eg. Remote obj ref to B must be available to A*
- **Remote Interface:**
  - Specifies the methods of an object that are available for invocation by remote objects in other processes.
  - *Eg. B and F must have remote interfaces*

Summary

Learnt the different Communication Paradigms:

- Inter process communication:
  - IPC Characteristics
  - Multicast Communication
  - Network Virtualization
  - Case study: Skype
- Indirect communication:
  - Group communication
- Remote Invocation:
  - Remote Procedure call

Text Books

- 1. George Coulouris, Jean Dollimore and Tim Kindberg , Distributed Systems: Concepts and Design, Fifth Edition, Pearson Education, 2011
- Website: http://www.cdk5.net/wp/
- 2. Pradeep K Sinha, Distributed Operating Systems: Concepts and Design, Prentice Hall of India

References:

- 1. A S Tanenbaum and M V Steen , Distributed Systems: Principles and paradigms, Pearson Education, 2007
- 2. M Solomon and J Krammer, Distributed Systems and Computer Networks, PHI

DISTRIBUTED COMPUTING

MODULE 4
DISTRIBUTED FILE SYSTEMS

Module 4 - Overview

- Distributed file system (DFS):
  - File service architecture
  - Network File System (Sun's NFS)
  - Andrew File System (AFS)
- Name Services

Objectives and Outcome

- To distinguish between the characteristics of local file systems and distributed file systems (DFS).
- To summarize the DFS requirements.
- To describe the file service architecture and implementation of DFS.
- To outline how a relatively simple, widely-used service like Sun NFS is designed.
- To understand how Andrew file system is designed.
- To explain name services.

COURSE OUTCOME:

- **CO4: Identify appropriate distributed system principles in ensuring transparency, consistency and fault-tolerance in distributed file systems. L3**

INTRODUCTION TO DISTRIBUTED FILE SYSTEM

- ✓ Distributed File System
- ✓ DFS Characteristics
- ✓ DFS Requirements

Introduction

- The **sharing of stored information** is the most important aspect of *distributed resource sharing*.
  - *Eg. Web servers provide a restricted data sharing in which files stored locally, in file systems at the server, are made available to clients throughout the Internet.*
- The requirements for sharing within local networks and intranets lead to a service that supports the **persistent storage of data and programs.**

File Systems

- File systems were originally developed for centralized computer systems and desktop computers as an OS facility.
- It provides a **convenient programming interface to disk storage** *(persistent local storage)*.
- **Access-control** and **file-locking** mechanisms made them useful for the sharing of data and programs.

Distributed File Systems (DFS)

- A DFS enables programs to **store and access remote files exactly as they do local ones,** allowing users to access files from any computer on a network.
  - It supports the **sharing of information** in the form of **files** and **hardware resources** in the form of **persistent storage** throughout an intranet.
- The **performance** and **reliability** experienced for access to files stored at a server should be comparable to that for files stored on local disks.
- Most effective in providing **shared persistent storage** for use in intranets.
- Eg. Sun NFS

File Service

- A **file service** enables programs to **store and access remote files exactly as they do local ones**, allowing users to access their files from any computer in an intranet.
- The concentration of **persistent storage at a few servers** reduces the need for local disk storage.
- The web servers can **store and access** the material from a **local distributed file system**.

Storage Systems and their properties

- Advent of distributed object oriented programming led to the need for **persistent storage** and **distribution of shared objects.**
  - Serializing objects is impractical for rapidly changing objects.
- Java RMI & CORBA ORB provide access to remote shared objects.
  - But offers **no persistence** of the objects and **replication.**
- **Distributed shared memory** (DSM): emulation of shared memory by replication of memory pages at each host.
  - **No automatic persistence**
- **Persistent object stores:** offers persistence for distributed shared objects
  - Eg: CORBA Persistent State Service and persistent extensions to Java
- **PerDiS** and **Khazana**: support the **automatic replication and persistent storage** of objects.

Storage Systems and their properties...

	Sharing	*Persistence*	*Distributed cache/replicas*	*Consistency maintenance*	*Example*
Main memory	✗	✗	✗	1	RAM
File system	✗	✓	✗	1	UNIX file system
Distributed file system	✓	✓	✓	✓	Sun NFS
Web	✓	✓	✓	✗	Web server
Distributed shared memory	✓	✗	✓	✓	Ivy (DSM, Ch. 6)
Remote objects (RMI/ORB)	✓	✗	✗	1	CORBA
Persistent object store	✓	✓	✗	1	CORBA Persistent State Service
Peer-to-peer storage system	✓	✓	✓	2	OceanStore (Ch. 10)

Types of consistency:

1: strict one-copy ✓: slightly weaker guarantees 2: considerably weaker guarantees

Storage Systems and their properties...

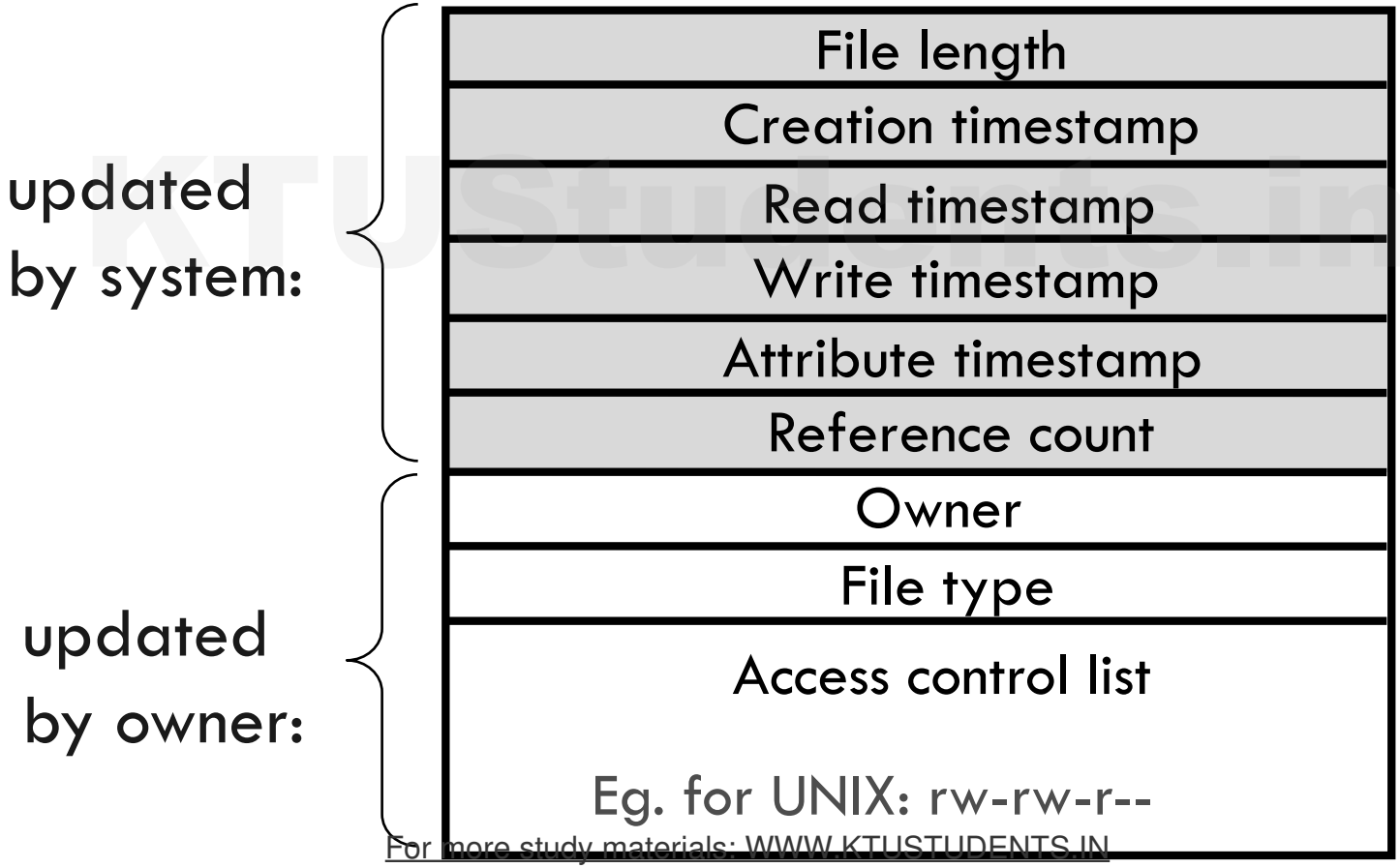
- Virtually all storage systems rely on the use of **caching** to optimize the performance of programs.
- Consistency between the copies stored at web proxy/client cache with the original server is maintained by explicit user actions.
- The consistency column indicates whether multiple copies are kept consistent when updates occur.
  - **Strict consistency** (**'1'** *for* ***one copy consistency***): Programs cannot observe any discrepancies between cached copies and stored data after an update *(as in in centralized systems)*.
  - **Specific consistency** (√): To maintain an approximation to strict consistency in distributed environment like NFS and Ivy.
- When distr. replicas are used, strict consistency is difficult to achieve.
- DFS such as Sun NFS and the AFS cache copies of portions of files at client computers, and they adopt specific consistency mechanisms to maintain an approximation to strict consistency – indicated by a **tick**.

Characteristics of File Systems

- **File System Responsibilities:**
  - **Organization, storage, retrieval, naming, sharing and protection of files.**
  - Provide a **programming interface** freeing the programmers from the details of storage allocation and layout.
- **File:**
  - Files are stored on disks or other non-volatile storage media.
  - Include data and attributes
    - » **Data**: Sequence of data items, can ***read*** and ***write*** data.
    - » **Attributes** is a single record consisting of details of files.

File Attributes and Record Structure

- Record structure has attribute like length of the file, timestamps, file type, owner's identity, access control lists etc.
- The shaded attributes are managed by the file system and are not normally updatable by user programs.



Characteristics of File Systems…

- File systems are designed to store and manage large no of files – **create, delete** and **naming** of files
- Naming is supported by directories
- **Directory:**
  - A **special file** that provides a mapping from text names to internal file identifiers.
  - Also include **names of other directories** for hierarchical file scheme.
- **File systems** also control **access** to **files, restricting access** according to users' **authorizations** and the **type** of **access** requested ***(read, write, execute)*.**

Characteristics of File Systems…

- **Metadata:**
  - **Extra information** stored by the files.
  - Needed for the **management of files.**
  - It includes attribute, directories and all the other persistent information used by the file system.

- **File System Architecture**
  - Layered structure.
  - Each layer depends on the layer below it.

File System Modules

- Figure shows a typical layered module structure for the implementation of a non-distributed file system in a conventional operating system.

Directory module:	relates file names to file IDs
File module:	relates file IDs to particular files
Access control module:	checks permission for operation requested
File access module:	reads or writes file data or attributes
Block module:	accesses and allocates disk blocks
Device module:	performs disk I/O and buffering

Characteristics of File Systems…

- **File system operations:**
  - Applications access the operations on files using **system calls** on **kernal** via library procedures.
  - In UNIX operations, some **file state information** is stored by the file system for each running program.
    - » Consists of **a list of currently opened files** with a **read-write pointer** to each, giving the **position** within the file at which the **next read or write** operation will be applied.
  - Unix file system also applies **access control** for files – by checking the **user's rights** and using the **mode** of access requested.

Unix File System Operations

filedes = open(name, mode)	Opens an existing file with the given *name*.
filedes = creat(name, mode)	Creates a new file with the given *name*. Both operations deliver a file descriptor referencing the open file. The *mode* is *read*, *write* or both.
status = close(filedes)	Closes the open file *filedes*.
count = read(filedes, buffer, n)	Transfers *n* bytes from the file referenced by *filedes* to *buffer*.
count = write(filedes, buffer, n)	Transfers *n* bytes to the file referenced by *filedes* from *buffer*. Both operations deliver the number of bytes actually transferred and advance the read-write pointer.
pos = lseek(filedes, offset, whence)	Moves the read-write pointer to *offset* (relative or absolute, depending on *whence*).
status = unlink(name)	Removes the file *name* from the directory structure. If the file has no other names, it is deleted.
status = link(name1, name2)	Adds a new name (*name2*) for a file (*name1*).
status = stat(name, buffer)	Puts the file attributes for file *name* into *buffer*.

Distributed File System Requirements

Efficiency

Goal for distributed file systems is usually performance comparable to local file system.

Distributed File System Requirements…

- Transparency:
  - access
  - location
  - mobility
  - performance
  - scaling
- Concurrent file updates
- File replication
- Consistency
- Fault tolerance
- Hardware and OS heterogeneity
- Security
- Efficiency

File service is most heavily loaded service in an intranet, so its functionality and performance are critical

DFS Requirements ...

- **Transparencies:** The design of the file service should support many of the transparency requirements for DS.
- *The following forms of transparency are partially or wholly addressed by current file services:*
  - **Access:** Client programs should be unaware of the distribution of files. **Single set of operations for accessing** local and remote files. Programs written to operate on local files are able to access remote files without modification.
  - **Location:** Client programs should **see a uniform file name space**. Files or groups of files may be relocated without changing their pathnames, and user programs see the same name space wherever they are executed.

DFS Requirements ...

- **Transparencies:**
  - **Mobility: Automatic relocation** of files must be possible. Neither client programs nor system admin tables in client nodes need to be changed when files are moved. This allows file mobility, either by system administrators or automatically.
  - **Performance:** Satisfactory performance of client programs across a specified range of system loads.
  - **Scaling:** Service can be expanded by incremental growth to deal with additional loads and network sizes.

DFS Requirements ...

- **Concurrent file updates:**
  - Changes to a file by one client **should not interfere** with the operation of other clients simultaneously accessing or changing the same file - **concurrency control**.
  - Concurrency control for access to shared data is **costly** to implement.
  - Follow UNIX standards in providing advisory or mandatory file or record-level locking.

DFS Requirements ...

- **File Replication:**
  - File service maintains **multiple** identical **copies** of files at different locations.

  ***Two Advantages*:**
  - Enables **load-sharing** between multiple servers making service more **scalable**.
  - Enhances **fault tolerance** by enabling clients to locate another server that holds a copy of the file when one has failed.
- Local access has better response (lower latency).
- Full replication is difficult to implement.
- **Caching**, *a limited form of replication,* of all or part of a file gives most of the benefits.

DFS Requirements ...

- **Hardware and OS Heterogeneity:**
  - Service can be accessed by clients running on (almost) any OS or hardware platform.
  - Design must be compatible with the file systems of different OSes.
  - **Service interface** should be defined so that client and server software can be implemented for different OSs and computers – **openess.**

DFS Requirements ...

- **Fault tolerance:**
  - Service must **continue to operate** in the face of client and server failures. A moderately fault-tolerant design is straightforward for simple servers.
  - To cope with comm. failures, *at-most-once* semantics or *at-least-once* semantics with *idempotent* operations can be used so that that duplicated requests do not result in invalid updates to files.
  - The servers can be **stateless,** so that they can be restarted and the service restored after a failure without any need to recover previous state.
  - Tolerance of disconnection or server failures requires **file replication,** which is more difficult to achieve. If the service is replicated, it can continue to operate even during a server crash.

DFS Requirements ...

- **Consistency**
  - Unix offers **one-copy update semantics** for operations on local files .
  - This is a model for concurrent access to files in which the file contents seen by all of the processes accessing or updating a given file are those that they would see if only a single copy of the file contents existed.
  - When files are replicated or cached at different sites, there is a **delay** in the propagation of modifications made at one site to all of the other sites that hold copies, and this may result in some deviation from one-copy semantics.

DFS Requirements ...

- **Security:**
  - Must maintain access control based on access control lists.
  - In DFS, client requests must be authenticated so that access control at server is based on user id.
  - protect the contents of request and reply messages with digital signatures and encryption of secret data.
  - Vulnerable to impersonation and other attacks
- **Efficiency:**
  - Goal for distributed file service is usually the performance comparable with or better than local file system.
  - It must be convenient to administer, providing operations and tools that enable system administrators to install and operate the system conveniently.

FILE SERVICE ARCHITECTURE

Three components:

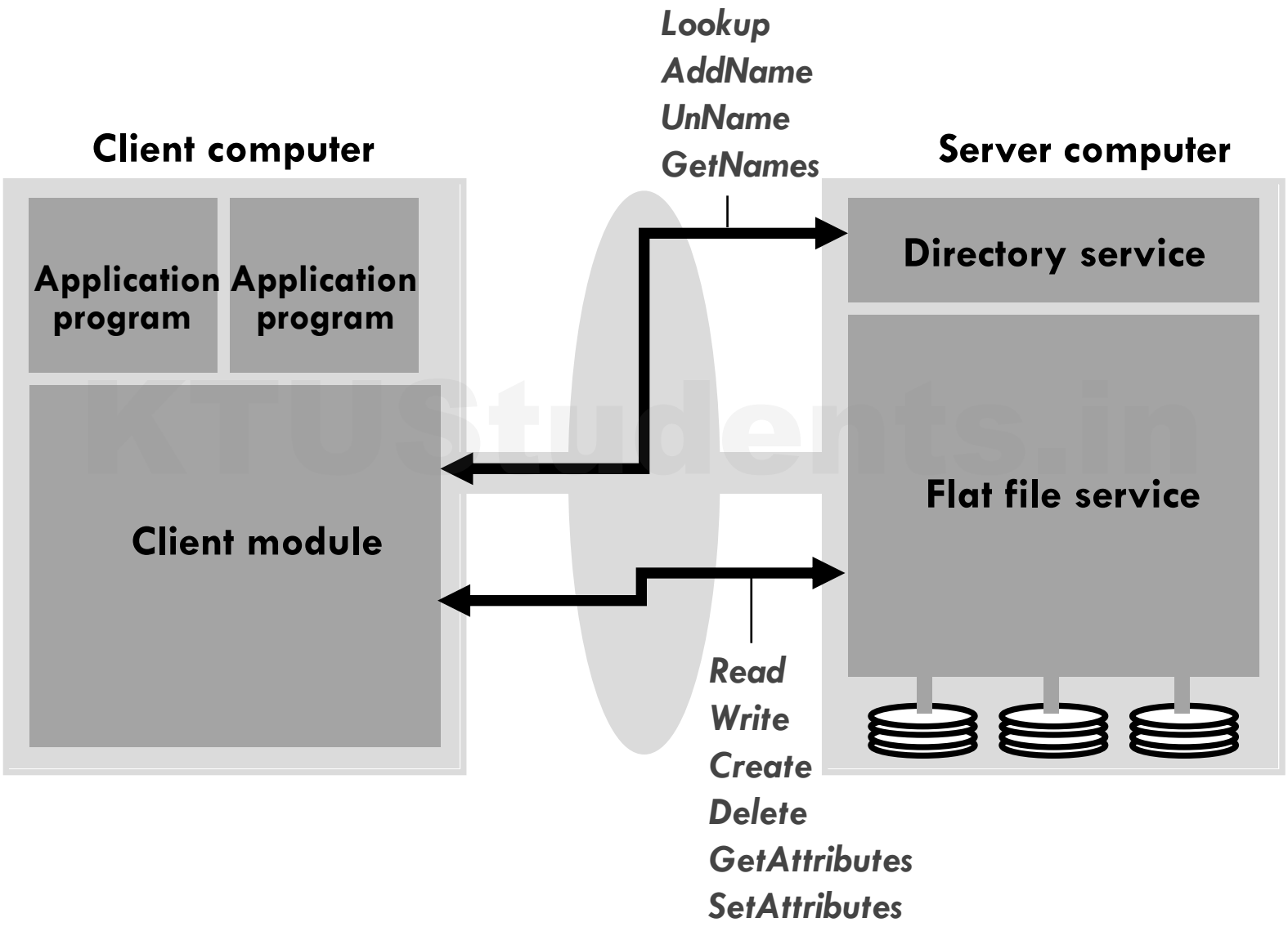
- ✓ **a flat file service**
- ✓ **a directory service**
- ✓ **a client module**

File Service Architecture

- A distributed file service architecture, structured as **three components** hides the concerns in **providing access to files.**
  - **A flat file service**
  - **A directory service**
  - **A client module**

- *The relevant modules and their relationships are shown in Figure.*

File Service Architecture ...



File Service Architecture...

- The **flat file service** and the **directory service:**
  - each export an **interface** for use by client programs.
  - Along with their *RPC interfaces*, they provide a comprehensive **set of operations for file access.**
- The **client module:**
  - provides a **single programming interface** with operations on files similar to those found in local file systems.

Responsibilities of modules

- **Flat File Service:**
  - Concerned with the implementation of **operations on the contents of file.**
  - **Unique File Identifiers (UFIDs)** are used to refer to files in all requests for operations.
  - UFIDs are *long sequences of bits* chosen so that each file has a unique ID among all of the files in a distributed system.
  - When a flat file service receives a request to create a file, it generates a new UFID for it and returns the UFID to the requester.

Responsibilities of modules…

⊙ Directory Service:

- Provides **mapping** between *text names* of files and their UFIDs.
  - » Clients may obtain the UFID of a file by quoting its text name to directory service.
- It also provides the functions needed **create 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 the files of flat file service.
- Directories hold **references to other directories** in a **hierarchic** scheme.

Responsibilities of modules…

⊙ Client Module:

- It runs on each computer and provides integrated services *(flat file and directory services)* as a **single API** to user level application programs.
  - » *Eg., in UNIX hosts, a client module emulates the full set of Unix file operations, interpreting UNIX multi-part file names by iterative requests to the directory service.*
- It holds information about the **network locations** of flat-file server and directory server processes
- Achieve better performance by implementing a **cache of recently used file blocks** at the client.

Flat File Service Interface

- This is the **RPC interface used by client** modules.
  - *not normally used directly by user-level programs.*
- A ***FileId*** is *invalid* if that file is not present in the server or if its access permissions are inappropriate for the operation requested.
  - *All functions except Create throw exception if the FileId contains invalid UFID or the user doesn't have access rights.*
- ***Read*** and ***Write***: Used for reading and writing files.
  - Needs '**i**' which specifies the position in the file.
- ***Read***: copies the sequence of *n* data items beginning at item *i* from the specified file into Data, which is then returned to the client.

Flat File Service Interface ...

- ***Write***: copies the sequence of data items in Data into the specified file beginning at item *i*.
  - It replaces the previous contents of the file at the corresponding position or extends the file if necessary.
- ***Create***: Creates a new empty file and returns its UFID that is generated.
- ***Delete***: Removes the file.
- ***GetAttribute, SetAttribute***: Enable clients to access the attributes of a file.
  - *GetAttribute* is available to any client who can read it.
  - *SetAttribute* is restricted to use by the directory service that provides access to file.
    - » **Length** and **timestamp** fields cannot be changed; they are maintained separately by the flat file service itself.

Flat File Service Operations

Read(FileId, i, n) -> Data — throws BadPosition	If 1 ≤ i ≤ 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 ≤ i ≤ 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 file attribute figure).

Comparison with UNIX

- This interface and the UNIX file system primitives are **functionally equivalent.**
- But, flat file service has **no *Open* and *Close* operations** – files can be accessed immediately by quoting the appropriate UFID.
- *Read* and *Write* calls include a parameter to indicate the **starting point** within the file, while UNIX do not have that.
  - In UNIX, each read or write starts at the current position of the read-write pointer, and the pointer is advanced by the number of bytes transferred after each read or write.
  - A **seek** operation is provided to enable the read-write pointer to be explicitly repositioned.

Comparison with UNIX...

- Flat file service differs from UNIX in ***fault tolerance*.**
  - **Repeatable Operations**
    - » Except for *create*, all operations are idempotent, allowing to use *at least one* semantics – clients may repeat calls for which they receive no reply.
    - » Repeated execution of *Create* produces a different new file for each call.
  - **Stateless Servers**
    - » Services can be restarted after crash **without any need** for clients or the server **to restore any state.**
- ***The UNIX file operations are neither idempotent nor consistent for a stateless implementation.***

Comparison with UNIX...

- A **read-write pointer** is **generated** by the UNIX file system whenever a file is opened, and it is **retained**, together with the results of access-control checks, **until the file is closed**.
- The UNIX read and write operations are **not idempotent**;
  - if an operation is accidentally repeated, the **automatic advance of the read-write pointer** results in access to a different portion of the file in the repeated operation.
- The read-write pointer is a **hidden, client-related state variable**.
- To mimic it in a file server, ***open*** and ***close*** operations would be needed, and the read-write pointer's value have to be retained by the server as long as the relevant file is open.
- By eliminating the read-write pointer, we have eliminated most of the need for the file server to retain state information on behalf of specific clients.

Access Control

- UNIX checks access rights against ***access mode*** when a file is opened using *open* call.
  - The user identity (UID) is retrieved from login.
  - Retained until *close* operation, so subsequent checks during read/write are not necessary.
- In distributed file system environment,
  - **access rights checks** are to be **done at the server.**
    - *because the server RPC interface is an otherwise unprotected point of access to files.*
  - UID has to be passed with each requests.
    - *the server is vulnerable to forged identities.*
  - If the results of access rights are retained in the server, it will no longer be stateless. Two approaches can be taken.

Access Control ...

- **Two Stateless** approaches:
- **Access check is made whenever a file name is converted to a UFID:**
  - Client gets back the results encoded in the form of a **capability** *(who can access and how)*.
  - C*apability* is submitted with each subsequent requests.
- **Access check for each request using UID:**
  - UID is submitted with every client request, and access checks are performed by the server for every file operation.
- Second approach is more common.
  - Used in NFS and AFS

Access Control ...

- Neither of these approaches overcomes the security problem concerning forged user identities, but it can be addressed by the use of **digital signatures.**
- **Kerberos** is an effective **authentication** scheme that has been applied to both NFS and AFS.

Directory Service Interface

- Provides a service for translating **text names to UFID's.**
- To do so, it maintains ***directory files*** containing the **mappings** between text names for files and UFID's.
- Each directory is stored as a **conventional file** with a UFID, so *directory service is a client of file service*.
- The RPC interface to directory handles operations on individual directories.
- For each operation, a **UFID** for the file containing the directory is required in the ***Dir*** parameter.

Directory Service Operations

- The RPC interface to a directory service:

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: the entry containing *Name* is removed 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.

Directory Service Interface...

- ***LookUp*** performs a single ***Name --> UFID*** translation
  - a building block for use in other services or in the client module to perform more complex translations, such as the **hierarchic** name interpretation.
- Operations for altering directories: *AddName, UnName*
- ***AddName*** adds an entry to the directory and increments the reference count field of the file attribute.
- ***UnName*** removes an entry from a directory and decrements the reference count.
  - If this causes the reference count to reach zero, the file is removed.

Directory Service Interface...

- ***GetNames*** enable clients to examine the contents of directories and to implement pattern matching operations on file names.
  - Returns all or a subset of the names stored in a given directory.
  - The names are selected by pattern matching against a regular expression supplied by the client.
  - *Pattern matching enables users to determine the names of one or more files by giving an incomplete specification of the characters in the names.*

Hierarchical file system

- **Directory Tree:** directories arranged in a tree structure.
  - Each directory is a special file which holds the names of the files and other directories that are accessible from it.
- ***Pathname*** - Reference a file or a directory
  - Multi-part name that represents a path through the tree. eg. "/etc/rc.d/init.d/nfsd"
  - The **root** has a distinguished name, and each file or directory has a name in a directory.
  - In UNIX, files can have several names, and they can be in the same or different directories.
  - A ***link*** operation adds a new name for a file to a specified directory.

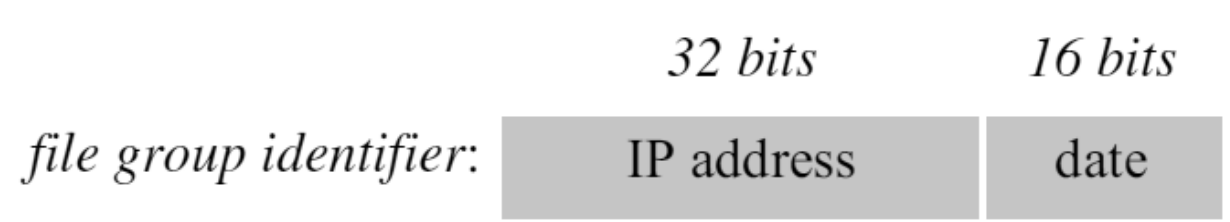
Hierarchical file system ...

- A UNIX-like file-naming system can be implemented by the client module using the flat file and directory services.
- 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.
- Client module can use a function that **gets the UFID of a file given its *pathname*.**
  - The function interprets the pathname starting from the root, using multiple ***Lookup*** to obtain the UFID of each directory in the path.
- In a hierarchical service, the file attributes should include a **type field** that **distinguishes between ordinary files and directories.**
  - This is used when following a path to ensure that each part of the name, except the last, refers to a directory.

File Groups

- A **file group** is a **collection of files** located on a given server.
- A server may hold several file groups.
- Groups can be moved between servers while maintaining the same names.
- But a file cannot change the group to which it belongs.
- Similar to a UNIX ***filesystem.***
- Helps with distributing the load between several servers.
- Initially created to support facilities for moving collections of files stored on removable media between computers.

File Groups



- File groups must have **globally unique identifiers** because they can be moved across systems or several distributed systems can be merged.
- Eg., whenever a new file group is created, a unique id 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.
- But, IP address cannot be used for 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.

File Groups…

- 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.

Transactions and Concurrency Control

Transactions and Concurrency Control

- A Transaction defines a sequence of server operations that is guaranteed to be atomic in the presence of multiple clients and server crash.
- All concurrency control protocols are based on serial equivalence and are derived from rules of conflicting operations.
  - Locks are used to order transactions that access the same object according to request order.
  - Optimistic concurrency control allows transactions to proceed until they are ready to commit, whereupon a check is made to see any conflicting operation on objects.
  - Timestamp ordering uses timestamps to order transactions that access the same object according to their starting time.

Banking Example

- Each account is represented by a remote object whose interface ***Account*** provides operations for making deposits and withdrawals and for enquiring about and setting the balance.
- Each branch of the bank is represented by a remote object whose interface ***Branch*** provides operations for creating a new account, for looking up an account by name and for enquiring about the total funds at that branch.
- Main issue: unless a server is carefully designed, its operations performed on behalf of different clients may sometimes interfere with one another. Such interference may result in incorrect values in the object.

Figure 13.1
Operations of the *Account* interface

deposit(amount)
deposit amount in the account
withdraw(amount)
withdraw amount from the account
getBalance() -> amount
return the balance of the account
setBalance(amount)
set the balance of the account to amount

Operations of the Branch interface

create(name) -> account
create a new account with a given name
lookUp(name) -> account
return a reference to the account with the given name
branchTotal() -> amount
return the total of all the balances at the branch

Simple Synchronization without Transactions

- The use of multiple threads is beneficial to the performance.
- Multiple threads may access the same objects.
  - For example, deposit and withdraw methods
- Synchronized keyword can be applied to method in Java, so only one thread at a time can access an object.

  (If one thread invokes a synchronized method on an object, then that object is locked, another thread that invokes one of the synchronized method will be blocked.)

Enhancing Client Cooperation by **synchronization of server operations**

- We have seen how clients may use a server as a means of sharing some resources.
  - E.g. some clients update the server's objects and other clients access them.
- In some applications, threads need to communicate and coordinate their actions.
- Producer and Consumer problem.
  - Wait and Notify actions.

What is a Transaction?

- Transaction - originally from database management systems.
- Clients require a sequence of separate requests to a server to be atomic in the sense that:
  - Other concurrent clients should not interfere; and
  - Either all of the operations must be completed successfully or they must have no effect at all in the presence of server crashes.

Atomicity

- All or nothing: a transaction either completes successfully, and effects of all of its operations are recorded in the object, or it has no effect at all.
  - Failure atomicity: effects are atomic even when server crashes
  - Durability: after a transaction has completed successfully, all its effects are saved in permanent storage for recover later.
- Isolation: each transaction must be performed without interference from other transactions. The intermediate effects of a transaction must not be visible to other transactions.

A client's banking transaction

Transaction T:
a.withdraw(100);
b.deposit(100);

Operations in *Coordinator* interface

openTransaction() -> trans;

starts a new transaction and delivers a unique TID *trans*. This identifier will be used in the other operations in the transaction.

closeTransaction(trans) -> (commit, abort);

ends a transaction: a *commit* return value indicates that the transaction has committed; an *abort* return value indicates that it has aborted.

abortTransaction(trans);

aborts the transaction.

Transaction life histories

Successful	*Aborted by client*	*Aborted by server*
openTransaction	*openTransaction*	*openTransaction*
operation	*operation*	*operation*
operation	*operation*	*operation*
:	:	server aborts transaction → :
operation	*operation*	*operation ERROR reported to client*
closeTransaction	*abortTransaction*	

If a transaction aborts for any reason (self abort or server abort), it must be guaranteed that future transaction will not see its effect either in the object or in their copies in permanent storage.

Concurrency Control

- Two well-known problems of concurrent transactions in the context of the banking example –
- The ‘lost update’ problem and
- The ‘inconsistent retrievals’ problem.

Concurrency Control: the lost update problem

Transaction *T*:		Transaction *U*:	
balance = b.getBalance(); *b.setBalance(balance*1.1);* *a.withdraw(balance/10)*		*balance = b.getBalance();* *b.setBalance(balance*1.1);* *c.withdraw(balance/10)*	
balance = b.getBalance();	$200		
		balance = b.getBalance();	$200
		*b.setBalance(balance*1.1);*	$220
*b.setBalance(balance*1.1);*	$220		
a.withdraw(balance/10)	$80		
		c.withdraw(balance/10)	$280

a, b and c initially have bank account balance are: 100, 200, and 300. T transfers an amount from a to b. U transfers an amount from c to b.

Concurrency Control: The inconsistent retrievals problem

Transaction *V*:		Transaction *W*:	
a.withdraw(100) *b.deposit(100)*		*aBranch.branchTotal()*	
a.withdraw(100);	$100		
		total = a.getBalance()	$100
		total = total+b.getBalance()	$300
		total = total+c.getBalance()	
b.deposit(100)	$300	:	

a, b accounts start with 200 both.

Serial equivalence

- If these transactions are done one at a time in some order, then the final result will be correct.
- If we do not want to sacrifice the concurrency, an interleaving of the operations of transactions may lead to the same effect as if the transactions had been performed one at a time in some order.
- We say it is a serially equivalent interleaving.
- The use of serial equivalence is a criterion for correct concurrent execution to prevent lost updates and inconsistent retrievals.

A serially equivalent interleaving of *T* and *U*

Transaction *T*:		**Transaction *U*:**	
balance = b.getBalance()		*balance = b.getBalance()*	
*b.setBalance(balance*1.1)*		*b.setBalance(balance*1.1)*	
a.withdraw(balance/10)		*c.withdraw(balance/10)*	
balance = b.getBalance()	$200		
*b.setBalance(balance*1.1)*	$220		
		balance = b.getBalance()	$220
		*b.setBalance(balance*1.1)*	$242
a.withdraw(balance/10)	$80		
		c.withdraw(balance/10)	$278

Conflicting Operations

- When we say a pair of operations conflicts we mean that their combined effect depends on the order in which they are executed. eg: read and write

- Three ways to ensure serializability:
  - Locking
  - Timestamp ordering
  - Optimistic concurrency control

T_0	T_1
read(A)	
write(A)	
	read(A)
	write(A)
read(B)	
write(B)	
	read(B)
	write(B)

Read and *write* operation conflict rules

Operations of different transactions		*Conflict*	*Reason*
read	*read*	No	Because the effect of a pair of *read* operations does not depend on the order in which they are executed
read	*write*	Yes	Because the effect of a *read* and a *write* operation depends on the order of their execution
write	*write*	Yes	Because the effect of a pair of *write* operations depends on the order of their execution

Recoverability from aborts

- Servers must record the effect of all committed transactions and none of the effects of the aborted transactions.
- Two problems associated with aborting transactions that may occur in the presence of serially equivalent execution of transactions:
  - Dirty reads
  - Premature writes

A dirty read when transaction *T* aborts

Transaction *T*:	**Transaction *U*:**
a.getBalance()	*a.getBalance()*
a.setBalance(balance + 10)	*a.setBalance(balance + 20)*

balance = a.getBalance()	$100
a.setBalance(balance + 10)	$110
balance = a.getBalance()	$110
a.setBalance(balance + 20)	$130
commit transaction	
abort transaction	

Dirty reads caused by a read in one transaction U and an earlier unsuccessful write in another transaction T on the same object.
T will be rolled back and restore the original a value, thus U will have seen a value that never existed. U is committed, so cannot be undone. U performs a dirty read.

Premature Write: Overwriting uncommitted values

Transaction *T*:		**Transaction *U*:**	
a.setBalance(105)		*a.setBalance(110)*	
	$100		
a.setBalance(105)	$105		
		a.setBalance(110)	$110

Premature write: related to the interaction between write operations on the same object belonging to different transactions.

a. If U aborts and then T commit, we got a to be correct 105.

Some systems restore value to "Before images" value for abort action, namely the value before all the writes of a transaction. a is 100, which is the before image of T's write. 105 is the before image of U's write.

b. Consider if U commits and then T aborts, we got wrong value of 100.

c. Similarly if T aborts then U aborts, we got 105, which is wrong and should be 100.

So to ensure correctness, write operations must be delayed until earlier transactions that updated the same object have either committed or aborted.

Strict executions of transactions

- Generally, it is required that transactions delay both their read and write operations so as to avoid both ‘dirty reads’ and ‘premature writes’.
- The executions of transactions are called strict if the service delays both read and write operations on an object until all transactions that previously wrote that object have either committed or aborted.

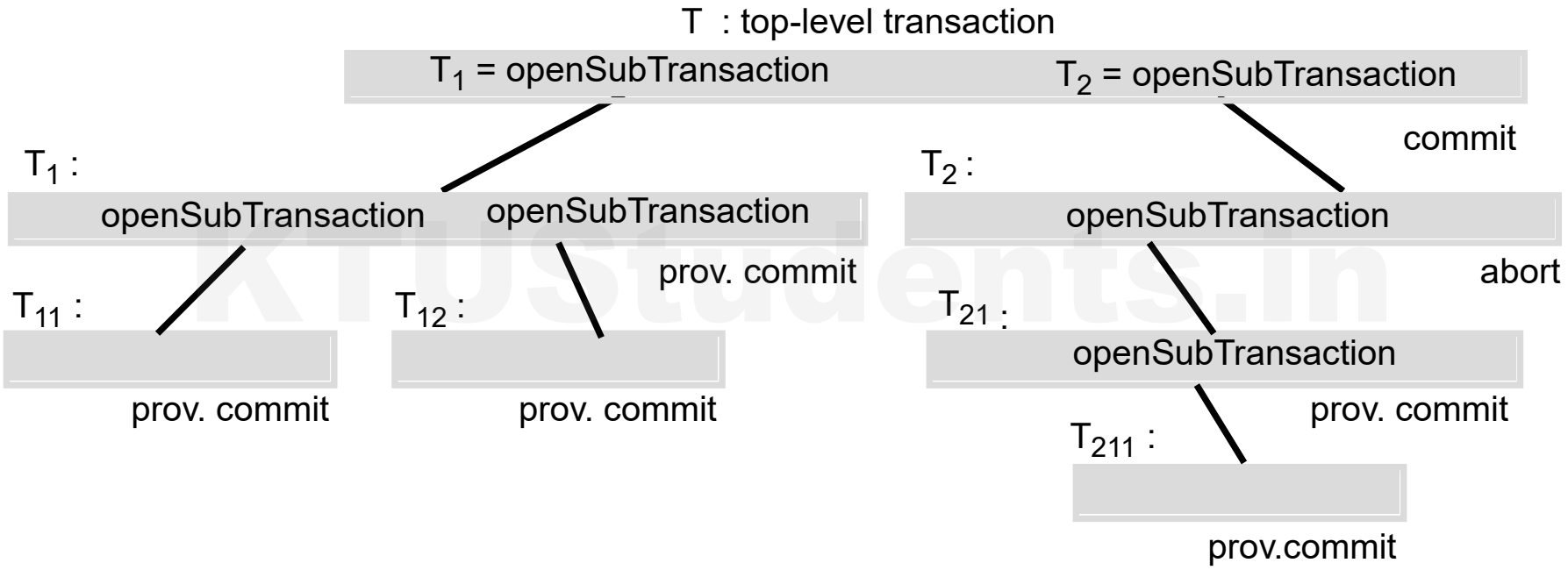
Tentative versions

- For a server of recoverable objects to participate in transactions, it must be designed so that any updates of objects can be removed if and when a transaction aborts.
- All of the update operations performed during a transaction are done in tentative versions of objects in volatile memory.
- The tentative versions are transferred to the objects only when a transaction commits, by which time they will also have been recorded in permanent storage.
- This is performed in a single step, during which other transactions are excluded from access to the objects that are being altered.

Nested transactions

- Nested transaction extend the transaction model by allowing transactions to be composed of other transactions.
- The outermost transaction in a set of nested transactions is called the top-level transaction.
- Transactions other than the top-level transaction are called subtransactions.

Nested transactions



Nested transactions

- The 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. Its decision to abort is final.
  - When a parent aborts, all of its subtransactions are aborted.
  - When a subtransaction aborts, the parent can decide whether to abort or not.
  - If the top-level transaction commits, then all of the subtransactions that have provisionally committed can commit too, provided that none of their ancestors has aborted.

Locks

- A simple example of a serializing mechanism is the use of exclusive locks.
- Server can lock any object that is about to be used by a client.
- If another client wants to access the same object, it has to wait until the object is unlocked in the end.

Transactions *T* and *U* with exclusive locks

Transaction *T*:		**Transaction *U*:**	
balance = b.getBalance() *b.setBalance(bal*1.1)* *a.withdraw(bal/10)*		*balance = b.getBalance()* *b.setBalance(bal*1.1)* *c.withdraw(bal/10)*	
Operations	Locks	Operations	Locks
openTransaction			
bal = b.getBalance()	lock *B*		
*b.setBalance(bal*1.1)*		*openTransaction*	
a.withdraw(bal/10)	Lock A	*bal = b.getBalance()*	waits for *T*'s lock on *B*
closeTransaction	unlock *A*, *B*	• • •	
			lock *B*
		*b.setBalance(bal*1.1)*	
		c.withdraw(bal/10)	lock *C*
		closeTransaction	unlock *B*, *C*

Two-phase locking

- Serial equivalence requires that all of a transaction's accesses to a particular object be serialized with respect to accesses by other transactions. All pair of conflicting operations of two transactions should be executed in the same order.
- To ensure this, a transaction is not allowed any new locks after it has released a lock.
- Two-phase locking
  - The first phase of each transaction is a 'growing phase', during which new locks are acquired.
  - In the second phase, the locks are released (a 'shrinking phase').
- Strict two-phase locking
  - Any locks applied during the progress of a transaction are held until the transaction commits or aborts.

read locks and write locks

- It is preferable to adopt a locking scheme that controls the access to each object so that there can be several concurrent transactions reading an object, or a single transaction writing an object, but not both – commonly referred to as a ‘many readers/single writer’ scheme.
- Two types of locks are used: read locks and write locks
  - Before a transaction’s read operation is performed, a read lock should be set on the object.
  - Before a transaction’s write operation is performed, a write lock should be set on the object.
  - Whenever it is impossible to set a lock immediately, the transaction must wait until it is possible to do so.
  - As pair of read operations from different transactions do not conflict, an attempt to set a read lock on an object with a read lock is always successful. Therefore, read locks are sometimes called shared lock.

Lock compatibility

For one object		*Lock requested*	
		read	*write*
Lock already set	*none*	OK	OK
	read	OK	wait
	write	wait	wait

An object can be read and write. From the compatibility table, we know pairs of read operations from different transactions do not conflict. So a **simple exclusive lock** used for both read and write **reduces concurrency** more than necessary.
(Many readers/Single writer)
Rules;

1. If T has already performed a read operation, then a concurrent transaction U must not write until T commits or aborts.
2. If T already performed a write operation, then concurrent U must not read or write until T commits or aborts.

Strict two-phase Locking Protocol

- Because transaction may abort, strict execution are needed to prevent dirty reads and premature writes, which are caused by read or write to same object accessed by another earlier unsuccessful transaction that already performed an write operation.
- So to prevent this problem, 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.
- Rule:
- Any **locks** applied during the progress of a transaction are **held until** the transaction **commits** or **aborts**.

Strict two-phase Locking Protocol

$$D = \begin{bmatrix} T1 & T2 \\ S(A) & \\ R(A) & \\ & S(A) \\ & R(A) \\ & X(B) \\ & R(B) \\ & W(B) \\ & Commit \\ X(C) & \\ R(C) & \\ W(C) & \\ Commit & \end{bmatrix}$$

Use of locks in strict two-phase locking

1. When an operation accesses an object within a transaction:
   (a) If the object is not already locked, it is locked and the operation proceeds.
   (b) If the object has a conflicting lock set by another transaction, the transaction must wait until it is unlocked.
   (c) If the object has a non-conflicting lock set by another transaction, the lock is shared and the operation proceeds.
   (d) If the object has already been locked in the same transaction, the lock will be promoted if necessary and the operation proceeds. (Where promotion is prevented by a conflicting lock, rule (b) is used.)
2. When a transaction is committed or aborted, the server unlocks all objects it locked for the transaction.

A transaction with a read lock that is shared by other transactions cannot promote its read lock to a write lock, because write lock will conflict with other read locks.

Lock class - implementation

```
public class Lock {
        private Object object;              // the object being protected by the lock
        private Vector holders;             // the TIDs of current holders
        private LockType lockType;          // the current type
        public synchronized void acquire(TransID trans,   LockType aLockType ){
                while(/*another transaction holds the lock in conflicing mode*/) {
                        try {
                                wait();
                        }catch ( InterruptedException e){/*...*/ }
                }
                if(holders.isEmpty()) { // no TIDs  hold lock
                        holders.addElement(trans);
                        lockType  = aLockType;
                } else if(/*another transaction holds the lock, share it*/ ) ){
                        if(/* this transaction not a holder*/) holders.addElement(trans);
                } else if (/* this transaction is a holder but needs a more exclusive lock*/)
                                lockType.promote();
                }
        }
```

Continues on next slide

Lock class - implementation

```
public synchronized void release(TransID trans ){
                holders.removeElement(trans);       // remove this holder
                // set locktype to none
                notifyAll();
        }
}
```

Locking rules for nested transactions

- every lock that is acquired by a successful subtransaction is inherited by its parent when it completes
- Parent transactions are not allowed to run concurrently with their child transactions

Deadlock with write locks

Transaction *T*		Transaction *U*	
Operations	**Locks**	**Operations**	**Locks**
a.deposit(100);	write lock *A*		
		b.deposit(200)	write lock *B*
b.withdraw(100)			
• • •	waits for *U*'s lock on *B*	*a.withdraw(200);*	waits for *T*'s lock on *A*
• • •		• • •	
• • •		• • •	
		• • •	

The wait-for graph for Figure 13.19

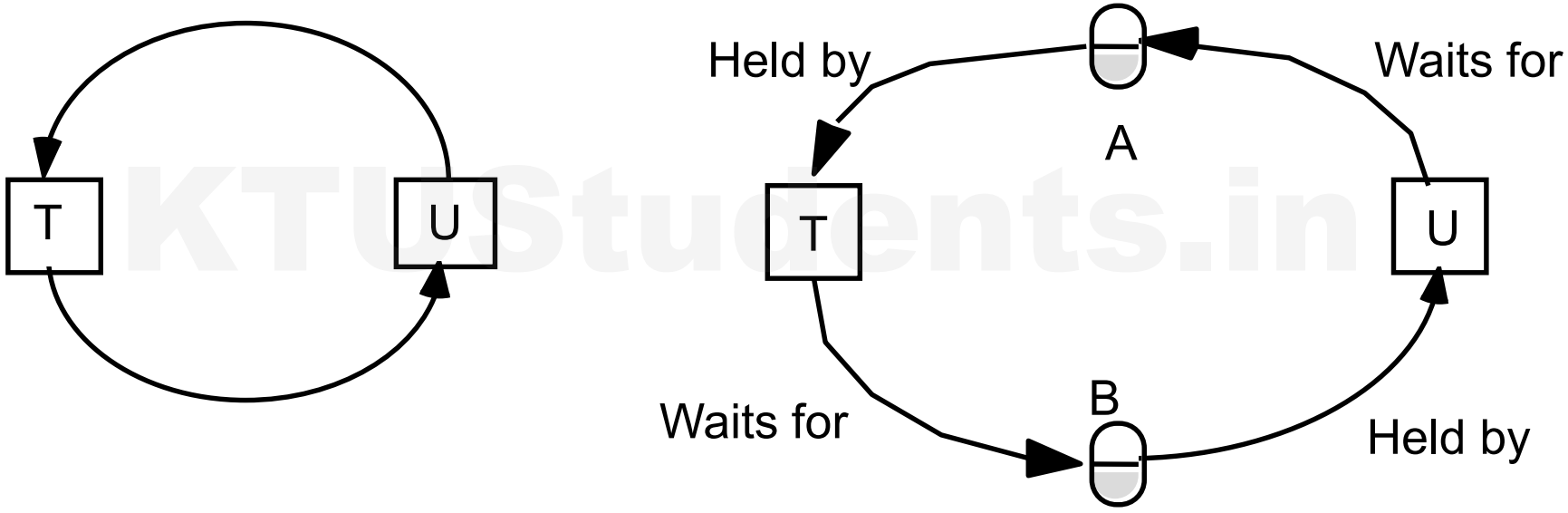


Figure 13.21
A cycle in a wait-for graph

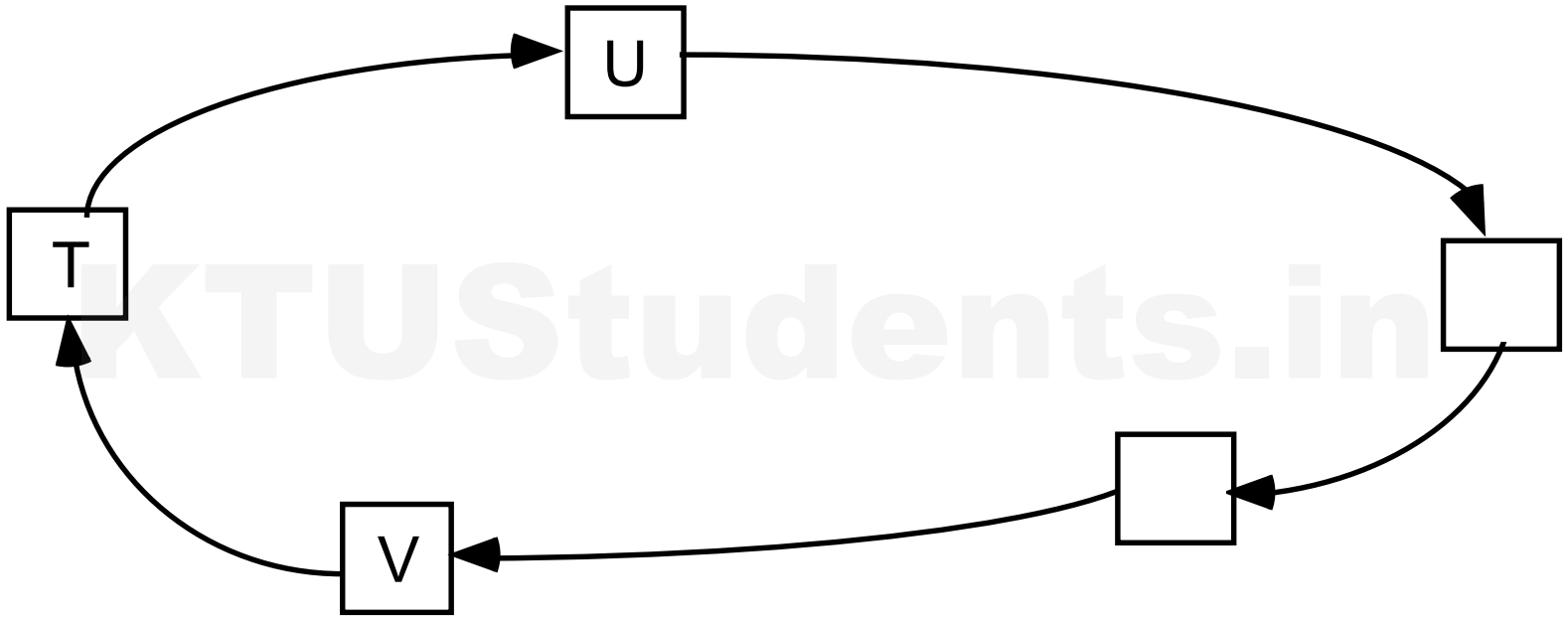
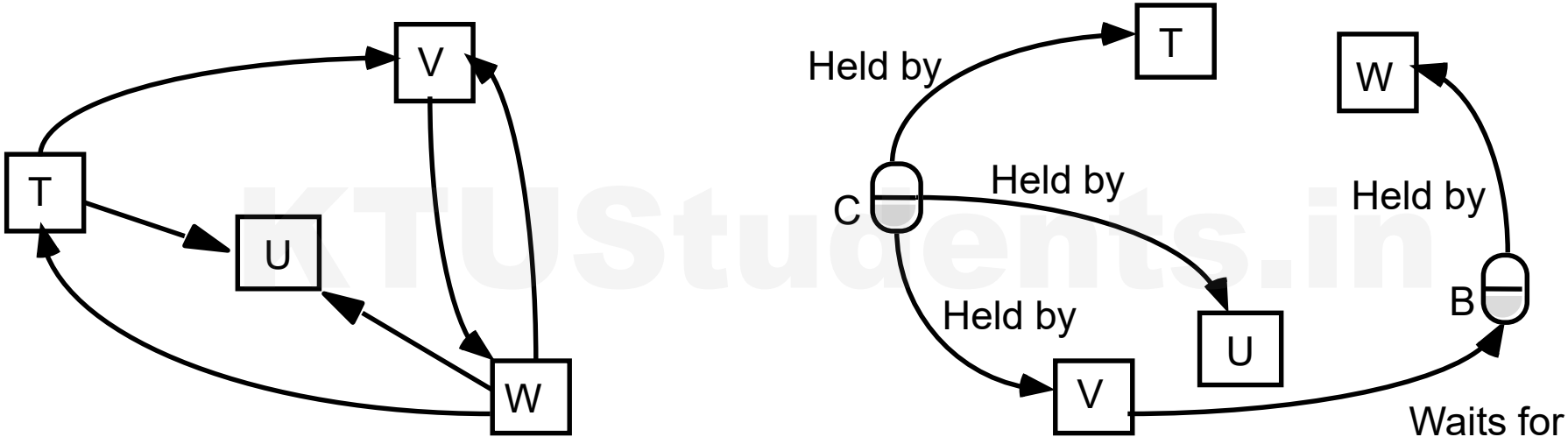


Figure 13.22
Another wait-for graph



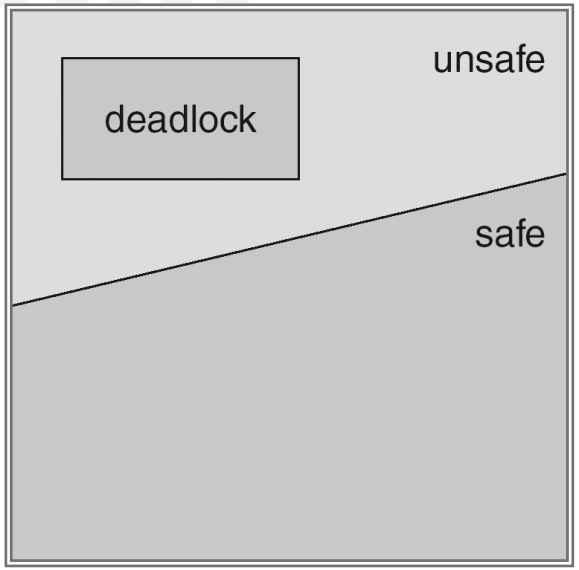
T and W then request write locks on object C and a deadlock arises. V is involved in two cycles.

Deadlock Prevention

- Deadlock prevention:
  - Simple way is to lock all of the objects used by a transaction when it starts. It should be done as an atomic action to prevent deadlock. a. inefficient, say lock an object you only need for short period of time. b. Hard to predict what objects a transaction will require.
  - Judge if system can remain in a Safe state by satisfying a certain resource request. Banker's algorithm.
  - Order the objects in certain order. Acquiring the locks need to follow this certain order.

Safe State

- System is in safe state if there exists a sequence $<P_1, P_2, \ldots, P_n>$ of ALL the processes is the systems such that for each P_i, the resources that P_i can still request can be satisfied by currently available resources + resources held by all the P_j, with $j < i$.



If a system is in safe state ⇒ no deadlocks.
If a system is in unsafe state ⇒ possibility of deadlock.
Avoidance ⇒ ensure that a system will never enter an unsafe state

- Banker's Algorithm

Deadlock Detection

- Deadlock may be detected by finding cycles in the wait-for-graph. Having detected a deadlock, a transaction must be selected for abortion to break the cycle.
  - If lock manager blocks a request, an edge can be added. Cycle should be checked each time a new edge is added.
  - One transaction will be selected to abort in case of cycle. Age of transaction and number of cycles involved when selecting a victim
- **Timeouts** is commonly used to resolve deadlock. Each lock is given a limited period in which it is invulnerable. After this time, a lock becomes vulnerable.
  - If no other transaction is competing for the object, vulnerable object remained locked. However, if another transaction is waiting, the lock is broken.

  Disadvantages:
  - Transaction aborted simply due to timeout and waiting transaction even if there is no deadlock. (may add deadlock detection)
  - Hard to set the timeout time

Resolution of the deadlock – Time-outs

Transaction T		Transaction U	
Operations	**Locks**	**Operations**	**Locks**
a.deposit(100);	write lock *A*		
		b.deposit(200)	write lock *B*
b.withdraw(100)			
•••	waits for *U*'s lock on *B*	*a.withdraw(200);*	waits for T's lock on *A*
	(timeout elapses)	•••	
T's lock on *A* becomes vulnerable, unlock *A*, abort T		•••	
		a.withdraw(200);	write locks *A* unlock *A*, *B*

Optimistic Concurrency Control

- Kung and Robinson [1981] identified a number of inherent disadvantages of locking and proposed an alternative optimistic approach to the serialization of transaction that avoids these drawbacks. Disadvantages of lock-based:
  - Lock maintenance represents an overhead that is not present in systems that do not support concurrent access to shared data. Locking sometimes are only needed for some cases with low probabilities.
  - The use of lock can result in deadlock. Deadlock prevention reduces concurrency severely. The use of timeout and deadlock detection is not ideal for interactive programs.
  - To avoid cascading aborts, locks cannot be released until the end of the transaction. This may reduce the potential for concurrency.

Optimistic Concurrency Control

- It is based on observation that, in most applications, the likelihood of two clients' transactions accessing the same object is low. Transactions are allowed to proceed as though there were no possibility of conflict with other transactions until the client completes its task and issues a *closeTransaction* request.
- When conflict arises, some transaction is generally aborted and will need to be restarted by the client.

Optimistic Concurrency Control

- Each transaction has the following phases:
  - **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. The tentative version allows the transaction to abort with no effect on the object, either during the working phase or if it fails validation due to other conflicting transaction. Several different tentative values of the same object may coexist. In addition, two records are kept of the objects accessed within a transaction, a read set and a write set containing all objects either read or written by this transaction. Read are performed on committed version (no dirty read can occur) and write record the new values of the object as tentative values which are invisible to other transactions.

Optimistic Concurrency Control

- **Validation phase:** When *closeTransaction* request is received, the transaction is validated to establish whether or not its operations on objects conflict with operations of other transaction on the same objects. If successful, then the transaction can commit. If fails, then either the current transaction or those with which it conflicts will need to be aborted.
- **Update phase:** If a transaction is validated, all of the changes recorded in its tentative versions are made permanent. Read-only transaction can commit immediately after passing validation. Write transactions are ready to commit once the tentative versions have been recorded in permanent storage.

Validation of Transactions

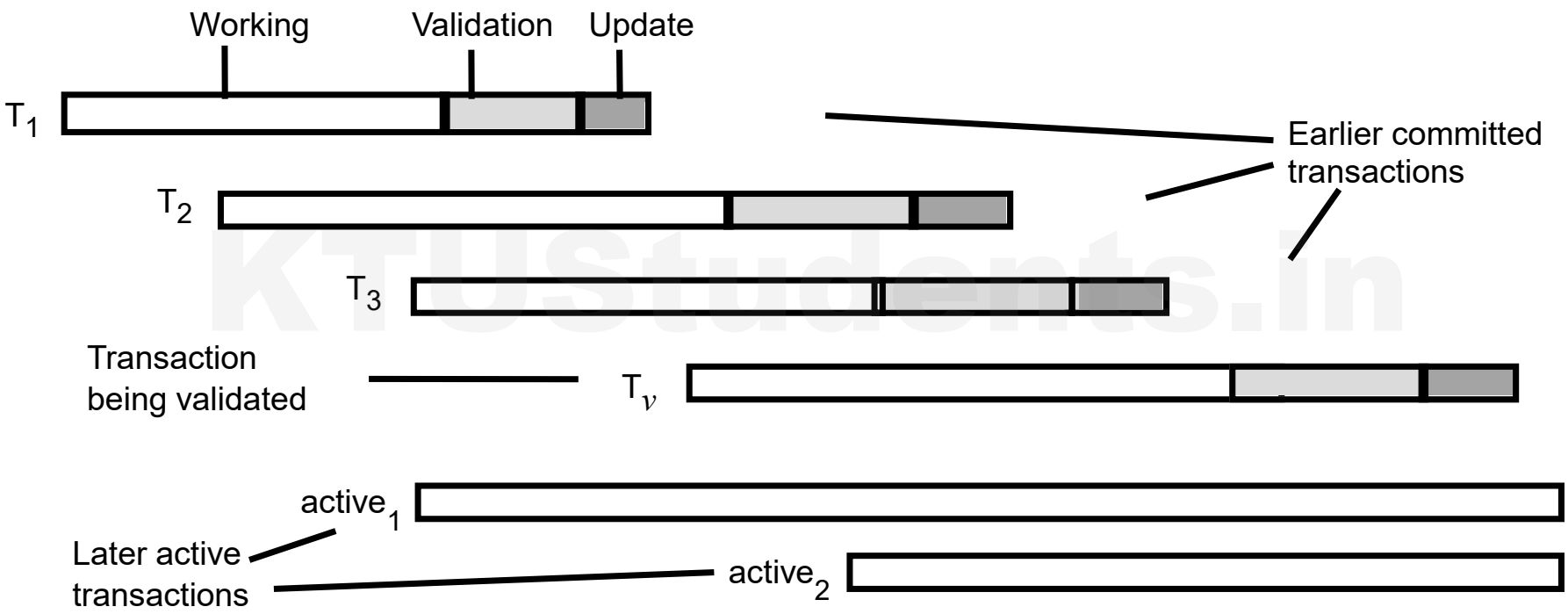
- Validation uses the read-write conflict rules to ensure that the scheduling of a particular transaction is serially equivalent with respect to all other overlapping transactions- that is, any transactions that had not yet committed at the time this transaction started. Each transaction is assigned a number when it enters the validation phase (when the client issues a *closeTransaction*). Such number defines its position in time. A transaction always finishes its working phase after all transactions with lower numbers. That is, a transaction with the number Ti always precedes a transaction with number Tj if i < j.

Serializability of transaction T with respect to transaction T_i

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

The validation test on transaction *Tv* is based on conflicts between operations in pairs of transaction *Ti* and *Tv*, for a transaction *Tv* to be serializable with respect to an overlapping transaction *Ti*, their operations must conform to the above rules.

Figure 13.28
Validation of transactions



Validation

- 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.

Validation of Transactions

Backward validation of transaction T_v

```
boolean valid = true;
for (int Ti = startTn+1; Ti <= finishTn; Ti++){
        if (read set of Tv intersects write set of Ti) valid = false;
}
```


Forward validation of transaction T_v

```
boolean valid = true;
for (int Tid = active1; Tid <= activeN; Tid++){
        if (write set of Tv intersects read set of Tid) valid = false;
}
```

Comparison of methods for Concurrency Control

- The timestamp ordering method is similar to two-phase locking in that both use pessimistic approaches in which conflicts between transactions are detected as each object is accessed. On the one hand, timestamp ordering decides the serialization order statically – when a transaction starts. On the other hand, two-phase locking decides the serialization order dynamically – according to the order in which objects are accessed. Timestamp ordering, and in particular multiversion timestamp ordering, is better than strict two-phase locking for read-only transactions. Two-phase locking is better when the operations in transactions are predominantly updates.
- The pessimistic methods differ in the strategy used when a conflicting access to an object is detected. Timestamp ordering aborts the transaction immediately, whereas locking makes the transaction wait – but with a possible later penalty of aborting to avoid deadlock.

Contd.

- When optimistic concurrency control is used, all transactions are allowed to proceed, but some are aborted when they attempt to commit, or in forward validation transactions are aborted earlier. This results in relatively efficient operation when there are few conflicts, but a substantial amount of work may have to be repeated when a transaction is aborted.
- Historically, the predominant method of concurrency control of access to data in distributed systems is by locking – for example, as mentioned earlier, the CORBA Concurrency Control Service is based entirely on the use of locks.

Share Document Applications

- The above concurrency control mechanisms are not always adequate for twenty-first-century applications that enable users to share documents over the Internet. Many of the latter use optimistic forms of concurrency control followed by conflict resolution instead of aborting one of any pair of conflicting operations. The following are some examples.
- Dropbox : Dropbox [www.dropbox.com] is a cloud service that provides file backup and enables users to share files and folders, accessing them from anywhere. Dropbox uses an optimistic form of concurrency control, keeping track of consistency and preventing clashes between users' updates – which are at the granularity of whole files. Thus if two users make concurrent updates to the same file, the first write will be accepted and the second rejected. However, Dropbox provides a version history to enable users to merge their updates manually or restore previous versions.

Contd.

- Google apps : Google Apps include Google Docs, a cloud service that provides web-based applications (word processor, spreadsheet and presentation) that allow users to collaborate with one another by means of shared documents. If several people edit the same document simultaneously, they will see each other's changes. In the case of a word processor document, users can see one another's cursors and updates are shown at the level of individual characters as they are typed by any participant. Users are left to resolve any conflicts that occur, but conflicts are generally avoided because users are continuously aware of each other's activities. In the case of a spreadsheet document, users' cursors and changes are displayed and updated at the granularity of single cells. If two users access the same cell simultaneously, the last update wins.
- Wikipedia : Concurrency control for editing is optimistic, allowing editors concurrent access to web pages in which the first write is accepted and a user making a subsequent write is shown an 'edit conflict' screen and asked to resolve the conflicts.

Coordination and Agreement

Introduction

- The goal is to introduce some topics and algorithms related to the issue of how processes *coordinate their actions and agree on shared values* in distributed systems, despite failures.
- An important distinction is whether the distributed system is *asynchronous or synchronous*
- In an asynchronous system, we can make no timing assumptions.
- In a synchronous system, assume that there are bounds
  - on the maximum message transmission delay
  - on the time to execute each step of a process
  - on clock drift rates
- The synchronous assumptions allow us to use timeouts to detect process crashes.

Distributed mutual exclusion

- Distributed processes often need to coordinate their activities.
- If a collection of processes share a resource or collection of resources, then often mutual exclusion is required to prevent interference and ensure consistency when accessing the resources – the critical section problem.

- eg: wireless Ad-hoc networks, car parking
  - No centralized server will help
- In a distributed system, require a solution to distributed mutual exclusion: one that is based solely on message passing.

12.2.1 Algorithms for mutual exclusion

- Consider a system of N processes p_i, $i = 1,2,\ldots, N$
  - Do not share variables.
  - Access common resources, but they do so in a critical section – assume that there is only one critical section.
- Assume that
  - The system is asynchronous
  - Processes do not fail
  - Message 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 an exit the critical section eventually succeed
ME3: (→ ordering)	If one request to enter the CS happened-before another, then entry to the CS is granted in that order

- (needs a global clock)

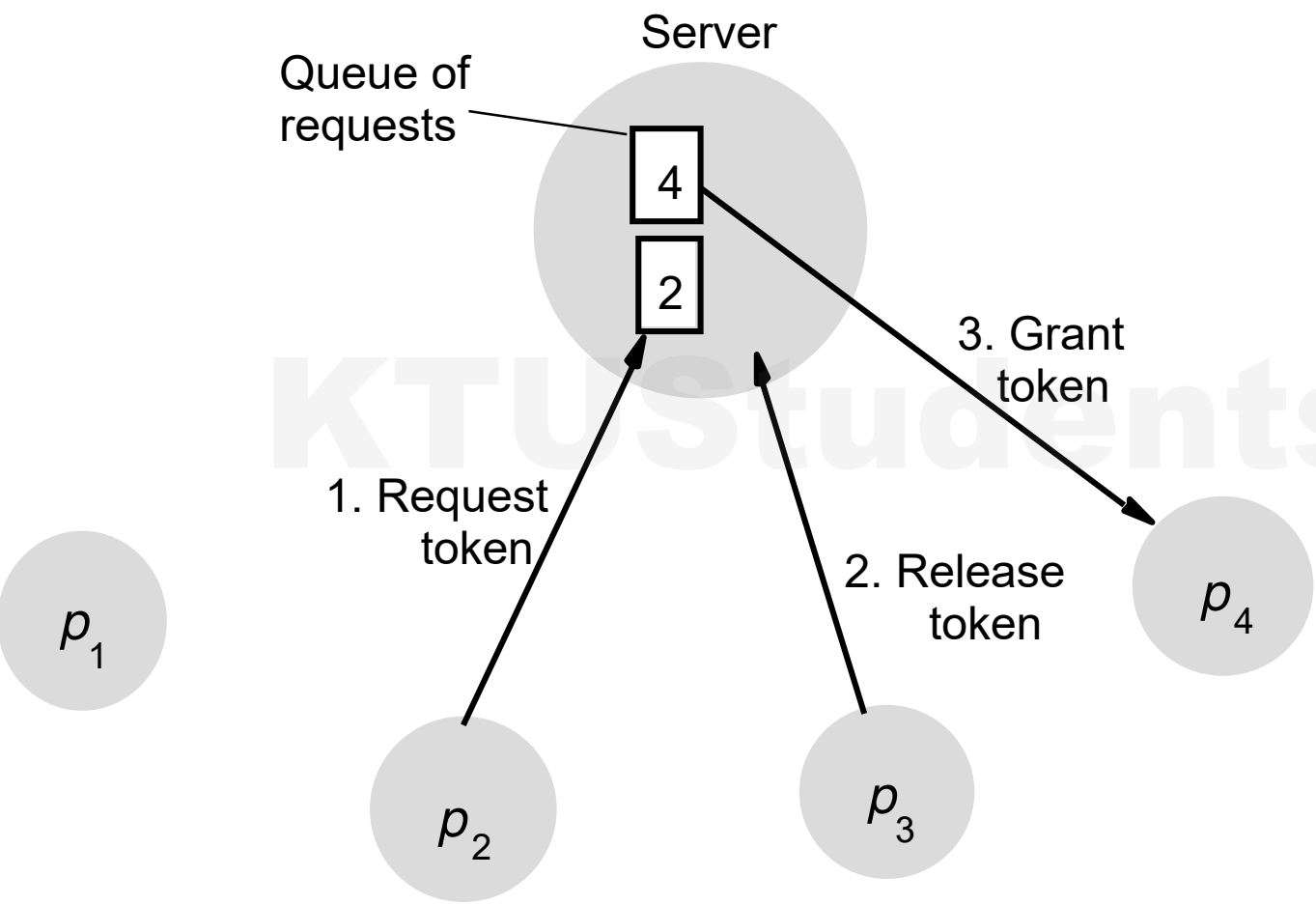
Performance Evaluation

- **Bandwidth** consumption, which is proportional to the number of messages sent in each entry and exit operations.
- The **client delay** incurred by a process at each entry and exit operation.
- **Throughput** of the system : Rate at which the collection of processes as a whole can access the critical section.
  - We can measure the effect using the **synchronization delay** between one process exiting the critical section and the next process entering it; the shorter the delay is, the greater the throughput is.

Central Server Algorithm

- The simplest way to grant permission to enter the critical section is to employ a server.
- A process sends a request message to server and awaits a reply from it.
  - A reply contains a token giving the permission to enter the critical section.
- If no other process has the token at the time of the request, then the server replies immediately, granting the token.
- If token is currently held by other processes, the server does not reply but queues the request.
- When a client exits the critical section, a message is sent to server, giving it back the token.
- If some processes are waiting in the queue, then the server chooses the oldest entry in the queue, removes it and replies to the corresponding process

Central Server algorithm: managing a mutual exclusion token for a set of processes



ME1: safety
ME2: liveness
Are satisfied but not
ME3: ordering

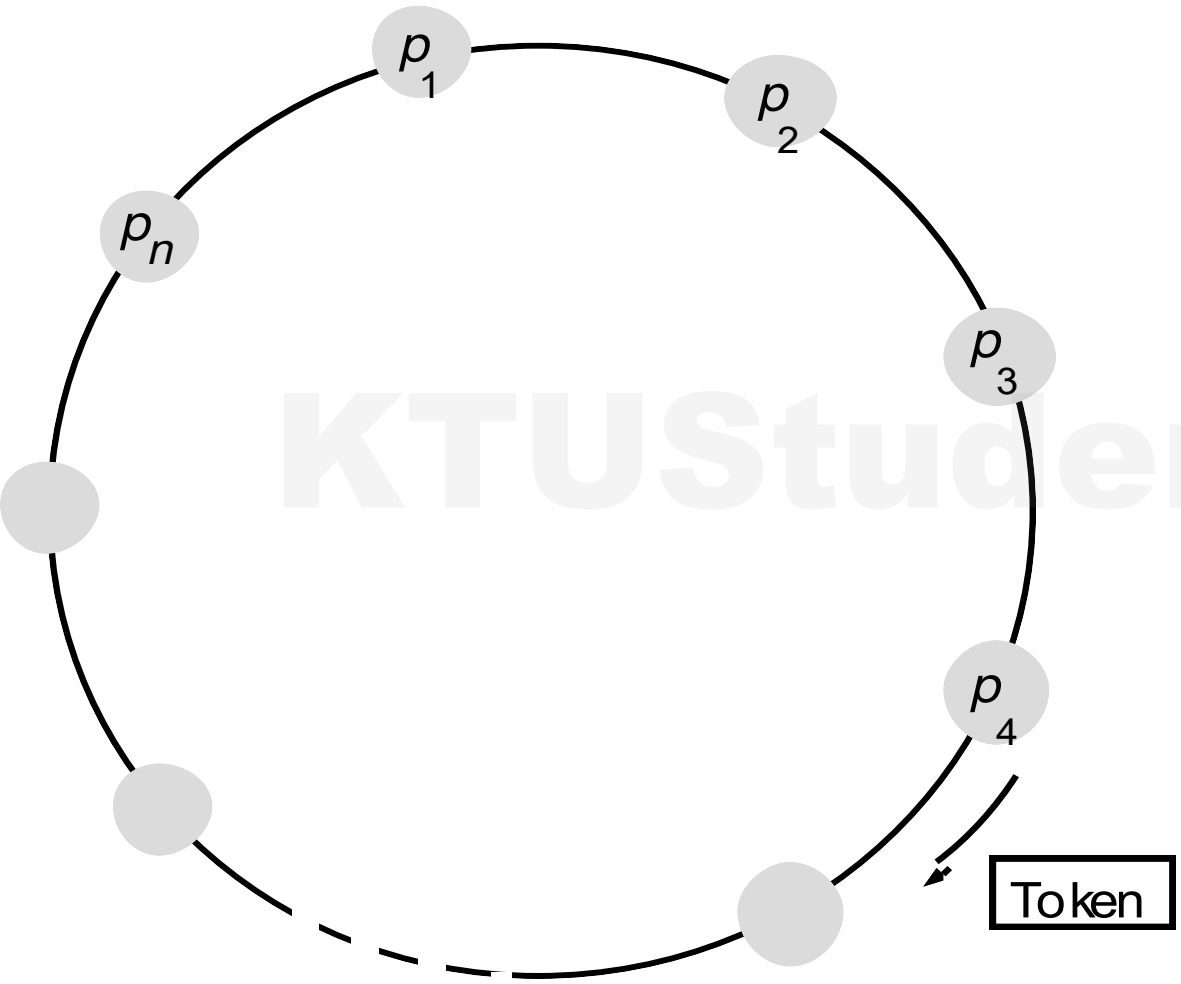
Bandwidth: entering takes two messages (request followed by a grant), **delayed** by the round-trip time; exiting takes one release message, and does **not delay** the exiting process.

Throughput is measured by synchronization delay, the round-trip of a release message and grant message.

Ring-based Algorithm

- Simplest way to arrange mutual exclusion between N processes without requiring an additional process is arrange them in a logical ring.
- Each process p_i has a communication channel to the next process in the ring, $p_{(i+1)/\text{mod } N}$.
- The unique **token** is in the form of a message passed from process to process in a single direction clockwise.
- If a process does not require to enter the CS when it receives the token, then it immediately forwards the token to its neighbor.
- A process requires the token waits until it receives it, but retains it.
- To exit the critical section, the process sends the token on to its neighbor.

A ring of processes transferring a mutual exclusion token



ME1: safety
ME2: liveness
Are satisfied but not
ME3: ordering

Bandwidth: continuously consumes the bandwidth except when a process is inside the CS. Exit only requires one message
Delay: experienced by process is zero message(just received token) to N messages(just pass the token).
Throughput: synchronization delay between one exit and next entry is anywhere from 1(next one) to N (self) message transmission.

Using Multicast and logical clocks

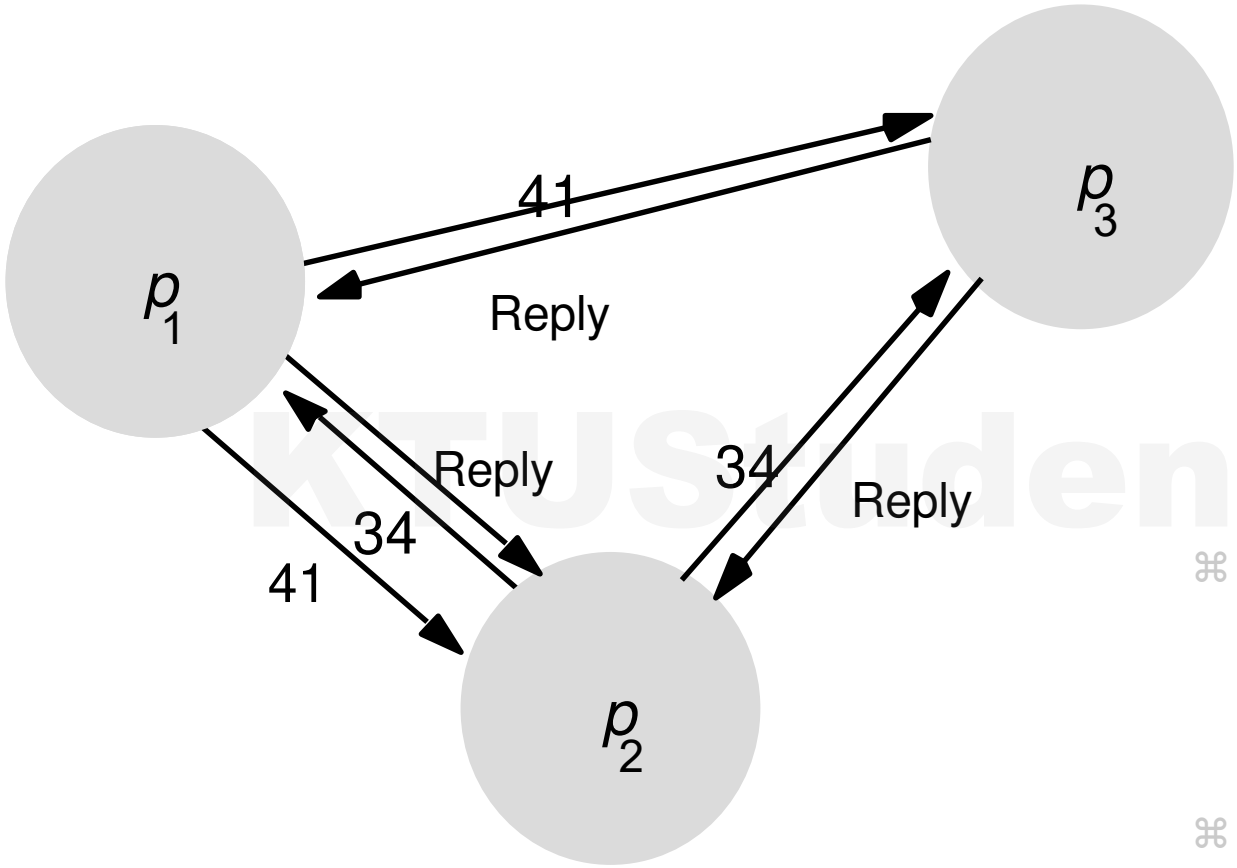
- Mutual exclusion between N peer processes based upon multicast.
- Processes that require entry to a critical section multicast a request message, and can enter it only when all the other processes have replied to this message.
- The condition under which a process replies to a request are designed to ensure ME1 ME2 and ME3 are met.
- Each process pi keeps a Lamport clock. Message requesting entry are of the form<T, pi>.
- Each process records its state of either RELEASE, WANTED or HELD in a variable state.
  - If a process requests entry and all other processes is RELEASED, then all processes reply immediately.
  - If some process is in state HELD, then that process will not reply until it is finished.
  - If some process is in state WANTED and has a smaller timestamp than the incoming request, it will queue the request until it is finished.
  - If two or more processes request entry at the same time, then whichever bears the lowest timestamp will be the first to collect N-1 replies.

Ricart and Agrawala's algorithm

On initialization
    state := RELEASED;
To enter the section
    state := WANTED;
    Multicast *request* to all processes;
    T := request's timestamp;
    } request processing deferred here
    Wait until (number of replies received = (*N* – 1));
    state := HELD;

On receipt of a request $<T_i, p_i>$ *at* p_j $(i \neq j)$
    if (*state* = HELD or (*state* = WANTED *and* $(T, p_j) < (T_i, p_i)$))
    then
        queue *request* from p_i without replying;
    else
        reply immediately to p_i;
    end if
To exit the critical section
    state := RELEASED;
    reply to any queued requests;

Multicast synchronization



- P1 and P2 request CS concurrently. The timestamp of P1 is 41 and for P2 is 34. When P3 receives their requests, it replies immediately. When P2 receives P1's request, it finds its own request has the lower timestamp, and so does not reply, holding P1 request in queue. However, P1 will reply. P2 will enter CS. After P2 finishes, P2 reply P1 and P1 will enter CS.
- Granting entry takes 2(N-1) messages, N-1 to multicast request and N-1 replies. **Bandwidth** consumption is high.
- **Client delay** is again 1 round trip time
- **Synchronization delay** is one message transmission time.

Maekawa's voting algorithm

- It is not necessary for all of its peers to grant access. Only need to obtain permission to enter from subsets of their peers, as long as the subsets used by any two processes overlap.
- Think of processes as voting for one another to enter the CS. A candidate process must collect sufficient votes to enter.
- Processes in the intersection of two sets of voters ensure the safety property ME1 by casting their votes for only one candidate.

Maekawa's voting algorithm

- A voting set V_i is associated with each process p_i.
- There is at least one common member of any two voting sets, the size of all voting set are the same size to be fair.
- The optimal solution to minimizes K is K~sqrt(N) and M=K.

$$V_i \subseteq \{p_1, p_2, ..., p_N\}$$

$$such\ that\ \ for\ all\ i, j = 1,2,...N:$$

$$p_i \in V_i$$

$$V_i \cap V_j \neq \varnothing$$

$$|V_i| = K$$

$$Each\ process\ is\ contained\ in\ M\ \ of\ \ the\ voting\ set\ V_i$$

Maekawa's algorithm

On initialization
  state := RELEASED;
  voted := FALSE;

For p_i to enter the critical section
  state := WANTED;
  Multicast *request* to all processes in V_i;
  Wait until (number of replies received = K);
  state := HELD;

On receipt of a request from p_i at p_j
  if (*state* = HELD *or voted* = TRUE)
  then
    queue *request* from p_i without replying;
  else
    send *reply* to p_i;
    voted := TRUE;
  end if

For p_i to exit the critical section
  state := RELEASED;
  Multicast *release* to all processes in V_i;

On receipt of a release from p_i at p_j
  if (queue of requests is non-empty)
  then
    remove head of queue – from p_k, say;
    send *reply* to p_k;
    voted := TRUE;
  else
    voted := FALSE;
  end if

Maekawa's algorithm

- ME1 is met. If two processes can enter CS at the same time, the processes in the intersection of two voting sets would have to vote for both. The algorithm will only allow a process to make at most one vote between successive receipts of a release message.
- Deadlock prone. For example, p1, p2 and p3 with V1={p1,p2}, V2={p2, p3}, V3={p3,p1}. If three processes concurrently request entry to the CS, then it is possible for p1 to reply to itself and hold off p2; for p2 rely to itself and hold off p3; for p3 to reply to itself and hold off p1. Each process has received one out of two replies, and none can proceed.
- If process queues outstanding request in happen-before order, ME3 can be satisfied and will be deadlock free.
- **Bandwidth** utilization is 2sqrt(N) messages per entry to CS and sqrt(N) per exit.
- **Client delay** is the same as Ricart and Agrawala's algorithm, one round-trip time.
- **Synchronization delay** is one round-trip time which is worse than R&A.

Fault tolerance

- What happens when messages are lost?
- What happens when a process crashes?

- None of the algorithm that we have described would tolerate the loss of messages if the channels were unreliable.
  - The central server algorithm can tolerate the crash failure of a client process that neither holds nor has requested the token.
  - The ring-based algorithm cannot tolerate any single process crash failure.
  - Maekawa's algorithm can tolerate some process crash failures: if a crashed process is not in a voting set that is required.

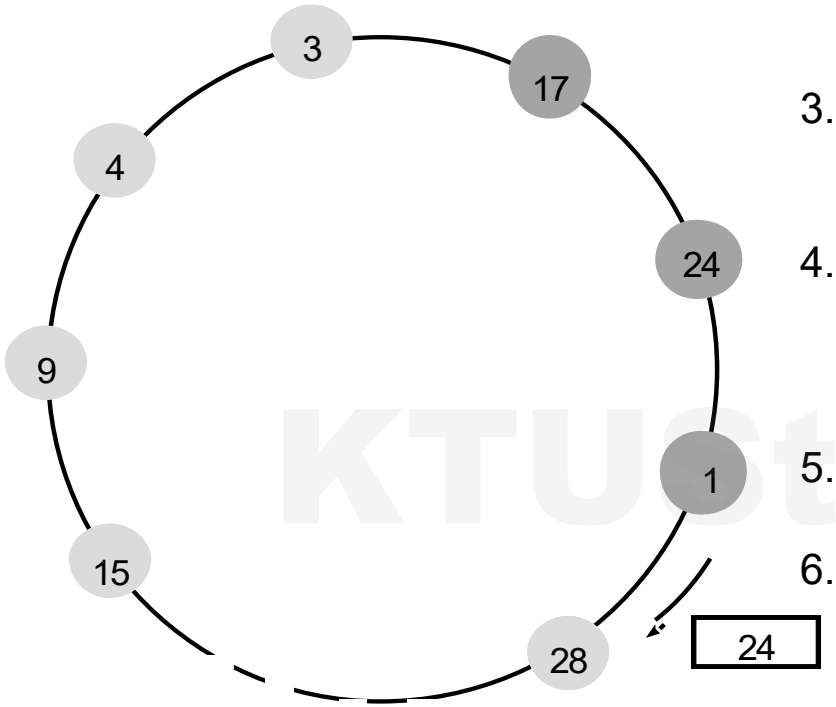
Elections

- Algorithm to choose a unique process to play a particular role is called an election algorithm.
  - e.g. central server for mutual exclusion, one process will be elected as the server. Everybody must agree. If the server wishes to retire, then another election is required to choose a replacement.
- Requirements:
  - E1 (**safety**): a participant p_i has $elected_i = \perp or\ elected_i = P$ where P is chosen as the non-crashed process at the end of run with the largest identifier. (concurrent elections possible.)
  - E2 (**liveness**): All processes P_i participate in election process and eventually set $elected_i \neq \perp or\ crash$

A ring based election algorithm

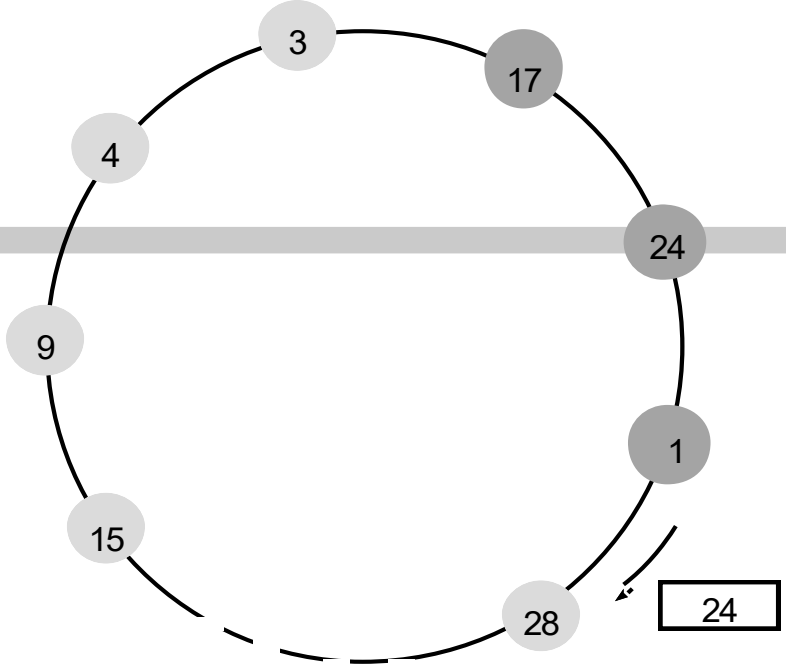
- All processes arranged in a logical ring.
- Each process has a communication channel to the next process.
- All messages are sent clockwise around the ring.
- Assume that no failures occur, and system is asynchronous.
- Goal is to elect a **single** process coordinator which has the largest identifier.

A ring-based election in progress



1. **Initially**, every process is marked as non-participant. Any process can begin an election.
2. The **starting** process marks itself as participant and place its identifier in a message to its neighbour.
3. A process receives a message and **compare** it with its own. If the arrived identifier is **larger**, it passes on the message.
4. If arrived identifier is **smaller** and receiver is not a participant, substitute its own identifier in the message and forward if. It does not forward the message if it is already a participant.
5. On forwarding of any case, the process marks itself as a participant.
6. If the received identifier is that of the receiver itself, then this process's identifier must be the greatest, and it becomes the **coordinator**.
7. The coordinator marks itself as non-participant, set **elected$_i$** and sends an **elected** message to its neighbour enclosing its ID.
8. When a process receives **elected** message, it marks itself as a non-participant, sets its variable **elected$_i$** and forwards the message.

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 darkened

- **E1 is met. All identifiers are compared, since a process must receive its own ID back before sending an elected message.**
- **E2 is also met due to the guaranteed traversals of the ring.**
- **Tolerates no failure :- makes ring algorithm of limited practical use.**

- If only a single process starts an election, the worst-performance case is then the anti-clockwise neighbour has the highest identifier. A total of N-1 messages is used to reach this neighbour. Then further N messages are required to announce its election. The elected message is sent N times. Making **3N-1 messages in all**.
- **Turnaround time** is also 3N-1 sequential message transmission time

The bully algorithm

- Allows process to crash during an election, although it assumes the message delivery between processes is reliable.
- Assume system is synchronous to use timeouts to detect a process failure.
- Assume each process knows which processes have higher identifiers and that it can communicate with all such processes.
- Three types of messages:
  - **Election** is sent to announce an election message. A process begins an election when it notices, through **timeouts**, that the coordinator has failed.
    $T = 2xT_{trans} + T_{process}$ from the time of sending
  - **Answer** is sent in response to an election message.
  - **Coordinator message** is sent to announce the identity of the elected process.

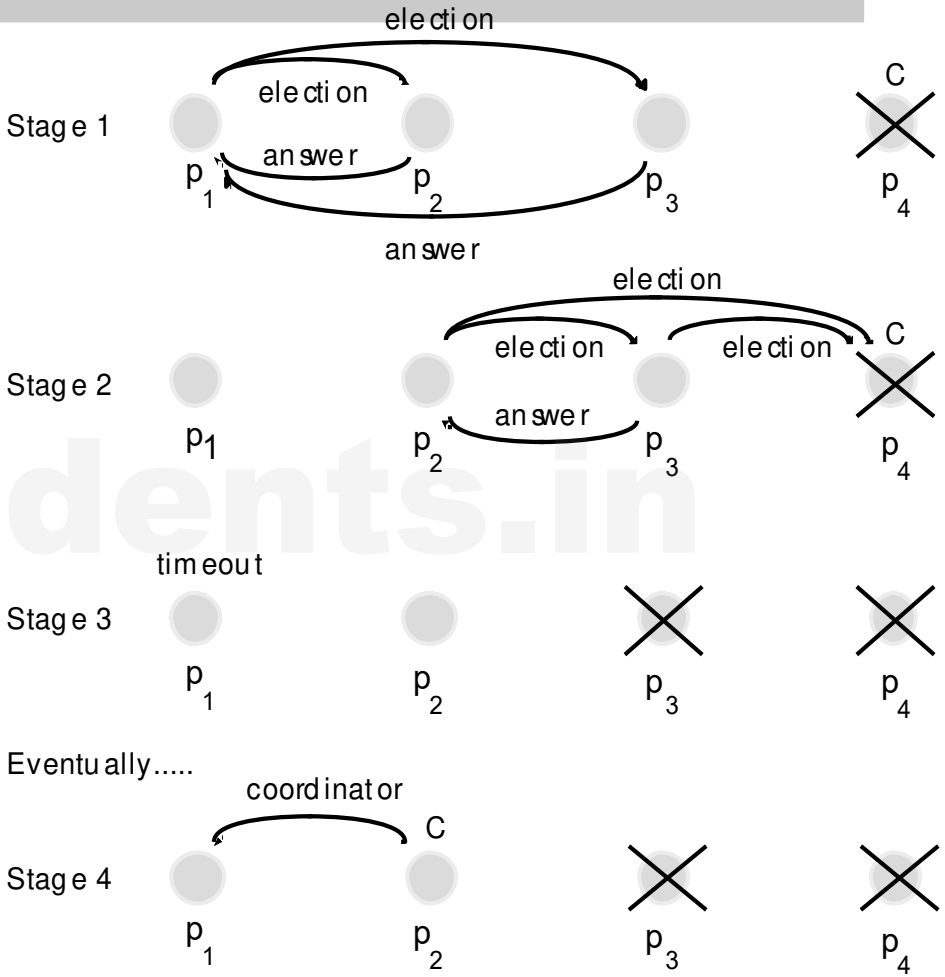
T_{trans} : transmission delay
$T_{process}$: processing delay

How does it start an election?

- The process that knows it has the highest identifier can elect itself as the coordinator simply by sending a *coordinator message to all processes with lower* identifiers.
- On the other hand, a process with a lower identifier can begin an election by sending an *election message to those processes that have a higher identifier*

The bully algorithm

1. The process begins an election by sending an election message to these processes that have a higher ID and awaits an answer in response.
2. If none arrives within time T, the process considers itself the coordinator and sends coordinator message to all processes with lower identifiers.
3. Otherwise, it waits a further time T' for coordinator message to arrive. If none, begins another election.
4. If a process receives a coordinator message, it sets its variable **elected$_i$** to be the coordinator ID.
5. If a process receives an election message, it sends back an answer message and begins another election unless it has begun one already.



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

The bully algorithm

- E1 may be broken if timeout is not accurate or replacement. (suppose P3 crashes and replaced by another process. P2 set P3 as coordinator and P1 set P2 as coordinator)
- E2 is clearly met by the assumption of reliable transmission.
- **Best case** the process with the second highest ID notices the coordinator's failure. Then it can immediately elect itself and send N-2 coordinator messages.
- The bully algorithm requires $O(N^2)$ messages in the **worst case** - that is, when the process with the least ID first detects the coordinator's failure. Because, then N-1 processes altogether begin election, each sending messages to processes with higher ID.