# LECTURE NOTES

# ON

# DISTRIBUTED SYSTEMS

## III B. Tech II semester (JNTUH-R15)


**Mr. RM NOORULLAH**
**Associate Professor**

**Mr. N V KRISHNA RAO**
**Associate Professor**

**Mr. CH SRIKANTH**
**Assistant Professor**

**Mr. M RAKESH**
**Assistant Professor**




**COMPUTER SCIENCE AND ENGINEERING**

**INSTITUTE OF AERONAUTICAL ENGINEERING**

**(AUTONOMOUS)**

**DUNDIGAL, HYDERABAD - 500 043**

# UNIT – I

## Characterization of Distributed System

### Introduction

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

**Distributed systems have the following significant consequences:**

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

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

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

**Examples of distributed systems**

**1. Internet:**

The modern Internet is a vast interconnected collection of computer networks of many different types, with the range of types increasing all the time and now including, for example, a wide range of wireless communication technologies such as WiFi, WiMAX, Bluetooth and third-generation mobile phone networks. The net result is that networking has become a pervasive resource and devices can be connected (if desired) at any time and in any place. The Internet is also a very large distributed system. It enables users, wherever they are, to make use of services such as the World Wide Web, email and file transfer. (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 – sub networks 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. Filtering might be done by source or destination, or a firewall might allow only those messages related

**2. Intranet:**

A portion of the Internet that is separately administered and has a boundary that can be configured to enforce local security policies Composed of several LANs linked by backbone connections Be connected to the Internet via a router.

**3. Mobile and ubiquitous computing:**

Technological advances in device miniaturization and wireless networking have led increasingly to the integration of small and portable computing devices into distributed systems. These devices include: Laptop computers, 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. In mobile computing, users who are away from their 'home' intranet (the intranet at work, or their residence) are still provided with access to resources via the devices they carry with them. They can continue to access the Internet; they can continue to access resources in

their home intranet; and there is increasing provision for users to utilize resources such as printers or even sales points that are conveniently nearby as they move around. The latter is also known as *location-aware* or *context-aware computing*. Mobility introduces a number of challenges for distributed systems, including the need to deal with variable connectivity and indeed disconnection, and the need to maintain operation in the face of device mobility.

**Ubiquitous computing:** is the harnessing of many small, cheap computational devices that are present in users' physical environments, including the home, office and even natural settings. 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 home. Equally, the washing machine could notify the user via a smart badge or phone when the washing is done.

**Resource sharing** is the primary motivation of distributed computing

Resources types

- Hardware, e.g. printer, scanner, camera.
- Data, e.g. file, database, web page.
- More specific functionality, e.g. search engine, file.

**Service**

- Manage a collection of related resources and present their functionalities to users and applications.

**Server**

- A process on networked computer that accepts requests from processes on other computers to perform a service and responds appropriately.

**Client**

- The requesting process.

**Remote invocation**

- A complete interaction between *client* and *server*, from the point when the *client* sends its request to when it receives the server's response.

**Motivation of WWW**

- Documents sharing between physicists of CERN.
- Web is an open system: it can be extended and implemented in new ways without disturbing its existing functionality.
- Its operation is based on communication standards and document standards.
- Respect to the types of 'resource' that can be published and shared on it.

**Hyper Text Mark-up Language**

A language which is used for specifying the contents and layout of web pages.

**Uniform Resource Locators**

Identify documents and other resources client-server architecture with HTTP by with browsers and other clients fetch documents and other resources from web servers.

**Major challenges in Distributed System:**

**1. Heterogeneity :**

The Internet enables users to access services and run applications over a heterogeneous collection of computers and networks. Although the Internet consists of many different sorts of network, their differences are masked by the fact that all of the computers attached to them use the Internet protocols to communicate with one another. For example, a computer attached to an Ethernet has an implementation of the Internet protocols over the Ethernet, whereas a computer on a different sort of network will need an implementation of the Internet protocols for that network. Data types such as integers may be represented in different ways on different sorts of hardware – for example; there are two alternatives for the byte ordering of integers. These differences in representation must be dealt with if messages are to be exchanged between programs running on different hardware. Although the operating systems of all computers on the Internet need to include an implementation of the Internet protocols, they do not necessarily all provide the same application programming interface to these protocols. For example, the calls for exchanging messages in UNIX are different from the calls in Windows.

## 2. 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), is an example. Some middleware, such as Java Remote Method Invocation (RMI), supports only a single programming language. Most middleware is implemented over the Internet protocols, which themselves mask the differences of the underlying networks, but all middleware deals with the differences in operating systems and hardware.

## 3. 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 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. Today, the most commonly used form of mobile code is the inclusion JavaScript programs in some web pages loaded into client browsers.

## 4. Openness :

The openness of a computer system is the characteristic that determines whether the system can be extended and re implemented in various ways. The openness of distributed systems is determined primarily by the degree to which new resource-sharing services can be added and be made available for use by a variety of client programs. Openness cannot be achieved unless the specification and documentation of the key software interfaces of the components of a system are made available to software developers. In a word, the key interfaces are *published*. This process is akin to the standardization of interfaces, but it often bypasses official standardization procedures, which are usually cumbersome and slow-moving. However, the publication of interfaces is only the starting point for adding and extending services in a distributed system. The challenge to designers is to tackle the complexity of distributed systems consisting of many components

engineered by different people. The designers of the Internet protocols introduced a series of documents called 'Requests For Comments', or RFCs, each of which is known by a number. 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.

**5. Security :**

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

In a distributed system, clients send requests to access data managed by servers, which involves sending information in messages over a network. For example: A doctor might request access to hospital patient data or send additions to that data.

**6. 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.

**7. Failure Handling:**

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

**8. Detecting Failures:**

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

**9. Masking failures:**

Some failures that have been detected can be hidden or made less severe. Two examples of hiding failures are messages can be retransmitted when they fail to arrive. File data can be written to a pair of disks so that if one is corrupted, the other may still be correct.

**10. 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 involve the users tolerating them as well. For example, when a web browser cannot contact a web server, it does not make the user wait forever while it keeps on trying – it informs the user about the problem, leaving them free to try again later. Services that tolerate failures are discussed in the paragraph on redundancy below.

**11. Recovery from failures :**

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

**12. Redundancy :**

Services can be made to tolerate failures by the use of redundant components. Consider the following examples: There should always be at least two different routes between any two routers in the Internet. In the Domain Name System, every name table is replicated in at least two different servers. 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.

**13. Concurrency :**

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

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

**Transparency:**

Transparency is defined as the concealment from the user and the application programmer of the separation of components in a distributed system, so that the system is perceived as a whole rather than as a collection of independent components.

The implications of transparency are a major influence on the design of the system software.

**Access transparency:**

It enables local and remote resources to be accessed using identicaloperations.

**Location transparency:**

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

**Concurrency transparency:**

It enables several processes to operate concurrently using sharedresources without interference between them.

**Replication transparency:**

It enables multiple instances of resources to be used to increase reliabilityand performance without knowledge of the replicas by users or application programmers.

**Failure transparency:**

Itenables the concealment of faults, allowing users and application programsto complete their tasks despite the failure of hardware or software components.

**Mobility transparency:**

It allows the movement of resources and clients within a system withoutaffecting the operation of users or programs.

**Performance transparency:**

Itallows the system to be reconfigured to improve performance asloads vary.

**Scaling transparency:**

Itallows the system and applications to expand in scale without change to thesystem structure or the application algorithms.

**Architectural Models**

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

**Software Layers**

The concept of layering is a familiar one and is closely related to abstraction. In a layered approach, a complex system is partitioned into a number of layers, with a given layer making use of the services offered by the layer below. A given layer therefore offers a software abstraction, with higher layers being unaware of implementation details, or indeed of any other layers beneath them. In terms of distributed systems, this equates to a vertical organization of services into service layers. A distributed service can be provided by one or more server processes, interacting with each other and with client processes in order to maintain a consistent system-wide view of the service's resources. For example, a network time service is implemented on the Internet based on the Network Time Protocol (NTP) by server processes running on hosts throughout the Internet that supply the current time to any client that requests it and adjust their version of the current time as a result of interactions with each other. Given the complexity of distributed systems, it is often helpful to organize such services into layers. the important terms *platform* and *middleware*, which define as follows:

**The important terms *platform* and *middleware*, which is defined as**

A platform for distributed systems and applications consists of the lowest-level hardware and software layers. These low-level layers provide services to the layers above them, which are implemented independently in each computer, bringing the system's programming interface up to a level that facilitates communication and coordination between processes. Intel x86/Windows, Intel x86/Solaris, Intel x86/Mac OS X, Intel x86/Linux and ARM/Symbian are major examples.

**System architectures**

**Client-Server:**This is the architecture that is most often cited when distributed systems arediscussed. It is historically the most important and remains the most widely employed. Figure 2.3 illustrates the simple structure in which processes take on the roles of being clients or servers. In particular, client processes interact with individual server processes in potentially separate host computers in order to access the shared resources that they manage.Servers may in turn be clients of other servers, as the figure indicates. For example, a web server is often a client of a local file server that manages the files in which the web pages are stored. Web servers and most other Internet services are clients of the DNS service, which translates Internet domain names to network addresses.Another web-related example concerns *search engines*, which enable users to look up summaries of information available on web pages at sites throughout the Internet. These summaries are made by programs called *web crawlers*, which run in the background at a search engine site using HTTP requests to access web servers throughout the Internet. Thus a search engine is both a server and a client: it responds to queries from browser clients and it runs web crawlers that act as clients of other web servers. In this example, the server tasks (responding to user queries) and the crawler tasks (making requests to other web servers) are entirely independent; there is little need to synchronize them and they may run concurrently. In fact, a typical search engine would normally include many concurrent threads of execution, some serving its clients and others running web crawlers. In Exercise 2.5, the reader is invited to consider the only synchronization issue that does arise for a concurrent search engine of the type outlined here.

**Peer-to-peer:**In this architecture all of the processes involved in a task or activity play similarroles, interacting cooperatively as *peers* without any distinction between client and server processes or the computers on which they run. In practical terms, all participating processes run the same program and offer the same set of interfaces to each other. While the client-server model offers a direct and relatively simple approach to the sharing of data and other resources, it scales poorly. The centralization of service provision and management implied by placing a service at a single address does not scale well beyond the capacity of the computer that hosts the service andthe bandwidth of its network connections. To make explicit all the relevant assumptions about the systems we are modeling.To make generalizations concerning what is possible or impossible, given those assumptions. The generalizations may take the form of general-purpose algorithms or desirable properties that are guaranteed. The guarantees aredependent on logical analysis and,

where appropriate, mathematical proof.The aspects of distributed systems that we wish to capture in our fundamental models are intended to help us to discuss and reason about.

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

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

**Security:** The modular nature of distributed systems and their openness exposes them to attack byboth external and internal agents. Our security model defines and classifies the forms that such attacks may take, providing a basis for the analysis of threats to a system and for the design of systems that are able to resist them.Interaction model fundamentally distributed systems are composed of many processes, interacting in complex ways.

**For example:** Multiple server processes may cooperate with one another to provide a service; the examples mentioned above were the Domain Name System, which partitions and replicates its data at servers throughout the Internet, and Sun's Network InformationService, which keeps replicated copies of password files at several servers in a local area network.A set of peer processes may cooperate with one another to achieve a common goal: for example, a voice conferencing system that distributes streams of audio data in a similar manner, but with strict real-time constraints. Most programmers will be familiar with the concept of an *algorithm* – a sequence of steps to be taken in order to perform a desired computation. Simple programs are controlled by algorithms in which the steps are strictly sequential. The behavior of the program and the state of the program's variables is determined by them. Such a program is executed as a single process. Distributed systems composed of multiple processes such as those outlined above are more complex. Their

behavior and state can be described by a *distributed algorithm* – a definition of the steps to be taken by each of the processes of which the system is composed, *including the transmission of messages between them*. Messages are transmitted betweenprocesses to transfer information between them and to coordinate their activity.Two significant factors affecting interacting processes in a distributed system: Communication performance is often a limiting characteristic. It is impossible to maintain a single global notion of time.

**Performance of communication channels:**The communication channels in our model arerealized in a variety of ways in distributed systems – for example, by an implementation of streams or by simple message passing over a computer network. Communication over a computer network has the following performance characteristics relating to latency, bandwidth and jitter: The delay between the start of a message's transmission from one process and the beginning of its receipt by another is referred to as *latency*. The latency includes:The time taken for the first of a string of bits transmitted through a network to reach itsdestination. For example, the latency for the transmission of a message through a satellite link is the time for a radio signal to travel to the satellite and back.The delay in accessing the network, which increases significantly when the network is heavilyloaded. For example, for Ethernet transmission the sending station waits for the network to be free of traffic.The time taken by the operating system communication services at both the sending and thereceiving processes, which varies according to the current load on the operating systems.The *bandwidth* of a computer network is the total amount of information thatcan be transmitted over it in a given time. When a large number of communication channels are using the same network, they have to share the available bandwidth.*Jitter* is the variation in the time taken to deliver a series of messages. Jitter is relevant tomultimedia 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 owninternal 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.

**Two variants of the interaction model:** In a distributed system it is hard to set limits on thetime that can be taken for process execution, message delivery or clock drift. Two opposing extreme positions provide a pair of simple models – the first has a strong assumption of time and the second makes no assumptions about time:

**Synchronous distributed systems:** Hadzilacos and Toueg define a synchronous distributed system to be one in which the following bounds are defined:

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

**Asynchronous distributed systems:** Many distributed systems, such as the Internet, are very useful without being able to qualify as synchronous systems. Therefore we need an alternative model. An asynchronous distributed system is one in which there are no bounds on:

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

**Message transmission delays:** for example, one message from process A to process B may be delivered in negligible time and another may take several years. In other words, a message may be received after an arbitrarily long time.

**Event ordering:** In many cases, we are interested in knowing whether an event (sending or receiving a message) at one process occurred before, after or concurrently with another event at another process. The execution of a system can be described in terms of events and their ordering despite the lack of accurate clocks. For example, consider the following set of exchanges between a group of email users, X, Y, Z and A, on a mailing list:

- User X sends a message with the subject *Meeting*.
- Users Y and Z reply by sending a message with the subject *Re: Meeting*.

In real time, X's message is sent first, and Y reads it and replies; Z then reads both X's message and Y's reply and sends another reply, which references both X's and Y's messages. But due to the independent delays in message delivery, the messages may be delivered as shown in the following figure and some users may view these two messages in the wrong order.

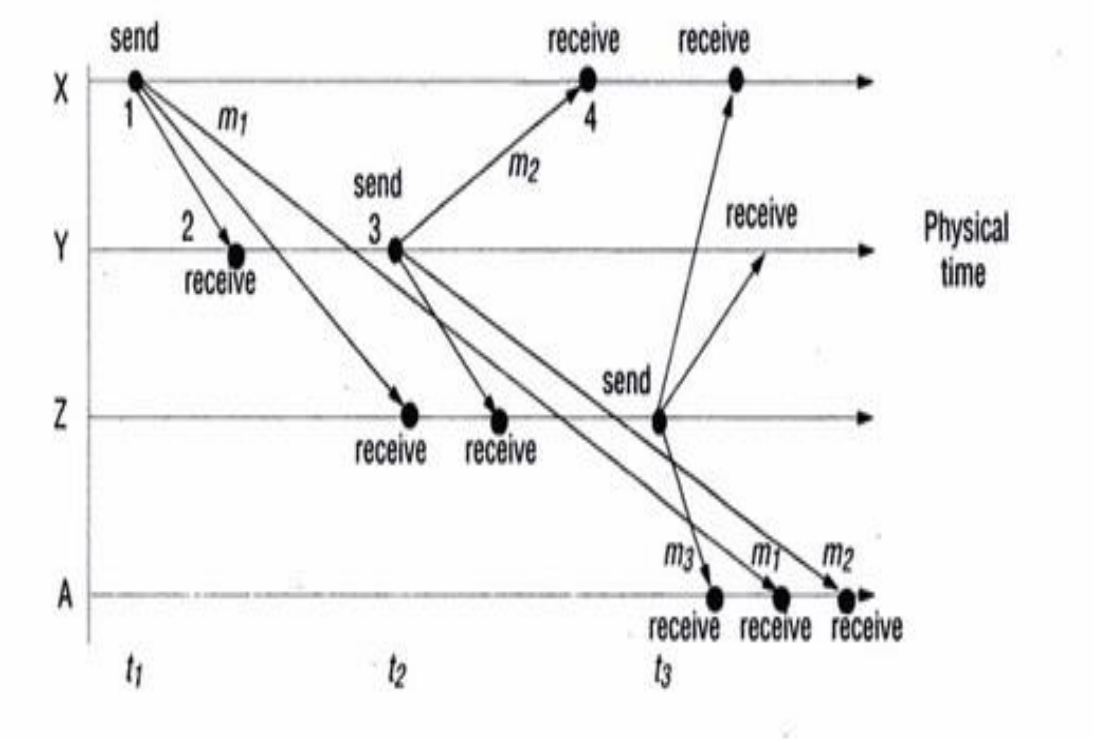


Figure**: Real-Time Ordering of Events**

## UNIT-II

# TIME AND GLOBAL STATES

**Failure model:**

In a distributed system both processes and communication channels may fail – that is, they may depart from what is considered to be correct or desirable behaviour. The failure model defines the ways in which failure may occur in order to provide an understanding of the effects of failures. Hadzilacos and Toueg provide a taxonomy that distinguishes between the failures of processes and communication channels. These are presented under the headings omission failures, arbitrary failures and timing failures.

**Omission failures:**The faults classified as*omission failures*refer to cases when a process orcommunication channel fails to perform actions that it is supposed to do.

**Process omission failures:**The chief omission failure of a process is to crash. When, say that aprocess has crashed we mean that it has halted and will not execute any further steps of its program ever. The design of services that can survive in the presence of faults can be simplified if it can be assumed that the services on which they depend crash cleanly – that is, their processes either function correctly or else stop. Other processes may be able to detect such acrash by the fact that the process repeatedly fails to respond to invocation messages. However, this method of crash detection relies on the use of timeouts – that is, a method in which one process allows a fixed period of time forsomething to occur. In an asynchronous system a timeout can indicate only that a process is not responding – it may have crashed or may be slow, or the messages may not have arrived.

**Communication omission failures**:Consider the communication primitives*send*and*receive*. Aprocess *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. The outgoing andincoming message buffers are typically provided by the operating system.

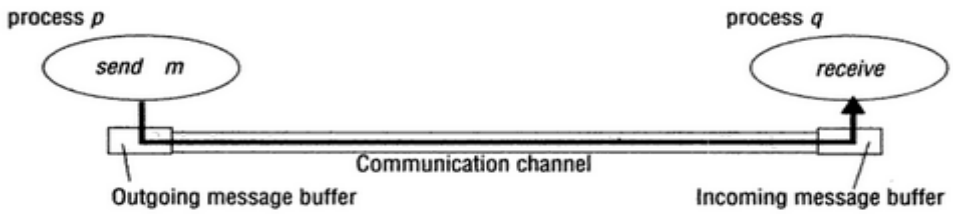


Figure: Processes and Channels

**Arbitrary failures:**

The term*arbitrary*or*Byzantine*failure is used to describe the worstpossible failure semantics, in which any type of error may occur. For example, a process may set wrong values in its data items, or it may return a wrong value in response to an invocation. An arbitrary failure of a process is one in which it arbitrarily omits intended processing steps or takes unintended processing steps. Arbitrary failures in processes cannot be detected by seeing whether the process responds to invocations, because it might arbitrarily omit to reply.Communication channels can suffer from arbitrary failures; for example, message contents may be corrupted, nonexistent messages may be delivered or real messages may be delivered more than once. Arbitrary failures of communication channels are rare because the communication software is able to recognize them and reject the faultymessages. For example, checksums are used to detect corrupted messages, and message sequence numbers can be used to detect nonexistent and duplicated messages.

**Timing failures:**

Timing failures are applicable in synchronous distributed systems where timelimits are set on process execution time, message delivery time and clock drift rate. Timing failures are listed in the following figure. 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.

**Masking failures:**

Each component in a distributed system is generally constructed from acollection of other components. It is possible to construct reliable services from components that exhibit failures. For

example, multiple servers that hold replicas of data can continue to provide a service when one of them crashes. Knowledge of the failure characteristics of a component can enable a new service to be designed to mask the failure of the components on which it depends. A service *masks* a failure either by hiding it altogether or by converting it into a more acceptable type of failure. For an example of the latter, checksums are used to mask corrupted messages, effectively converting an arbitrary failure into an omission failure. The omission failures can be hidden by using a protocol that retransmits messages that do not arrive at their destination. Even process crashes may be masked, by replacing the process and restoring its memory from information stored on disk by its predecessor.The term reliable communicat*ion* is defined in terms of validity and integrity as follows:

**Validity:**

Any message in the outgoing message buffer is eventually delivered to the incomingmessage 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.

**Security model:**

The sharing of resources as a motivating factor for distributed systems, and in Section 2.3 we described their architecture in terms of processes, potentially encapsulating higher-level abstractions such as objects, components or services, and providing access to them through interactions with other processes. That architectural model provides the basis for our security model: the security of a distributed system can be achieved by securing the processes and the channels used for their interactions and by protecting the objects that they encapsulate against unauthorized access. Protection is described in terms of objects, although the concepts apply equally well to resources of all types, Protecting object:Server that manages a collection of objects on behalf of some users. The users can run client programs that send invocations to the server to

perform operations on the objects. The server carries out the operation specified in each invocation and sends the result to the client.Objects are intended to be used in different ways by different users. For example, some objects may hold a user's private data, such as their mailbox, and other objects may hold shared data such as web pages. To support this, *access rights* specify who is allowed to perform the operations of an object – for example, who is allowed to read or to write its state.

**Securing processes and their interactions:**

Processes interact by sending messages. Themessages are exposed to attack because the network and the communication service that they use

**The enemy:**

To model security threats, we postulate an enemy (sometimes also known as theadversary) 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 the following figure. Such attacks can be made simply by using a computer connected to a network to run a program that reads network messages addressed to other computers on the network, or a program that generates messages that make false requests to services, purporting to come from authorized users. The attack may come from a computer that is legitimately connected to the network or from one that is connected in an unauthorized manner. The threats from a potential enemy include *threats toprocesses* and *threats to communication channels*.

**Defeating security threats:**

Cryptography and shared secrets: Suppose that a pair of processes (for example, a particularclient 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:**

This is the science of keeping messages secure, and *encryption* is the process ofscrambling 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 basicauthentication technique is to include in a message an encrypted portion that contains enough of the contents of the message to guarantee its authenticity. The authentication portion of a request to a file server to read part of a file, for example, might include a representation of the requesting principal's identity, the identity of the file and the date and time of the request, all encrypted with a secret key shared between the file server and the requesting process. The server would decrypt this and check that it corresponds to the unencrypted details specified in the request.

**Secure Channels:**

Encryption and authentication are used to build secure channels as a servicelayer on top of existing communication services. A secure channel is a communication channel connecting a pair of processes, each of which acts on behalf of a principal, as shown in the following figure. 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. Communication aspects of middleware, although the principles discussed are more widely applicable. This one is concerned with the design of the components shown in the darker layer.

The application program interface to UDP provides a *message passing* abstraction– the simplest form of interposes communication. This enables a sending process to transmit a single message to a receiving process. The independent packets containing these messages are called *datagram's*. 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. The application program interface to TCP provides the abstraction of a two-way *stream* between pairs of processes. The information communicated consists of a stream of data items with no

message boundaries. Streams provide a building block for producer-consumer communication. A producer and a consumer form a pair of processes in which the role of the first is to produce data items and the role of the second is to consume them. The data items sent by the producer to the consumer are queued on arrival at the receiving host until the consumer is ready to receive them. The consumer must wait when no data items are available. The producer must wait if the storage used to hold the queued data items is exhausted.

## DISTRIBUTED SHARED MEMORY

Distributed shared memory (DSM) is an abstraction used for sharing data between computers that do not share physical memory. Processes access DSM by reads and updates to what appears to be ordinary memory within their address space. However, an underlying runtime system ensures transparently that processes executing at different computers observe the updates made by one another.The main point of DSM is that it spares the programmer the concerns of message passing when writing applications that might otherwise have to use it. DSM is primarily a tool for parallel applications or for any distributed application or group of applications in which individual shared data items can be accessed directly. DSM is in general less appropriate in client-server systems, where clients normally view server-held resources as abstract data and access them by request (for reasons of modularity and protection).Message passing cannot be avoided altogether in a distributed system: in the absence of physically shared memory, the DSM runtime support has to send updates in messages between computers. DSM systems manage replicated data: each computer has a local copy of recently accessed data items stored in DSM, for speed of access.In distributed memory multiprocessors and clusters of off-the-shelf computing components (see Section 6.3), the processors do not share memory but are connected by a very high-speed network. These systems, like general-purpose distributed systems, can scale to much greater numbers of processors than a shared-memory multiprocessor's 64 or so. A central question that has been pursued by the DSM and multiprocessor research communities is whether the investment in knowledge of shared memory algorithms and the associated software can be directly transferred to more scalable distributed memory architecture.

### Message passing versus DSM

As a communication mechanism, DSM is comparable with message passing rather than with request-reply-based communication, since its application to parallel processing, in particular,

entails the use of asynchronous communication. The DSM and message passing approaches to programming can be contrasted as follows:

**Programming model***:*

Under the message passing model, variables have to be marshalled from one process, transmitted and unmarshalled into other variables at the receiving process. By contrast, with shared memory the processes involved share variables directly, so no marshalling is necessary – even of pointers to shared variables – and thus no separate communication operations are necessary.

**Efficiency:**

Experiments show that certain parallel programs developed for DSM can be made to perform about as well as functionally equivalent programs written for message passing platforms on the same hardware – at least in the case of relatively small numbers of computers (ten or so). However, this result cannot be generalized. The performance of a program based on DSM depends upon many factors, as we shall discuss below – particularly the pattern of data sharing.

**Implementation approaches to DSM**

Distributed shared memory is implemented using one or a combination of specialized hardware, conventional paged virtual memory or middleware:

**Hardware:**

Shared-memory multiprocessor architectures based on a NUMA architecture rely on specialized hardware to provide the processors with a consistent view of shared memory. They handlememory LOAD and STORE instructions by communicating with remote memory and cache modules as necessary to store and retrieve data.

**Paged virtual memory*:***

Many systems, including Ivy and Mether , implement DSM as a region of virtual memory occupying the same address range in the address space of every participating process.

*#include "world.h" struct*
*shared { int a, b; };*
*Program Writer:*

```
main()

{

struct shared *p;

methersetup(); /* Initialize the Mether runtime */
p = (struct shared *)METHERBASE;

/* overlay structure on METHER segment */ p-
>a = p->b = 0; /* initialize fields to zero */

while(TRUE){ /* continuously update structure fields */ p
–>a = p –>a + 1;

p –>b = p –>b - 1;

}

}
```

**Program Reader:**

```
main()

{

struct shared *p;
methersetup();

p = (struct shared *)METHERBASE;

while(TRUE){ /* read the fields once every second */
printf("a = %d, b = %d\n", p –>a, p –>b);

sleep(1);

}}
```

**Middleware:**

Some languages such as Orca, support forms of DSM without any hardware or paging support, in a platform-neutral way. In this type of implementation, sharing is implemented by

communication between instances of the user-level support layer in clients and servers. Processes make calls to this layer when they access data items in DSM. The instances of this layer at the different computers access local data items and communicate as necessary to maintain consistency.

**Design and implementation issues:**

The synchronization model used to access DSM consistently at the application level; the DSM consistency model, which governs the consistency of data values accessed from different computers; the update options for communicating written values between computers; the granularity of sharing in a DSM implementation; and the problem of thrashing.

**Structure:**

A DSM system is just such a replication system. Each application process is presented with some abstraction of a collection of objects, but in this case the 'collection' looks more or less like memory. That is, the objects can be addressed in some fashion or other.

Different approaches toDSM vary in what they consider to be an 'object' and in how objects are addressed. We consider three approaches, which view DSM as being composed respectively of contiguous bytes, language-level objects or immutable data items.

**Byte-oriented:**

This type of DSM is accessed as ordinary virtual memory – a contiguous array of bytes. It is the view illustrated above by the Mether system. It is also the view of many other DSM systems, including Ivy.It allows applications (and language implementations) to impose whatever data structures they want on the shared memory. The shared objects are directly addressible memory locations (in practice, the shared locations may be multi-byte words rather than individual bytes). The only operations upon those objects are *read* (or LOAD) and *write* (or STORE). If *x* and *y*are two memory locations, then we denote instances of these operations as follows:

$R(x)a$ – a *read* operation that reads the value $a$ from location $x$.
$W(x)b$ – a *write* operation that stores value $b$ at location $x$.

**Object-oriented:**

The shared memory is structured as a collection of language-level objects with higher-level semantics than simple *read* / *write* variables, such as stacks and dictionaries. The contents of the shared memory are changed only by invocations upon these objects and never by direct access to their member variables. An advantage of viewing memory in this way is that object semantics can be utilized when enforcing consistency.

**Immutable data:**

When reading or taking a tuple from tuple space, a process provides a tuple specification and the tuple space returns any tuple that matches that specification – this is a type of associative addressing. To enable processes to synchronize their activities, the *read* and *take* operations both block until there is a matching tuple in the tuple space.

**Synchronization model:**

Many applications apply constraints concerning the values stored in shared memory. This is as true of applications based on DSM as it is of applications written for sharedmemory multiprocessors (or indeed for any concurrent programs that share data, such as operating system kernels and multi-threaded servers). For example, if *a* and*b* are two variables stored in DSM, then a constraint might be that *a*=*b* always. If two or moreprocesses execute the following code:then an inconsistency may arise. Suppose *a* and*b* are initially zero and that process 1gets as far as setting *a* to 1. Before it can increment *b*, process 2 sets *a* to 2 and *b* to 1.

**Consistency model:**

The local replica manager is implemented by a combination of middleware (the DSM runtime layer in each process) and the kernel. It is usual for middleware to perform the majority of DSM processing. Even in a page-based DSM implementation, the kernel usually provides only basic page mapping, page-fault handling and communication mechanisms and middleware is responsible for implementing the page-sharing policies. If DSM segments are persistent, then one or more storage servers (for example, file servers) will also act as replica managers.

**Sequential consistency:** A DSM system is said to be sequentially consistent if *for any execution* there is some interleaving of the series of operations issued by all the processes that satisfies the following two criteria:

SC1: The interleaved sequence of operations is such that if R(x) a occurs in the sequence, then either the last write operation that occurs before it in the interleaved sequence is W(x) a, or no write operation occurs before it and *a* is the initial value of *x*.

SC2: The order of operations in the interleaving is consistent with the program order in which each individual client executed them.

**Coherence:**

Coherence is an example of a weaker form of consistency. Under coherence, every process agrees on the order of write operations to the same location, but they do not necessarily agree on the ordering of write operations to different locations. We can think of coherence as sequential consistency on a locationby- location basis. Coherent DSM can be implemented by taking a protocol for implementing sequential consistency and applying it separately to each unit of replicated data – for example, each page.

**Weak consistency:**

This model exploits knowledge of synchronization operations in order to relax memory consistency, while appearing to the programmer to implement sequential consistency (at least, under certain conditions that are beyond the scope of this book). For example, if the programmer uses a lock to implement a critical section, then a DSM system can assume that no other process may access the data items accessed under mutual exclusion within it. It is therefore redundant for the DSM system to propagate updates to these items until the process leaves the critical section.While items are left with 'inconsistent' values some of the time, they are not accessed at those points; the execution appears to be sequentially consistent.

**Update options:**

Two main implementation choices have been devised for propagating updates made by one process to the others: write-update and write-invalidate. These are applicable to a variety of DSM consistency models, including sequential consistency. In outline, the options are as follows:

***Write-update*:** The updates made by a process are made locally and multicast to all other replicamanagers possessing a copy of the data item, which immediately modify the data read by local processes. Processes read the local copies of data items, without the need for communication. In addition to allowing multiple readers, several processes may write the same data item at the same time; this is known as multiple-reader/multiple-writer sharing.

***Write-invalidate*:** This is commonly implemented in the form of multiple-reader/ single-writersharing. At any time, a data item may either be accessed in read-only mode by one or more processes, or it may be read and written by a single process. An item that is currently accessed in read-only mode can be copied indefinitely to other processes. When a process attempts to write to it, a multicast message is first sent to all other copies to invalidate them and this is acknowledged before the write can take place; the other processes are thereby prevented from reading stale data (that is, data that are not up to date). Any processes attempting to access the data item are blocked if a writer exists.

**Granularity:**

An issue that is related to the structure of DSM is the granularity of sharing. Conceptually, all processes share the entire contents of a DSM. As programs sharing DSM execute, however, only certain parts of the data are actually shared and then only for certain times during the execution. It would clearly be very wasteful for the DSM implementation always to transmit the entire contents of DSM as processes access and update it.

**Thrashing:**

A potential problem with write-invalidate protocols is thrashing. Thrashing is said to occur where the DSM runtime spends an inordinate amount of time invalidating and transferring shared data compared with the time spent by application processes doing useful work. It occurs when several processes compete for the same data item, or for falsely shared data items.

**The system model:**

The basic model to be considered is one in which a collection of processes shares a segment of DSM. The segment is mapped to the same range of addresses in each process, so that meaningful pointer values can be stored in the segment. The processes execute at computers equipped with a paged memory management unit. We shall assume that there is only one process per computer that accesses the DSM segment. There may in reality be several such processes at a computer. However, these could then share DSM pages directly (the same page frame can be used in the page tables used by the different processes). The only complication would be to coordinate fetching and propagating updates to a page when two or more local processes access it. This description ignores such details. Paging is transparent to the application components within processes; they can logically both read and write any data in DSM. However, the DSM runtime

restricts page access permissions in order to maintain sequential consistency when processing reads and writes. Paged memory management units allow the access permissions to a data page to be set to none, read-only or read-write.

**The problem of write-update:**

The previous section outlined the general implementation alternatives of write-update and write-invalidation. In practice, if the DSM is page-based, then write-update is used only if writes can be buffered. This is because standard page-fault handling is unsuited to the task of processing every single write update to a page.

**Write invalidation:**

Invalidation-based algorithms use page protection to enforce consistent data sharing. When a process is updating a page, it has read and write permissions locally; all other processes have no access permissions to the page. When one or more processes are reading the page, they have read-only permission; all other processes have no access permissions (although they may acquire read permissions). No other combinations are possible.

**Invalidation protocols**

Two important problems remain to be addressed in a protocol to implement the invalidation scheme:

How to locate *owner*(*p*) for a given page *p*.

Where to store *copyset*(*p*).

For Ivy, Li and Hudak [1989] describe several architectures and protocols that take varying approaches to these problems. The simplest we shall describe is their improved centralized manager algorithm. In it, a single server called a manager is used to store the location (transport address) of *owner*(*p*) for every page *p*. The manager could be one of the processes running the application, or it could be any other process. In this algorithm, the set *copyset*(*p*) is stored at *owner*(*p*). That is, the identifiers and transport addresses of the members of*copyset*(*p*) are stored.

**Using multicast to locate the owner:** Multicast can be used to eliminate the manager completely. When a process faults, it multicasts its page request to all the other processes. Only the process that owns the page replies. Care must be taken to ensure correct behaviour if two clients request the

same page at more or less the same time: each client must obtain the page eventually, even if its request is multicast during transfer of ownership.

**A dynamic distributed manager algorithm:**

The owner of a page is located by following chains of hints that are set up as ownership of the page is transferred from computer to computer. The length of the chain – that is, the number of forwarding messages necessary to locate the owner – threatens to increase indefinitely. The algorithm overcomes this by updating the hints as more upto- date values become available. Hints are updated and requests are forwarded as follows: The first three updates follow simply from the protocol for transferring page ownership and providing read-only copies. The rationale for the update when forwarding requests is that, for write requests, the requester will soon be the owner, even though it is not currently. In fact, in Li and Hudak's algorithm, assumed here, the *probOwner*update is made whether the request is for read access or writes access.

**Other Consistency Models**

Models of memory consistency can be divided into *uniform models*, which do not distinguish between types of memory access, and *hybrid models*, which do distinguish between ordinary and synchronization accesses (as well as other types of access).

Other uniform consistency models include:

**Causal consistency**: Reads and writes may be related by the happened-before relationship . This is defined to hold between memory operations when either (a) they are made by the same process; (b) a process reads a value written by another process; or (c) there exists a sequence of such operations linking the two operations. The model's constraint is that the value returned by a read must be consistent with the happened-before relationship.

**Processor consistency:** The memory is both coherent and adheres to the pipelined RAM model (see below). The simplest way to think of processor consistency is that the memory is coherent and that all processes agree on the ordering of any two write accesses made by the same process– that is, they agree with its program order.

**Pipelined RAM:** All processors agree on the order of writes issued by any given processor In addition to release consistency, hybrid models include:

**Entry consistency**: Entry consistency was proposed for the Midway DSM system. In this model, every shared variable is bound to a synchronization object such as a lock, which governs access to that variable. Any process that first acquires the lock is guaranteed to read the latest value of the variable. A process wishing to write the variable must first obtain the corresponding lock in 'exclusive' mode – making it the only process able to access the variable.

Several processes may read the variable concurrently by holding the lock in nonexclusive mode. Midway avoids the tendency to false sharing in release consistency, but at the expense of increased programming complexity.

**Scope consistency:** This memory model [Iftode *et al.* 1996] attempts to simplify theprogramming model of entry consistency. In scope consistency, variables are associated with synchronization objects largely automatically instead of relying on the programmer to associate locks with variables explicitly. For example, the system can monitor which variables are updated in a critical section.

**Weak Consistency:** Weak consistency [Dubois *et al.* 1988] does not distinguish between *acquire* and *release* synchronization accesses. One of its guarantees is that all previous ordinary accesses complete before *either* type of synchronization access completes.

**Common Object Request Broker Architecture (CORBA):**

CORBA is a middleware design that allows application programs to communicate with one another irrespective of their programming languages, their hardware and software platforms, the networks they communicate over and their implementers.Applications are built from CORBA objects, whichimplement interfaces defined in CORBA's interface definition language, IDL. Clients access the methods in the IDL interfaces of CORBA objects by means of RMI. The middleware component that supports RMI is called the Object Request Broker or ORB.

**Introduction**

The OMG (Object Management Group) was formed in 1989 with a view to encouraging the adoption of distributed object systems in order to gain the benefits of object-oriented programming for software development and to make use of distributed systems, which were becoming widespread. To achieve its aims, the OMG advocated the use of open systems based on standard object-oriented interfaces. These systems would be built from heterogeneous hardware, computer networks, operating systems and programming languages.An important motivation was to allow

distributed objects to be implemented in any programming language and to be able to communicate with one another. They therefore designed an interface language that was independent of any specific implementation language. They introduced a metaphor, the *object request broker*(or ORB), whose role is to help a client to invoke a method on an object. This role involves locating the object, activating the object if necessary and then communicating the client's request to the object, which carries it out and replies.In 1991, a specification for an object request broker architecture known as CORBA (Common Object Request Broker Architecture) was agreed by a group of companies. This was followed in 1996 by the CORBA 2.0 specification, which defined standards enabling implementations made by different developers to communicate with one another. These standards are called the General Inter-ORB protocol or GIOP. It is intended that GIOP can be implemented over any transport layer with connections. The implementation of GIOP for the Internet uses the TCP protocol and is called the Internet Inter-ORB Protocol or IIOP [OMG 2004a]. CORBA 3 first appeared in late 1999 and a component model has been added recently. The main components of CORBA's language-independent RMI framework are the following: An interface definition language known as IDL,The GIOP defines an external data representation, called CDR. It also defines specific formats for the messages in a request-reply protocol. In addition to request and reply messages, it specifies messages for enquiring about the location of an object, for cancelling requests and for reporting errors.The IIOP, an implementation of GIOP defines a standard form for remote object references,

**CORBA RMI**

Programming in a multi-language RMI system such as CORBA RMI requires more of the programmer than programming in a single-language RMI system such as Java RMI. The following new concepts need to be learned:the object model offered by CORBA, the interface definition language and its mapping onto the implementation language.

**CORBA's object model:**

The CORBA object model is similar to the one described in , but clients are not necessarily objects – a client can be any program that sends request messages to remote objects and receives replies. The term CORBA object is used to refer to remote objects. Thus, a CORBA object implements an IDL interface, has a remote object reference and is able to respond to invocations of methods in its IDL interface. A CORBA object can be implemented by a language that is not object oriented, for example without the concept of class. Since implementation languages will

have different notions of class or even none at all, the class concept does not exist in CORBA. Therefore classes cannot be defined in CORBA IDL, which means that instances of classes cannot be passed as arguments.

**CORBA IDL**

These are preceded by definitions of two structs, which are used as parameter types in defining the methods. Note in particular that Graphical Object is defined as a struct , whereas it was a class in the Java RMI example. A component whose type is a struct has a set of fields containing values of various types like the instance variables of an object, but it has no methods.

**Parameters and results in CORBA IDL:**

Each parameter is marked as being for input or output or both, using the keywords in, out or inout illustrates a simple example of the use of those keywords. The semantics of parameter passing are as follows:

**Passing CORBA objects:**

Any parameter whose type is specified by the name of an IDL interface, such as the return value*Shape* in line 7, is a reference to a CORBA object and the value of a remote object reference ispassed.

**Passing CORBA primitive and constructed types:**

Arguments of primitive and constructed types are copied and passed by value. On arrival, a new value is created in the recipient's process. For example, the *structGraphicalObject* passed as argument (in line 7) produces a new copy of this *struct* at the server.

**Type Object :**

*Object* is the name of a type whose values are remote object references. It is effectively a common super type of all of IDL interface types such as *Shape* and *ShapeList*.

**Exceptions in CORBA IDL:**

CORBA IDL allows exceptions to be defined in interfaces and thrown by their methods. To illustrate this point, we have defined our list of shapes in the server as a sequence of a fixed length (line 4) and have defined *FullException* (line 6), which is thrown by the method *newShape*(line 7) if the client attempts to add a shape when the sequence is full.

**Invocation semantics:**

Remote invocation in CORBA has at-most-once call semantics as the default. However, IDL may specify that the invocation of a particular method has maybe semantics by using the one way keyword. The client does not block on one way requests, which can be used only for methods without results.

**The CORBA Naming service:**

It is a binder that provides operations including rebind for servers to register the remote object references of CORBA objects by name and resolve for clients to look them up by name. The names are structured in a hierarchic fashion, and each name in a path is inside a structure called a NameComponent. This makes access in a simple example seem rather complex.

**CORBA pseudo objects:**

Implementations of CORBA provide interfaces to the functionality of the ORB that programmers need to use. In particular, they include interfaces to two of the components in the ORB core and the Object Adaptor.

**CORBA client and server example:**

This is followed by a discussion of callbacks in CORBA. We use Java as the client and server languages, but the approach is similar for other languages. The interface compiler *idlj* can be applied to the CORBA interfaces to generate the following items:

The equivalent Java interfaces – two per IDL interface. The name of the first Java interface ends in *Operations* – this interface just defines the operations in the IDL interface. The Java second interface has the same name as the IDL interface and implements the operations in the first interface as well as those in an interface suitable for a CORBA object.

The server skeletons for each *idl* interface. The names of skeleton classes end in *POA* , for example *ShapeListPOA.*

The proxy classes or client stubs, one for each IDL interface. The names of these classes end in *Stub*, for example *_ShapeListStub*

A Java class to correspond to each of the *structs* defined with the IDL interfaces. In our example, classes *Rectangle* and *GraphicalObject* are generated. Each of these classes contains a declaration of one instance variable for each field in the corresponding *struct* and a pair of constructors, but no other methods.

Classes called helpers and holders, one for each of the types defined in the IDL interface. A helper class contains the *narrow* method, which is used to cast down from a given object reference to the class to which it belongs, which is lower down the class hierarchy. For example, the *narrow*

method in *ShapeHelper* casts down to class *Shape* . The holder classes deal with *out* and *inout* arguments, which cannot be mapped directly onto Java.

**Server program:**

The server program should contain implementations of one or more IDL interfaces. For a server written in an object-oriented language such as Java or C++, these implementations are implemented as servant classes. CORBA objects are instances of servant classes.

When a server creates an instance of a servant class, it must register it with the POA, which makes the instance into a CORBA object and gives it a remote object reference. Unless this is done, the CORBA object will not be able to receive remote invocations. Readers who studied Chapter 5 carefully may realize that registering the object with the POA causes it to be recorded in the CORBA equivalent of the remote object table.

**Client program:**

It creates and initializes an ORB (line 1), then contacts the Naming Service to get a reference to the remote *ShapeList* object by using its *resolve* method (line 2). After that it invokes its method *allShapes*(line 3) to obtain a sequence of remote object references to all the *Shapes* currentlyheld at the server. It then invokes the *getAllState* method (line 4), giving as argument the first remote object reference in the sequence returned; the result is supplied as an instance of the*GraphicalObject*class.

**Callbacks:**

Callbacks can be implemented in CORBA in a manner similar to the one described for Java RMI For example, the *WhiteboardCallback* interface may be defined as follows:

```
interfaceWhiteboardCallback
{
    oneway
    void callback(in int version);
};
```

This interface is implemented as a CORBA object by the client, enabling the server to send the client a version number whenever new objects are added. But before the server can do this, the client needs to inform the server of the remote object reference of its object. To make this possible, the *ShapeList*interface requires additional methods such as *register* and *deregister*, as follows:

*int register(in WhiteboardCallback callback); void deregister(in intcallbackId);*

After a client has obtained a reference to the *ShapeList* object and created an instance of *WhiteboardCallback,* it uses the *register* method of*ShapeList*to inform the server that it isinterested in receiving callbacks. The *ShapeList* object in the server is responsible for keeping a list of interested clients and notifying all of them each time its version number increases when a new object is added.

**The Architecture of CORBA**

The architecture is designed to support the role of an object request broker that enables clients to invoke methods in remote objects, where both clients and servers can be implemented in a variety of programming languages. The main components of the CORBA architecture are illustrated in Figure(a). CORBA provides for both static and dynamic invocations. Static invocations are used when the remote interface of the CORBA object is known at compile time, enabling client stubs and server skeletons to be used. If the remote interface is not known at compile time, dynamic invocation must be used. Most programmers prefer to use static invocation because it provides a more natural programming model.

**ORB core:** The role of the ORB core is similar to that of the communication module. In addition, an ORB core provides an interface that includes the following: operations enabling it to be started and stopped operations to convert between remote object references and strings operations to provide argument lists for requests using dynamic invocation.

**Object adapter:** The role of an *object adapter* is to bridge the gap between CORBA objects with IDL interfaces and the programming language interfaces of the corresponding servant classes. This role also includes that of the remote reference and dispatcher modules. An object adapter has the following tasks: it creates remote object references for CORBA objects; it dispatches each RMI via a skeleton to the appropriate servant;it activates and deactivates servants. An object adapter gives each CORBA object a unique *object name,* which forms part of its remote object reference. The same name is used each time an object is activated. The object name may be specified by the

application program or generated by the object adapter. Each CORBA object is registered with its object adapter, which may keep a remote object table that maps the names of CORBA objects to their servants.

**Portable Object Adapter:**

The CORBA 2.2 standard for object adapters is called the Portable Object Adapter. It is called portable because it allows applications and servants to be run on ORBs produced by different developers [Vinoski 1998]. This is achieved by means of the standardization of the skeleton classes and of the interactions between the POA and the servants. The POA supports CORBA objects with two different sorts of lifetimes: those whose lifetimes are restricted to that of the process their servants are instantiated in; those whose lifetimes can span the instantiations of servants in multiple processes.

**Skeletons**

Skeleton classes are generated in the language of the server by an IDL compiler. As before, remote method invocations are dispatched via the appropriate skeleton to a particular servant, and the skeleton unmarshals the arguments in request messages and marshals exceptions and results in reply messages.

**Client stubs/proxies:**

These are in the client language. The class of a proxy (for object oriented languages) or a set of stub procedures (for procedural languages) is generated from an IDL interface by an IDL compiler for the client language. As before, the client stubs/proxies marshal the arguments in invocation requests and unmarshal exceptions and results in replies.

**Implementation repository:**

An implementation repository is responsible for activating registered servers on demand and for locating servers that are currently running. The object adapter name is used to refer to servers when registering and activating them. An implementation repository stores a mapping from the names of object adapters to the pathnames of files containing object implementations. Object implementations and object adapter names are generally registered with the implementation repository when server programs are installed. When object implementations are activated in servers, the hostname and port number of the server are added to the mapping.

## UNIT-3

## INTER PROCESS COMMUNICATION

### API for the Internet protocols:

The general characteristics of interprocess communication and then discuss the Internet protocols as an example, explaining how programmers can use them, either by means of UDP messages or through TCP streams.

### Characteristics of Inter Process Communication:

Message passing among a set of processes can be carried by two message communication methods, send and receive, described in terms of destinations and messages. To communicate, one process sends a message (a sequence of bytes) to a destination and another process at the destination receives the message. This activity requires the communication of data from the sending process to the collecting process and may include the synchronization of the two processes.

### Synchronous and Asynchronous Communication:

A queue is associated with each message destination. Sending processes cause messages to be added to remote queues and receiving processes remove messages from local queues. Communication between the sending and receiving processes may be either synchronous or asynchronous. In the synchronous form of communication, the sending and receiving processes synchronize at every message. In this case, both send and receive are blocking operations. Whenever a send is issued the sending process (or thread) is blocked until the corresponding receive is issued. Whenever a receive is issued by a process (or thread), it blocks until a message arrives. In the asynchronous form of communication, the use of the send operation is no blocking in that the sending process is allowed to proceed as soon as the message has been copied to a local buffer, and the transmission of the message proceeds in parallel with the sending process.

The receive operation can have blocking and non-blocking variants. In the non-blocking variant, the receiving process proceeds with its program after issuing a receive operation, which provides a buffer to be filled in the background, but it must separately receive notification that its buffer has been filled, by polling or interrupt. In a system environment such as Java, which supports multiple threads in a single process, the blocking receive has no disadvantages, for it can

be issued by one thread while other threads in the process remain active, and the simplicity of synchronizing the receiving threads with the incoming message is a substantial advantage. Non-blocking communication appears to be more efficient, but it involves extra complexity in the receiving process associated with the need to acquire the incoming message out of its flow of control. For these reasons, today's systems do not generally provide the no blocking form of receive.

**Message destinations:**

It explains that in the Internet protocols, messages are sent to (Internet address, local port) pairs. A local port is a message destination within a computer, specified as an integer. A port has exactly one receiver but can have many senders. Processes may use multiple ports to receive messages. Any process that knows the number of a port can send a message to it. Servers generally publicize their port numbers for use by clients.

**Reliability:**

As far as the validity property is concerned, a point-to-point message service can be described as reliable if messages are guaranteed to be delivered despite a 'reasonable' number of packets being dropped or lost. In contrast, a point-to-point message service can be described as unreliable if messages are not guaranteed to be delivered in the face of even a single packet dropped or lost. For integrity, messages must arrive uncorrupted and without duplication.

Ordering: Some applications require that messages be delivered in sender order – that is, 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 forms of communication (UDP and TCP) use the socket abstraction, which provides an endpoint for communication between processes. Sockets originate from BSD UNIX but are also present in most other versions of UNIX, including Linux as well as Windows and the Macintosh OS. Interposes communication consists of transmitting a message between a socket in one process and a socket in another process, is shown in the. For a process to receive messages, its socket must be bound to a local port and one of the Internet addresses of the computer on which it runs. Messages sent to a particular Internet address and port number can be received only by a process whose socket is associated with that Internet address and port number. Processes may use the same

socket for sending and receiving messages. Each computer has a large number (216) of possible port numbers for use by local processes for receiving messages. Any process may make use of multiple ports to receive messages, but a process cannot share ports with other processes on the same computer. However, 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: As the IP packets underlying UDP and TCP are sent to Internet addresses, Java provides a class, InetAddress, that represents Internet addresses. Users of this class refer to computers by Domain Name System (DNS) hostnames. For example, instances of InetAddress that contain Internet addresses can be created by calling a static method of InetAddress, giving a DNS hostname as the argument. The method uses the DNS to get the corresponding Internet address. For example, to get an object representing the Internet address of the host whose DNS name is bruno.dcs.qmul.ac.uk, use:

```
InetAddressaComputer = InetAddress.getByName("bruno.dcs.qmul.ac.uk");
```

This method can throw an UnknownHostException. Note that the user of the class does not need to state the explicit value of an Internet address. In fact, the class encapsulates the details of the representation of Internet addresses. Thus the interface for this class is not dependent on the number of bytes needed to represent Internet addresses – 4 bytes in IPv4 and 16 bytes in IPv6.

**UDP Datagram Communication:**

A datagram sent by UDP is transmitted from a sending process to a receiving process without acknowledgement or retries. If a failure occurs, the message may not arrive. A datagram is transmitted between processes when one process sends it and another receives it. To send or receive messages a process must first create a socket bound to an Internet address of the local host and a local port. A server will bind its socket to a server port – one that it makes known to clients so that they can send messages to it. A client binds its socket to any free local port. The receive method returns the Internet address and port of the sender, in addition to the message, allowing the recipient to send a reply.

The following are some issues relating to datagram communication:

**Message size:** The receiving process needs to specify an array of bytes of a particular size in which to receive a message. If the message is too big for the array, it is truncated on arrival. The underlying IP protocol allows packet lengths of up to 216 bytes, which includes the headers as well

as the message. However, most environments impose a size restriction of 8 kilobytes. Any application requiring messages larger than the maximum must fragment them into chunks of that size. Generally, an application, for example DNS, will decide on a size that is not excessively large but is adequate for its intended use.

**Blocking:** Sockets normally provide non-blocking sends and blocking receives for datagram communication (a non-blocking receive is an option in some implementations). The send operation returns when it has handed the message to the underlying UDP and IP protocols, which are responsible for transmitting it to its destination. On arrival, the message is placed in a queue for the socket that is bound to the destination port. The message can be collected from the queue by an outstanding or future invocation of receive on that socket. Messages are discarded at the destination if no process already has a socket bound to the destination port.

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

**Receive from any:** The receive method does not specify an origin for messages. Instead, an invocation of receive gets a message addressed to its socket from any origin. The receive method returns the Internet address and local port of the sender, allowing the recipient to check where the message came from. It is possible to connect a datagram socket to a particular remote port and Internet address, in which case the socket is only able to send messages to and receive messages from that address.

**Failure model for UDP datagram's:** A failure model for communication channels and defines reliable communication in terms of two properties: integrity and validity. The integrity property requires that messages should not be corrupted or duplicated. The use of a checksum ensures that there is a negligible probability that any message received is corrupted. UDP datagrams suffer from the following failures:

Omission failures: Messages may be dropped occasionally, either because of a checksum error or because no buffer space is available at the source or destination. To simplify the discussion, we

regard send-omission and receive-omission failures as omission failures in the communication channel.

**Ordering:** Messages can sometimes be delivered out of sender order. Applications using UDP datagrams are left to provide their own checks to achieve the quality of reliable communication they require. A reliable delivery service may be constructed from one that suffers from omission failures by the use of acknowledgements.

**Use of UDP:** For some applications, it is acceptable to use a service that is liable to occasional omission failures. For example, the Domain Name System, which looks up DNS names in the Internet, is implemented over UDP. Voice over IP (VOIP) also runs over UDP. UDP datagrams are sometimes an attractive choice because they do not suffer from the overheads associated with guaranteed message delivery. There are three main sources of overhead: the need to store state information at the source and destination; the transmission of extra messages; latency for the sender.

Java API for UDP datagrams The Java API provides datagram communication by means of two classes: Datagram Packet and Datagram Socket. 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 Internet address and local port number of the destination socket, as follows:

**Datagram packet:** array of bytes containing message length of message Internet address port number, An instance of Datagram Packet may be transmitted between processes when one process sends it and another receives it. UDP server repeatedly receives a request and sends it back to the client.

**Datagram Socket:** This class supports sockets for sending and receiving UDP datagrams. It provides a constructor that takes a port number as its argument, for use by processes that need to use a particular port. It also provides a no-argument constructor that allows the system to choose a free local port. These constructors can throw a Socket Exception if the chosen port is already in use or if a reserved port (a number below 1024) is specified when running over UNIX. UDP server repeatedly receives a request and sends it back to the client.

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

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

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

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

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

**Message destinations:** A pair of communicating processes establishes a connection before they can communicate over a stream. Once a connection is established, the processes simply read from and write to the stream without needing to use Internet addresses and ports. Establishing a connection involves a connect request from client to server followed by an accept request from server to client before any communication can take place. This could be a considerable overhead for a single client-server request and reply.

**Java API for TCP streams:**The Java interface to TCP streams is provided in the classes.

**Server Socket and Socket:**

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

**Socket:** This class is for use by a pair of processes with a connection. The client uses a constructor to create a socket, specifying the DNS hostname and port of a server. This constructor not only creates a socket associated with a local port but also connects it to the specified remote computer and port number. It can throw an Unknown Host Exception if the hostname is wrong or an IO Exception if an IO error occurs. TCP client makes connection to server, sends request and receives reply, TCP server makes a connection for each client and then echoes the client's request

**External data representation and marshalling:**

The information stored in running programs is represented as data structures – for example, by sets of interconnected objects – whereas the information in messages consists of sequences of bytes. Irrespective of the form of communication used, the data structures must be flattened (converted to a sequence of bytes) before transmission and rebuilt on arrival. The individual primitive data items transmitted in messages can be data values of many different types, and not all computers store primitive values such as integers in the same order. The representation of floating-point numbers also differs between architectures. There are two variants for the ordering of integers: the so-called big-endian order, in which the most significant byte comes first; and little-endian order, in which it comes last.

Another issue is the set of codes used to represent characters: for example, the majority of applications on systems such as UNIX use ASCII character coding, taking one byte per character, whereas the Unicode standard allows for the representation of texts in many different languages and takes two bytes per character. One of the following methods can be used to enable any two computers to exchange binary data values: The values are converted to an agreed external format before transmission and converted to the local form on receipt; if the two computers are known to be the same type, the conversion to external format can be omitted. The values are transmitted in the sender's format, together with an indication of the format used, and the recipient converts the values if necessary. Note, however, that bytes themselves are never altered during transmission. To support RMI or RPC, any data type that can be passed as an argument or returned as a result must be able to be flattened and the individual primitive data values represented in an agreed format. An agreed standard for the representation of data structures and primitive values is called an external data representation.

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

**CORBA's:** common data representation, which is concerned with an external representation for the structured and primitive types that can be passed as the arguments and results of remote method invocations in CORBA. It can be used by a variety of programming languages.

**XML (Extensible Markup Language):**The language which defines a textual format for representing structured data. It was originally intended for documents containing textual self-describing structured data – for example documents accessible on the Web – but it is now also used to represent the data sent in messages exchanged by clients and servers in web services. In the first two cases, the marshalling and unmarshalling activities are intended to be carried out by a middleware layer without any involvement on the part of the application programmer. Even in the case of XML, which is textual and therefore more accessible to hand-encoding, software for marshalling and unmarshalling is available for all commonly used platforms and programming environments. Because marshalling requires the consideration of all the finest details of the representation of the primitive components of composite objects, the process is likely to be error-prone if carried out by hand. Compactness is another issue that can be addressed in the design of automatically generated marshalling procedures. In the first two approaches, the primitive data types are marshalled into a binary form. In the third approach (XML), the primitive data types are represented textually. The textual representation of a data value will generally be longer than the equivalent binary representation. The HTTP protocol, which is described in Chapter 5, is another example of the textual approach. Another issue with regard to the design of marshalling methods is whether the marshalled data should include information concerning the type of its contents. For example, CORBA's representation includes just the values of the objects transmitted, and nothing about their types. On the other hand, both Java serialization and XML do include type information, but in different ways. Java puts all of the required type information into the serialized form, but XML documents may refer to externally defined sets of names (with types) called namespaces.

**CORBA's Common Data Representation (CDR)**

**CORBA CDR for constructed types:**

CORBA CDR is the external data representation defined with CORBA 2.0. CDR can represent all of the data types that can be used as arguments and return values in remote invocations in CORBA. These consist of 15 primitive types, which include short (16-bit), long (32-bit), unsigned short, unsigned long, float (32-bit), double (64-bit), char, boolean (TRUE, FALSE), octet (8-bit), and any (which can represent any basic or constructed type); together with a range of composite types, which are described in Figure 4.7. Each argument or result in a remote invocation is represented by a sequence of bytes in the invocation or result message.

Marshalling in CORBA: Marshalling operations can be generated automatically from the specification of the types of data items to be transmitted in a message. The types of the data structures and the types of the basic data items are described in CORBA IDL, which provides a notation for describing the types of the arguments and results of RMI methods.

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

**Extensible Markup Language (XML):** XML is a markup language that was defined by the World Wide Web Consortium (W3C) for general use on the Web. In general, the term markup language refers to a textual encoding that represents both a text and details as to its structure or its appearance. Both XML and HTML were derived from SGML (Standardized Generalized Markup Language) [ISO 8879], a very complex markup language. HTML was designed for defining the appearance of web pages. XML was designed for writing structured documents for the Web. XML data items are tagged with 'markup' strings. The tags are used to describe the logical structure of the data and to associate attribute-value pairs with logical structures. That is, in XML, the tags relate to the structure of the text that they enclose, in contrast to HTML, in which the tags specify how a browser could display the text. For a specification of XML, see the pages on XML provided by W3C []. XML is used to enable clients to communicate with web services and for defining the interfaces and other properties of web services. However, XML is also used in many other ways, including in archiving and retrieval systems – although an XML archive may be larger than a binary one, it has the advantage of being readable on any computer. Other examples of uses of

XML include for the specification of user interfaces and the encoding of configuration files in operating systems.

**XML definition of the Person structure.**

**Remote object references:**

Java and CORBA that support the distributed object model. It is not relevant to XML. When a client invokes a method in a remote object, an invocation message is sent to the server process that hosts the remote object. This message needs to specify which particular object is to have its method invoked. A remote object reference is an identifier for a remote object that is valid throughout a distributed system. A remote object reference is passed in the invocation message to specify which object is to be invoked. Chapter 5 explains that remote object references are also passed as arguments and returned as results of remote method invocations, that each remote object has a single remote object reference and that remote object references can be compared to see whether they refer to the same remote object. Here, we discuss the external representation of remote object references.

Client-server communication: public byte[] disoperation (RemoteObjectRef o, intmethodId, byte[] arguments) sends a request message to the remote object and returns the reply. The arguments specify the remote object, the method to be invoked and the arguments of that method. public byte[] getRequest (); acquires a client request via the server port. public void sendReply (byte[] reply, InetAddressclientHost, intclientPort); sends the reply message reply to the client at its Internet address and port.

**Group communication:**

A multicast operation is more appropriate – this is an operation that sends a single message from one process to each of the members of a group of processes, usually in such a way that the membership of the group is transparent to the sender. There is a range of possibilities in the desired behavior of a multicast. The simplest multicast protocol provides no guarantees about message delivery or ordering. Multicast messages provide a useful infrastructure for constructing distributed systems with the following characteristics:

**Fault tolerance based on replicated services:** A replicated service consists of a group of servers. Client requests are multicast to all the members of the group, each of which performs an identical operation. Even when some of the members fail, clients can still be served.

Discovering services in spontaneous networking: Section 1.3.2 defines service discovery in the context of 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.

**Better performance through replicated data:**Data are replicated to increase the performance of a service – in some cases replicas of the data are placed in users' computers. Each time the data changes, the new value are multicast to the processes managing the replicas.

**Propagation of event notifications:** Multicast to a group may be used to notify processes when something happens. For example, in Facebook, when someone changes their status, all their friends receive notifications. Similarly, publish subscribe protocols may make use of group multicast to disseminate events to subscribers

IP multicast – An implementation of multicast communication

**IP multicast:** IP multicast is built on top of the Internet Protocol (IP). Note that IP packets are addressed to computers – ports belong to the TCP and UDP levels. IP multicast allows the sender to transmit a single IP packet to a set of computers that form a multicast group. The sender is unaware of the identities of the individual recipients and of the size of the group. A multicast group is specified by a Class D Internet address – that is, an address whose first 4 bits are 1110 in IPv4. At the application programming level, IP multicast is available only via UDP. An application program performs multicasts by sending UDP datagrams with multicast addresses and ordinary port numbers. It 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. The following details are specific to IPv4.

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

Multicast address allocation: As discussed in Chapter 3, Class D addresses (that is, addresses in the range 224.0.0.0 to 239.255.255.255) are reserved for multicast traffic and managed globally by the Internet Assigned Numbers Authority (IANA). The management of this address space is reviewed annually, with current practice documented in RPC 3171. This document defines a partitioning of this address space into a number of blocks, including:

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).

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

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

**Multicast peer joins a group and sends and receives datagrams:**

Fault tolerance based on replicated services: Consider a replicated service that consists of the members of a group of servers that start in the same initial state and always perform the same operations in the same order, so as to remain consistent with one another. This application of multicast requires that either all of the replicas or none of them should receive each request to perform an operation – if one of them misses a request, it will become inconsistent with the others.

In most cases, this service would require that all members receive request messages in the same order as one another.

**Discovering services in spontaneous networking:** One way for a process to discover services in spontaneous networking is to multicast requests at periodic intervals, and for the available services to listen for those multicasts and respond. An occasional lost request is not an issue when discovering services.

**Better performance through replicated data:** Consider the case where the replicated data itself, rather than operations on the data, are distributed by means of multicast messages. The effect of lost messages and inconsistent ordering would depend on the method of replication and the importance of all replicas being totally up-to-date.

**Propagation of event notifications:** The particular application determines the qualities required of multicast. For example, the Jini lookup services use IP multicast to announce their existence

**Introduction:** Programming Models for Distributed Communications

**Remote Procedure Calls:** Client programs call procedures in server programs.

**Remote Method Invocation:**Objects invoke methods of objects on distributed hosts.

**Event-based Programming Model:**Objects receive notice of events in other objects in which they have interest.

**Middleware:** Middleware: software that allows a level of programming beyond processes and message passing .Uses protocols based on messages between processes to provide its higher-level abstractions such as remote invocation and events Supports location transparency. Usually uses an interface definition language (IDL) to define interfaces

**Interfaces in Programming Languages:** Current PL allow programs to be developed as a set of modules that communicate with each other. Permitted interact ions between modules are defined by interfaces. A specified interface can be implemented by different modules without the need to modify other modules using the interface.

**Interfaces in Distributed Systems:** When modules are in different processes or on different hosts there are limitations on the interactions that can occur. Only actions with parameters that are fully specified and understood can communicate effectively to request or provide services to modules in

another process. A service interface allows a client to request and a server to provide particular services. A remote interface allows objects to be passed as arguments to and results from distributed modules.

Object Interfaces An interface defines the signatures of a set of methods, including arguments, argument types, return values and exceptions. Implementation details are not included in an interface. A class may implement an interface by specifying behavior for each method in the interface. Interfaces do not have constructors.

**Communication between Distributed Objects:**

**The Object Model**

**Object References:** Objects are accessed by object references. Object references can be assigned to variables, passed as arguments, and returned as the result of a method. Can also specify a method to be invoked on that object.

**Interfaces:** Provide a definition of the signatures of a set of methods without specifying their implementation. Define types that can be used to declare the type of variables or of the parameters and return values of methods

**Actions:** Objects invoke methods in other objects. An invocation can include additional information as arguments to perform the behavior specified by the method Effects of invoking a method The state of the receiving object may be changed A new object may be instantiated Further invocations on methods in other objects may occur An exception may be generated if there is a problem encountered

**Exceptions:** Provide a clean way to deal with unexpected events or errors. A block of code can be defined to throw an exception when errors or unexpected conditions occur. Then control passes to code that catches the exception

**Garbage Collection:** Provide a means of freeing the space that is no longer needed. Java (automatic), C++ (user supplied)

**Distributed Objects:** Physical distribution of objects into different processes or computers in a distributed system. Object state consists of the values of its instance variables. Object methods invoked by remote method invocation (RMI). Object encapsulation: object state accessed only by the object methods

**Design Issues for RMI:** Two design issues that arise in extension of local method invocation for RMI. The choice of invocation semantics. Although local invocations are executed exactly once, this cannot always be the case for RMI due to transmission error.

Either request or reply message may be lost, either server or client may be crashed and The level of transparency. Make remote invocation as much like local invocation as possible

**RMI Design Issues: Invocation Semantics**

**Error handling for delivery guarantees:**

- Retry request message: whether to retransmit the request message until either a reply is received or the server is assumed to have failed
- Duplicate filtering: when retransmissions are used, whether to filter out duplicate requests at the server
- Retransmission of results: whether to keep a history of result messages to enable lost results to be retransmitted without re-executing the operations.

Choices of invocation semantics:

- Maybe: the method executed once or not at all (no retry nor retransmit).
- At-least-once: the method executed at least once.
- At-most-once: the method executed exactly once.

RMI Design Issues: Transparency: Transparent remote invocation: like a local call

- Marshalling/unmarshalling.
- Locating remote objects.
- Accessing/syntax.

**Differences between local and remote invocations:**

**Latency:** a remote invocation is usually several orders of magnitude greater than that of a local one.
**Availability:** remote invocation is more likely to fail.
**Errors/exceptions:** failure of the network? Server? Hard to tell syntax might need to be different to handle different local vs. remote errors/exceptions (e.g. Argus) consistency on the remote machine.

**Implementation of RMI , Communication module:**

- Two cooperating communication modules carry out the request-reply protocols: message type, request ID, remote object reference, Transmit request and reply, messages between client and server.
- Implement specific invocation semantics.
- The communication module in the server selects the dispatcher for the class of the object to be invoked, passes on local reference from remote reference module, returns request.

# UNIT-4

# DISTRIBUTED FILE SYSTEM

A file system is efficient for the system, storage, retrieval, naming, sharing, and protection of files. File systems give directory services, which transform a file name (possibly a hierarchical one) into an internal identifier (e.g. Inode, FAT index). They comprise a representation of the file data itself and techniques for accessing it (read/write). The file system is efficient for managing access to the data and for making low-level operations such as buffering frequently used data and resulting disk I/O requests.

A distributed file system is to determine several degrees of transparency to the user and the system:

**Access transparency:** Clients are oblivious that files are distributed and can obtain them in the same fashion as local files are accessed.

**Location transparency:** A logical namespace exists comprising local as well as remote files. The file name does not provide it location.

**Concurrency transparency:** Whole clients have the identical view of the state of the file system. This suggests that if one process is altering a file, any other processes on the same system or remote systems that are accessing the files will see the changes in a consistent manner.

**Failure transparency:** The client and client applications should behave correctly after a server failure.

**Heterogeneity:** File service should be implemented over various hardware and operating system platforms.

**Scalability:** The file system should operate properly in small environments (1 machine, a dozen machines) and also compare appropriately to huge ones (hundreds through tens of thousands of systems).

**Replication transparency:** To maintain scalability, we may need to replicate files over multiple servers. Clients should be ignorant of this.

**Migration transparency:** Files should be available to move around without the client's knowledge. It maintains the fine-grained distribution of data: Performance optimization will be achieved when we determine individual objects near the processes that use them.

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

**File service types:** To implement a remote system with file service, we have to choose one of two models of the method. One of these is the upload/download method. In this model, there are two fundamental methods: read file transfers an total file from the server to the soliciting client, and compose file copies the file back to the server. It is a simple design and effective in that it provides local access to the file when it is being applied. Three problems are obvious. It can be wasteful if the client wants access to only a small portion of the file data. It can be problematical if the client doesn't have sufficient space to cache the entire file. Eventually, what happens if others need to alter the same file. The second type is a remote access model. The file service implements remote operations such as open, close, read bytes, write bytes, get attributes, etc. The file system itself works on servers. The disadvantage of this method is the servers are obtained for the duration of file access preferably than once to download the file and again to upload it. Another significant characteristic in providing file service is that of explaining the difference between directory service and file service. A directory service, in the setting of file systems, maps human-friendly textual titles for files to their internal locations, which can be used by the file service. The file service itself presents the file interface (this is mentioned above). Another element of file distributed file systems is the client module. This is the client-side interface for file and directory service. It implements a local file system interface to client software (for instance, the node file system layer of a UNIX kernel).

**File service architecture:** This is a general architectural model that underpins both NFS and AFS. It is based upon a distribution of responsibilities among three modules. A client module that follows a conventional file system interface for application programs, and server modules, that make operations for clients on directories and on files. The architecture is intended to allow a stateless implementation of the server module.

**SUN NFS**: Sun Microsystems's Network File System (NFS) has been widely adopted in industry and in academic environments since its introduction in 1985. The design and development of NFS

were undertaken by staff at Sun Microsystems in 1984. Although several distributed file services had already been developed and used in universities and research laboratories, NFS was the first file service that was designed as a product. The design and implementation of NFS have achieved success both technically and commercially.

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

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

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

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

**Flat file service interface:**

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

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

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

- A similar construct is used in a UNIX file system.
- It helps with distributing the load of file serving between several servers.

- File groups have identifiers which are unique throughout the system (and hence for an open system, they must be globally unique).

To construct globally unique ID we use some unique attribute of the machine on which it is created. Ex: IP number, even though the file group may move subsequently.

**NFS access control and authentication:**

The NFS server is stateless server, so the user's identity and access rights must be checked by the server on each request. In the local file system they are checked only on the file's access permission attribute. Every client request is accompanied by the user ID and group ID. It is not shown in the Figure 8.9 because they are inserted by the RPC system. Kerberos has been integrated with NFS to provide a stronger and more comprehensive security solution.

**Mount service**

**Mount operation:**mount( remote host, remote directory, local directory)

**The Andrew File System (AFS):**

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

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

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

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

When a user process in a client computer issues an open system call for a file in the shared file space and there is not a current copy of the file in the local cache, the server holding the file is

located and is sent a request for a copy of the file. The copy is stored in the local UNIX file system in the client computer. The copy is then opened and the resulting UNIX file descriptor is returned to the client.

Naming Services:

What are Naming Services?

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

**What are Naming Services?**

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

**Key benefits:**

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

The role of names and name services

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

**Requirements for name spaces**

Allow simple but meaningful names to be used

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

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

**Name Services and the Domain Name System:**

A name service stores a collection of one or more naming contexts, sets of bindings between textual names and attributes for objects such as computers, services, and users. The major operation that a name service supports is to resolve names.

**Uniform Resource Identifiers:**

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

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

**Uniform Resource Names:**

Uniform Resource Names (URNs) are URIs that are used as pure resource names rather than locators.

For example, the URI:

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

**Navigation:**

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

Namespaces allows for structure in names.

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

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

This decomposition facilitates the resolution of the name into the corresponding resource Moreover, structured namespaces allows for iterative navigation…

**Iterative navigation:**

**Reason for NFS iterative name resolution:**

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

**Server controlled navigation:**

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

**Recursive:** it is performed by the naming server the server becomes like a client for the next server this is necessary in case of client connectivity constraints

**Non recursive:** it is performed by the client or the first server the server bounces back the next hop to its client

Non-recursive and recursive server-controlled navigation

DNS offers recursive navigation as an option, but iterative is the standard technique. Recursive navigation must be used in domains that limit client access to their DNS information for security reasons. The Domain Name System is a name service design whose main naming database is used across the Internet.

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

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

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

DNS - The Internet Domain Name System

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

Rapidly resolves domain names to IP addresses

- Exploits caching heavily
- Typical query time ~100 milliseconds
- Scales to millions of computers
- Partitioned database
- Caching

Resilient to failure of a server

- Replication

DNS server functions and configuration :

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

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

Other functions:

- get mail host for a domain
- reverse resolution - get domain name from IP address
- Host information - type of hardware and OS
- Well-known services - a list of well-known services offered by a host

Other attributes can be included (optional)

**DNS resource records:**

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

**Global Name Service (GNS):**

The GNS manages a naming database that is composed of a tree of directories holding names and values. Directories are named by multi-part pathnames referred to a root, or relative to a working directory, much like file names in a UNIX file system. Each directory is also assigned an integer, which serves as a unique directory identifier (DI). A directory contains a list of names and references. The values stored at the leaves of the directory tree are organized into value trees, so that the attributes associated with names can be structured values.

Names in the GNS have two parts: <directory name, value name>. The first part identifies a directory; the second refers to a value tree, or some portion of a value tree.

**Consensus and related problems:**

Problems of agreement

For processes to agree on a value (consensus) after one or more of the processes has proposed what that value should be

**Covered topics:** byzantine generals, interactive consistency, totally ordered multicast The byzantine generals problem: a decision whether multiple armies should attack or retreat, assuming that united action will be more successful than some attacking and some retreating

Another example might be space ship controllers deciding whether to proceed or abort. Failure handling during consensus is a key concern

Assumptions

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

Consensus Process

- Each process pi begins in an undecided state and proposes a single value vi, drawn from a set D (i=1…N).
- Processes communicate with each other, exchanging values
- Each process then sets the value of a decision variable di and enters the decided state
- Requirements for Consensus

Three requirements of a consensus algorithm

**Termination:** Eventually every correct process sets its decision variable

**Agreement:** The decision value of all correct processes is the same: if pi and pj are correct and have entered the decided state, then di=dj (i,j=1,2, …, N)

**Integrity:** If the correct processes all proposed the same value, then any correct process in the decided state has chosen that value

The byzantine generals problem

Problem description

- Three or more generals must agree to attack or to retreat
- One general, the commander, issues the order
- Other generals, the lieutenants, must decide to attack or retreat
- One or more generals may be treacherous

A treacherous general tells one general to attack and another to retreat

Difference from consensus is that a single process supplies the value to agree on

**Requirements:**

**Termination:** eventually each correct process sets its decision variable

**Agreement:** the decision variable of all correct processes is the same

**Integrity:** if the commander is correct, then all correct processes agree on the value that the commander has proposed (but the commander need not be correct)

The interactive consistency problem

**Interactive consistency:** all correct processes agree on a vector of values, one for each process. This is called the decision vector.

Another variant of consensus

Requirements

**Termination:** eventually each correct process sets its decision variable

**Agreement:** the decision vector of all correct processes is the same

**Integrity:** if any process is correct, then all correct processes decide the correct value for that process

Relating consensus to other problems

Consensus (C), Byzantine Generals (BG), and Interactive Consensus (IC) are all problems concerned with making decisions in the context of arbitrary or crash failures We can sometimes generate solutions for one problem in terms of another. For example

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

Consensus in a Synchronous System

Up to f processes may have crash failures, all failures occurring during f+1 rounds. During each round, each of the correct processes multicasts the values among themselves. The algorithm guarantees all surviving correct processes are in a position to agree.

Note: any process with f failures will require at least f+1 rounds to agree

Limits for solutions to Byzantine Generals

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

**Thrashing**

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

## UNIT- 5

## TRANSACTIONS AND CONCURRENCY

CORBA includes specifications for services that may be required by distributed objects. In particular, the Naming Service is an essential addition to any ORB. The CORBA services include the following:

**Naming Service:**

**Event Service and Notification Service: Security service:**

**Trading service:** In contrast to the Naming Service which allows CORBA objects to be located by name, the Trading Service [OMG 2000a] allows them to be located by attribute – that is, it is a directory service. Its database contains a mapping from service types and their associated attributes onto remote object references of CORBA objects. The service type is a name, and each attribute is a name-value pair. Clients make queries by specifying the type of service required, together with other arguments specifying constraints on the values of attributes, and preferences for the order in which to receive matching offers. Trading servers can form federations in which they not only use their own databases but also perform queries on behalf of one anothers' clients.

**Transaction service and concurrency control service:** The object transaction service [OMG 2003] allows distributed CORBA objects to participate in either flat or nested transactions. The client specifies a transaction as a sequence of RMI calls, which are introduced by begin and terminated by commit or rollback (abort). The ORB attaches a transaction identifier to each remote invocation and deals with begin, commit and rollback requests. Clients can also suspend and resume transactions. The transaction service carries out a two-phase commit protocol. The concurrency control service [OMG 2000b] uses locks to apply concurrency control to the access of CORBA objects. It may be used from within transactions or independently.

**Persistent state service:**An persistent objects can be implemented by storing them in a passive form in a persistent object store while they are not in use and activating them when they are needed. Although ORBs activate CORBA objects with persistent object references, getting their implementations from theimplementation repository, they are not responsible for saving and restoring the state of CORBA objects.

**Life cycle service:** The life cycle service defines conventions for creating, deleting, copying and moving CORBA objects. It specifies how clients can use factories to create objects in particular locations, allowing persistent storage to be used if required. It defines an interface that allows clients to delete CORBA objects or to move or copy them to a specified location.

**CORBA Naming Service:** The CORBA Naming Service is a sophisticated example of the binder described in Chapter 5. It allows names to be bound to the remote object references of CORBA objects within naming contexts. a naming context is the scope within which a set of names applies – each of the names within a context must be unique. A name can be associated with either an object reference for a CORBA object in an application or with another context in the naming service.

The names used by the CORBA Naming Service are two-part names, called Name Components, each of which consists of two strings, one for the name and the other for the kind of the object. The kind field provides a single attribute that is intended for use by applications and may contain any useful descriptive information; it is not interpreted by the Naming Service.Although CORBA objects are given hierarchic names by the Naming Service, these names cannot be expressed as pathnames like those of UNIX files.

**CORBA Event Service:**The CORBA Event Service specification defines interfaces allowing objects of interest, called suppliers, to communicate notifications to subscribers, called consumers. The notifications are communicated as arguments or results of ordinary synchronous CORBA remote method invocations. Notifications may be propagated either by being pushed by the supplier to the consumer or pulled by the consumer from the supplier. In the first case, the consumers implement the PushConsumer interface which includes a method push that takes any CORBA data type as argument. Consumers register their remote object references with the suppliers. The supplier invokes the push method, passing a notification as argument. In the second case, the supplier implements the PullSupplier interface, which includes a method pull that receives any CORBA data type as its return value. Suppliers register their remote object references with the consumers. The consumers invoke the pull method and receive a notification as result.

The notification itself is transmitted as an argument or result whose type is any, which means that the objects exchanging notifications must have an agreement about the contents of notifications. Application programmers, however, may define their own IDL interfaces with notifications of any desired type.

Event channels are CORBA objects that may be used to allow multiple suppliers to communicate with multiple consumers in an asynchronous manner. An event channel acts as a buffer between suppliers and consumers. It can also multicast the notifications to the consumers. Communication via an event channel may use either the push or pull style. The two styles may be mixed; for example, suppliers may push notifications to the channel and consumers may pull notifications from it.

**CORBA Notification Service**

The CORBA Notification Service extends the CORBA Event Service, retaining all of its features including event channels, event consumers and event suppliers. The event service provides no support for filtering events or for specifying delivery requirements. Without the use of filters, all the consumers attached to a channel have to receive the same notifications as one another. And without the ability to specify delivery requirements, all of the notifications sent via a channel are given the delivery guarantees built into the implementation.

The notification service adds the following new facilities:

- Notifications may be defined as data structures. This is an enhancement of the limited utility provided by notifications in the event service, whose type could only be either any or a type specified by the application programmer.
- Event consumers may use filters that specify exactly which events they are interested in. The filters may be attached to the proxies in a channel. The proxies will forward notifications to event consumers according to constraints specified in filters in terms of the contents of each notification.
- Event suppliers are provided with a means of discovering the events the consumers are interested in. This allows them to generate only those events that are required by the consumers.
- Event consumers can discover the event types offered by the suppliers on a channel, which enables them to subscribe to new events as they become available.
- It is possible to configure the properties of a channel, a proxy or a particular event. These properties include the reliability of event delivery, the priority of events, the ordering required (for example, FIFO or by priority) and the policy for discarding stored events.
- An event type repository is an optional extra. It will provide access to the structure of events, making it convenient to define filtering constraints.

A structured event consists of an event header and an event body. The following Example illustrates the contents of the header.The following example illustrates the information in the body of a structured event:

Filter objects are used by proxies in making decisions as to whether to forward each notification. A filter is designed as a collection of constraints, each of which is a data structure with two components: A list of data structures, each of which indicates an event type in terms of its domain name and event type, for example, "home", "burglar alarm". The list includes all of the event types to which the constraint should apply. A string containing a boolean expression involving the values of the event types listed above. For example:

("domain type" == "home" && "event type" == "burglar alarm") && ("bell" != "ringing" !! "door" == "open")

**CORBA Security Service**

The CORBA Security Service [Blakley 1999, Baker 1997, OMG 2002b] includes the following: Authentication of principles (users and servers); generating credentials for principals (that is, certificates stating their rights); delegation of credentials is supported Access control can be applied to CORBA objects when they receive remote method invocations. Access rights may for example be specified in access control lists (ACLs). Security of communication between clients and objects, protecting messages for integrity and confidentiality.Auditing by servers of remote method invocations.Facilities for non-repudiation. When an object carries out a remote invocation on behalf of a principal, the server creates and stores credentials that prove that the invocation was done by that server on behalf of the requesting principal. CORBA allows a variety of security policies to be specified according to requirements. A message-protection policy states whether client or server (or both) must be authenticated, and whether messages must be protected against disclosure and/or modification. Access control takes into account that many applications have large numbers of users and even larger numbers of objects, each with its own set of methods. Users are supplied with a special type of credential called a privilege according to their roles.

**RELEASE CONSISTENCY AND MUNIN CASE STUDY**

Release consistency was introduced with the Dash multiprocessor, which implements DSM in hardware, primarily using a write-invalidation protocol [Lenoskietal. 1992]. Munin and Treadmarks [Keleher et al. 1992] have adopted a software implementation of it. Release

consistency is weaker than sequential consistency and cheaper to implement, but it has reasonable semantics that are tractable to programmers.

The idea of release consistency is to reduce DSM overheads by exploiting the fact that programmers use synchronization objects such as semaphores, locks and barriers. A DSM implementation can use knowledge of accesses to these objects to allow memory to become inconsistent at certain points, while the use of synchronization objects nonetheless preserves application-level consistency.

**Memory accesses:**In order to understand release consistency – or any other memory model that takes synchronization into account – we begin by categorizing memory accesses according to their role, if any, in synchronization. Furthermore, we shall discuss how memory accesses may be performed asynchronously to gain performance and give a simple operational model of how memory accesses take effect. DSM implementations on general-purpose distributed systems may use message passing rather than shared variables to implement synchronization.

**Types of memory access:** The main distinction is between competing accesses and noncompeting (ordinary) accesses. Two accesses are competing if: they may occur concurrently (there is no enforced ordering between them) and at least one is a write.

So two read operations can never be competing; a read and a write to the same location made by two processes that synchronize between the operations (and so order them) are non-competing. We further divide competing accesses into synchronization and no synchronization accesses. Synchronization accesses are read or write operations that contribute to synchronization; non-synchronization accesses are read or write operations that are concurrent but that do not contribute to synchronization.

Performing asynchronous operations:In view of the asynchronous operation that we have outlined, we distinguish between the point at which a read or write operation is issued – when the process first commences execution of the operation – and the point when the instruction is performed or completed.

We shall assume that our DSM is at least coherent. It means that every process agrees on the order of write operations to the same location. Given this assumption, we may speak unambiguously of the ordering of write operations to a given location.

**Release consistency:** The requirements that we wish to meet are:

- to preserve the synchronization semantics of objects such as locks and barriers;
- to gain performance, we allow a degree of a synchronicity for memory operations;
- to constrain the overlap between memory accesses in order to guarantee executions that provide the equivalent of sequential consistency.

**Munin:** The Munin DSM design [Carter et al. 1991] attempts to improve the efficiency of DSM by implementing the release consistency model. Furthermore, Munin allows programmers to annotate their data items according to the way in which they are shared, so that optimizations can be made in the update options selected for maintaining consistency. It is implemented upon the V

Munin sends update or invalidation information as soon as a lock is released.

The programmer can make annotations that associate a lock with particular data items. In this case, the DSM runtime can propagate relevant updates in the same message that transfers the lock to a waiting process – ensuring that the lock's recipient has copies of the data it needs before it accesses them.

**Sharing annotations:** Munin implements a variety of consistency protocols, which are applied at the granularity of individual data items. The protocols are parameterized according to the following options:

- Whether to use a write-update or write-invalidate protocol;
- Whether several replicas of a modifiable data item may exist simultaneously;
- Whether or not to delay updates or invalidations (for example, under release consistency);
- Whether the item has a fixed owner, to which all updates must be sent;
- Whether the same data item may be modified concurrently by several writers;
- Whether the data item is shared by a fixed set of processes;
- Whether the data item may be modified.

**Read-only:** No updates may be made after initialization and the item may be freely copied.

**Migratory:** Processes typically take turns in making several accesses to the item, at least one of which is an update. For example, the item might be accessed within a critical section. Munin always gives both read and write access together to such an object, even when a process takes a read fault. This saves subsequent write-fault processing.

**Write-shared:** Several processes update the same data item (for example, an array) concurrently, but this annotation is a declaration from the programmer that the processes do not update the same parts of it. This means that Munin can avoid false sharing but must propagate only those words in the data item that are actually updated at each process. To do this, Munin makes a copy of a page (inside a write-fault handler) just before it is updated locally. Only the differences between the two versions are sent in an update.

**Producer-consumer:** The data object is shared by a fixed set of processes, only one of which updates it. As we explained when discussing thrashing above, a writeupdate protocol is most suitable here. Moreover, updates may be delayed under the model of release consistency, assuming that the processes use locks to synchronize their accesses.

Reduction: The data item is always modified by being locked, read, updated and unlocked. An example of this is a global minimum in a parallel computation, which must be fetched and modified atomically if it is greater than the local minimum. These items are stored at a fixed owner. Updates are sent to the owner, which propagates them.

**Result:** Several processes update different words within the data item; a single process reads the whole item. For example, different 'worker' processes might fill in different elements of an array, which is then processed by a 'master' process. The point here is that the updates need only be propagated to the master and not to the workers (as would occur under the 'write-shared' annotation just described).

**Conventional:** The data item is managed under an invalidation protocol similar to that described in the previous section. No process may therefore read a stale version of the data item.

**Interface repository:** The role of the interface repository is to provide information about registered IDL interfaces to clients and servers that require it. For an interface of a given type it can supply the names of the methods and for each method, the names and types of the arguments and exceptions. Thus, the interface repository adds a facility for reflection to CORBA.

**Dynamic invocation interface**: The dynamic invocation interface allows clients to make dynamic invocations on remote CORBA objects. It is used when it is not practical to employ proxies. The client can obtain from the interface repository the necessary information about the methods available for a given CORBA object. The client may use this information to construct an invocation with suitable arguments and send it to the server.

**Dynamic skeletons:**If a server uses dynamic skeletons, then it can accept invocations on the interface of a CORBA object for which it has no skeleton. When a dynamic skeleton receives an invocation, it inspects the contents of the request to discover its target object, the method to be invoked and the arguments. It then invokes the target.

**Legacy code:** The term legacy code refers to existing code that was not designed with distributed objects in mind. A piece of legacy code may be made into a CORBA object by defining an IDL interface for it and providing an implementation of an appropriate object adapter and the necessary skeletons.

**CORBA Interface Definition Language**

The CORBA Interface Definition Language, IDL, provides facilities for defining modules, interfaces, types, attributes and method signatures. IDL has the same lexical rules as C++ but has additional keywords to support distribution, for example interface, any, attribute, in, out, inout, readonly, raises. It also allows standard C++ preprocessing facilities.

**IDL Modules**

The module construct allows interfaces and other IDL type definitions to be grouped in logical units. A module defines a naming scope, which prevents names defined within a module clashing with names defined outside it.

**IDL interface**

An IDL interface describes the methods that are available in CORBA objects that implement that interface. Clients of a CORBA object may be developed just from the knowledge of its IDL interface.

**IDL methods**

The general form of a method signature is:

[oneway] <return_type><method_name> (parameter1,...,parameterL) [raises (except1,..., exceptN)] [context (name1,..., nameM)]

where the expressions in square brackets are optional. For an example of a method signature that contains only the required parts, consider:

voidgetPerson(in string name, out Person p);

**IDL types:**

IDL supports fifteen primitive types, which include short (16-bit), long (32- bit), unsigned short, unsigned long, float (32-bit), double (64-bit), char, Boolean (TRUE, FALSE), octet (8-bit), and any (which can represent any primitive or constructed type).

**Attributes:**

IDL interfaces can have attributes as well as methods. Attributes are like public class fields in Java. Attributes may be defined as readonly where appropriate. The attributes are private to CORBA objects, but for each attribute declared, a pair of accessor methods is generated automatically by the IDL compiler, one to retrieve the value of the attribute and the other to set it. For readonly attributes, only the getter method is provided. For example, the PersonList interface defined in Figure 5.2 includes the following definition of an attribute: readonly attribute string listname;

**Inheritance:**

IDL interfaces may be extended. For example, if interface B extends interface A, this means that it may add new types, constants, exceptions, methods and attributes to those of A. An extended interface can redefine types, constants and exceptions, but is not allowed to redefine methods. A value of an extended type is valid as the value of a parameter or result of the parent type. For example, the type B is valid as the value of a parameter or result of the type A.

interface A { }; interface B: A{ }; interface C {}; interface Z : B, C{};