UNIT: 4
DISTRIBUTED COMPUTING

Introduction To Distributed Programming:

- Distributed computing is a model in which components of a software system are shared among multiple computers. Even though the components are spread out across multiple computers, they are run as one system. This is done in order to improve efficiency and performance.

- Distributed computing allows different users or computers to share information. Distributed computing can allow an application on one machine to leverage processing power, memory, or storage on another machine. Some applications, such as word processing, might not benefit from distribution at all.

- In parallel computing, all processors may have access to a shared memory to exchange information between processors. In distributed computing, each processor has its own private memory (distributed memory). Information is exchanged by passing messages between the processors.

- Distributed computing systems are omnipresent in today's world. The rapid progress in the semiconductor and networking infrastructures have blurred the differentiation between parallel and distributed computing systems and made distributed computing a workable alternative to high-performance parallel architectures.

- However attractive distributed computing may be, developing software for such systems is hardly a trivial task. Many different models and technologies have been proposed by academia and industry for developing robust distributed software systems.

- Despite a large number of such systems, one fact is clear that the software for distributed computing can be written efficiently using the principles of distributed-object computing.

- The concept of objects residing in different address spaces and communicating via messages over a network meshes well with the principles of distributed computation. Out of the existing alternatives, Java-RMI (Remote Method Invocation) from Sun Micro systems, CORBA (Common Object Request Broker Architecture) from Object Management Group, and DCOM (Distributed Component Object Model) from Microsoft are the most popular distributed-object models among researchers and practitioners.

System models

Architectural model

An **architectural model** of a **distributed system** defines the way in which the components of the **system** interact with each other and the way in which they are mapped onto an underlying network of computers. E.gs. include the client-server **model** and the peer process **model**.

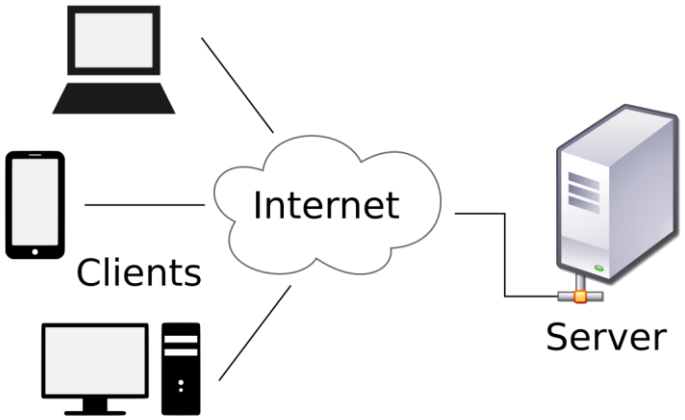
Client server model

Client–server model is a distributed application structure that partitions tasks or workloads between the providers of a resource or service, called servers, and service requesters, called clients.[1] Often clients and servers communicate over a computer network on separate hardware, but both client and server may reside in the same system. A server host runs one or more server programs, which share their resources with clients.

A client does not share any of its resources, but it requests content or service from a server. Clients, therefore, initiate communication sessions with servers, which await incoming requests. Examples of computer applications that use the client-server model are email, network printing, and the World Wide Web.

The client-server characteristic describes the relationship of cooperating programs in an application. The server component provides a function or service to one or many clients, which initiate requests for such services. Servers are classified by the services they provide. For example, a web server serves web pages and a file server serves computer files. A shared resource may be any of the server computer's software and electronic components, from programs and data to processors and storage devices. The sharing of resources of a server constitutes a service

Whether a computer is a client, a server, or both, is determined by the nature of the application that requires the service functions. For example, a single computer can run a web servers and file server software at the same time to serve different data to clients making different kinds of requests. Client software can also communicate with server software within the same computer.[2] Communication between servers, such as to synchronize data, is sometimes called inter-server or server-to-server communication.

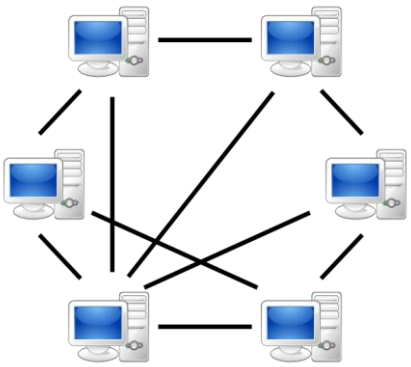


Peer to peer model

Peer-to-peer (**P2P**) computing or networking is a distributed application architecture that partitions tasks or workloads between peers. Peers are equally privileged, equipotent participants in the application. They are said to form a peer-to-peer network of nodes.

Peers make a portion of their resources, such as processing power, disk storage or network bandwidth, directly available to other network participants, without the need for central coordination by servers or stable hosts.[1] Peers are both suppliers and consumers of resources, in contrast to the traditional client-server model in which the consumption and supply of resources is divided. Emerging collaborative P2P systems are going beyond the era of peers doing similar things while sharing resources, and are looking for diverse peers that can bring in unique resources and capabilities to a virtual community thereby empowering it to engage in greater tasks beyond those that can be accomplished by individual peers, yet that are beneficial to all the peers.[2]

While P2P systems had previously been used in many application domains,[3] the architecture was popularized by the file sharing system Napster, originally released in 1999. The concept has inspired new structures and philosophies in many areas of human interaction. In such social contexts, peer-to-peer as a meme refers to the egalitarian social networking that has emerged throughout society, enabled by Internet technologies in general.

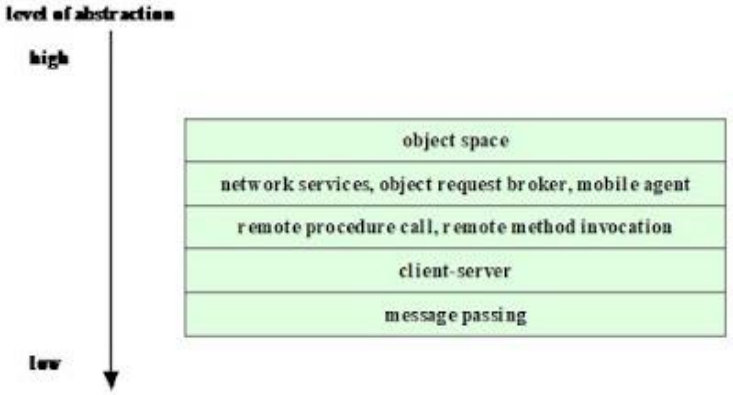


Distributed algorithm paradigms

What is Paradigm?

Paradigm means **"A Pattern, Example or Model."**

The paradigms will be presented in the order of their level of abstraction.



At the lowest level of abstraction is **Message passing** which encapsulates the least amount of detail and the **Object Space** occupies the highest level of abstraction among all the paradigms.

1.MESSAGE PASSING PARADIGM:

The basic approach to interprocess communication is Message Passing.

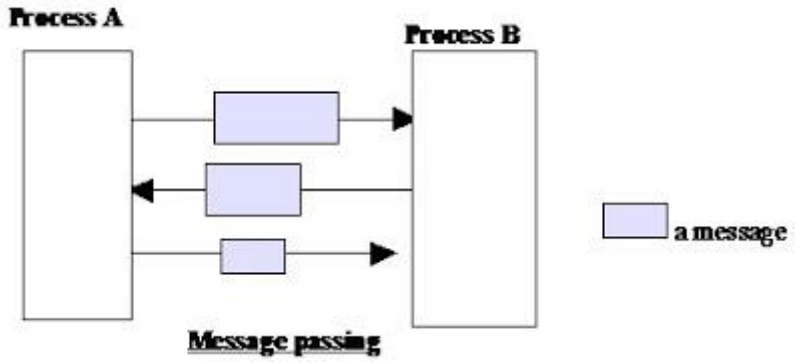
The data representing messages are exchanged between two processes, a sender and receiver.

A Process sends a message representing a request. The message is delivered to a receiver, which processes the request and sends a message in response.

The basic operations required to support the message passing paradigm are SEND and RECEIVE.

The abstraction provided by this model, the interconnected processes perform input and output to each other, in a manner similar to file input and output.

The Socket application programming interface is based on this paradigm.



2.CLIENT-SERVER PARADIGM:

The Client-Server model assigns assymmetric roles to two collaborating processes.

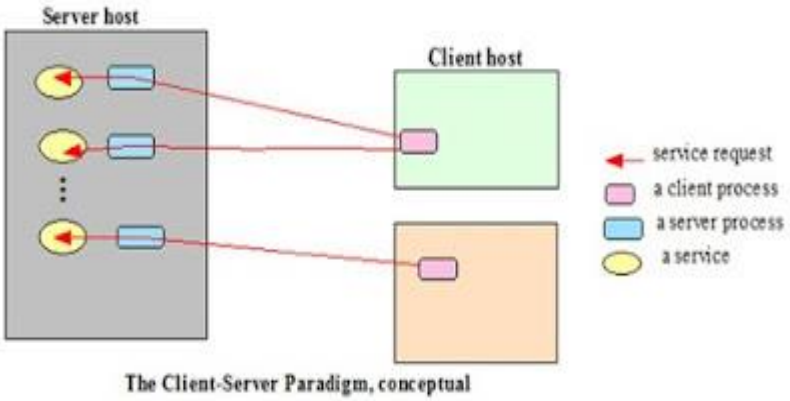
One process, the server, plays the role of a service provider which waits passively for the arrival of requests.

The other process, the client, issues specific requests to the server and awaits the server's response.

The Client-Server model provides an efficient abstraction for the delivery of network services.

Operations required include those for a server process to listen and accept requests, and for a client process to issue requests and accepts responses.

Example: HTTP, FTP, DNS, Finger, Gopher, etc.,



The Client-Server Paradigm, conceptual

3.PEER-TO-PEER PARADIGM:

The Client-Server paradigm has the role for server fixed to be a listener, it cannot initiate any action.

The Peer-to-peer paradigm, the participating processes play equal roles, with equivalent capabilities and responsibilities.

Each participant may issue a request to another participant and receive a response.

The Peer-to-peer paradigm is more appropriate for applications such as instant messaging, peer-to-peer file transfers, video conferencing and collaborative work.

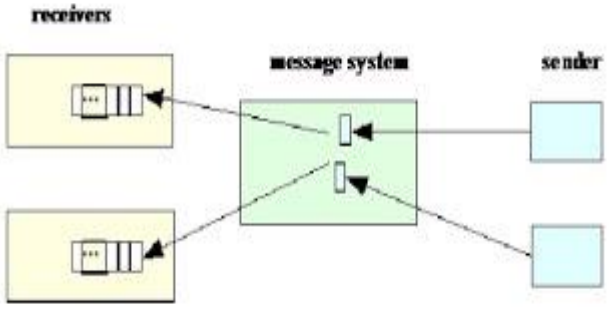
4.MESSAGE SYSTEM PARADIGM (MOM):

The Message System or Message Oriented Middleware (MOM) paradigm is an elaboration of the basic message passing paradigm.

In this paradigm, a message system serves as an intermediary among separate, independent processes.

A sender deposits a message with the message system, which forwards it to a message queue associated with each receiver.

Once a message sent, the sender is free to move on to other tasks.



SYSTEM PARADIGM:

The message system acts as a switch for messages through which processes exchange messages asynchronously, in a decoupled manner.

There are two main subtypes of message system models,

- The Point-To-Point Message Model
- The Publish/Subscribe Message Model

a)The Point-To-Point Message Model:

In this paradigm, a message system forwards a message from the sender to the receiver's message queue.

A receiving process extracts the messages from its message queue, and handles each one accordingly.

Compared to the basic message-passing model, this paradigm provides the additional abstraction for asynchronous operations.

b)The Publish/Subscribe Message Model:

In this model, each message is associated with a specific topic or event.

Applications interested in the occurrence of a specific event may subscribe to messages for that event.

This model offers a powerful abstraction for multicasting or group communication.

The Publish operation allows a process to multicast to a group of processes, and the Subscribe operation allows a process to listen for such multicast.

5.REMOTE PROCEDURE CALL (RPC):

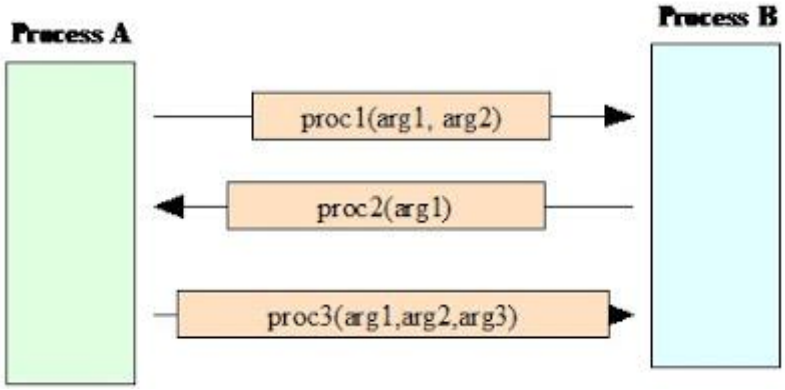
The Remote Procedure Call model provides such an abstraction. In this model, interprocess communications proceed as procedure, or function, calls which are familiar to application programmers.

A remote procedure call involves two independent processes, which may reside on separate machines.

A process A, wishing to make a request to another process B, issues a procedure call to B, passing with the call a list of argument values.

A remote procedure call triggers a predefined action in a procedure provided by process B.

At the completion of the procedure, process B returns a value to process A.



6.DISTRIBUTED OBJECTS PARADIGMS:

The idea of applying object oriented to distributed applications is a natural extension of object-oriented software development. Applications access objects distributed over a network.

Objects provide methods, through the invocation of which an application obtains access to services.

Object-oriented paradigms include:

- Remote Method Invocation(RMI)
- The Object Request Broker Paradigm(ORB)
- The Object Space Paradigm
- The Mobile Agent Paradigm
- The Network Services Paradigm
- The Groupware Paradigm

a)Remote Method Invocation(RMI):

Remote Method Invocation is the object-oriented equivalent of remote method calls. In this model, a process invokes the methods in an object, which may reside in a remote host.

b)The Object Request Broker Paradigm(ORB):

In the object broker paradigm, an application issues requests to an object request broker(ORB), which directs the request to an appropriate object that provides the desired service.

c)The Object Space Paradigm:

The Object Space paradigm assumes the existence of logical entities known as Object Spaces. The participants of an application converge in a common object space.

d)The Mobile Agent Paradigm:

A Mobile Agent is a transportable Program or Object. In this model, an agent is launched from an originating host.

e)The Network Services Paradigm:

In this paradigm, service providers register themselves with directory servers on a network. A process desiring a particular service contacts the directory server at run time, and if the service is available, will be provided a reference to the service.

f)The Collaborative Application (Groupware) Paradigm:

In this model, processes participate in a collaborative session as a group. Each participating process may contribute input to part or the entire group.

Inter Process Communication

Interprocess communication is the mechanism provided by the operating system that allows processes to communicate with each other.

This communication could involve a process letting another process know that some event has occurred or the transferring of data from one process to another.

IPC is very important to the design process for microkernels and nanokernels, which reduce the number of functionalities provided by the kernel.

Those functionalities are then obtained by communicating with servers via IPC, leading to a large increase in communication when compared to a regular monolithic kernel.

IPC interfaces generally encompass variable analytic framework structures. These processes ensure compatibility between the multi-vector protocols upon which IPC models rely.

Approaches of Interprocess Communication:

The different approaches to implement interprocess communication are given as follows

Pipe

A pipe is a data channel that is unidirectional. Two pipes can be used to create a two-way data channel between two processes. This uses standard input and output methods. Pipes are used in all POSIX systems as well as Windows operating systems.

Socket

The socket is the endpoint for sending or receiving data in a network. This is true for data sent between processes on the same computer or data sent between different computers on the same network. Most of the operating systems use sockets for interprocess communication.

File

A file is a data record that may be stored on a disk or acquired on demand by a file server. Multiple processes can access a file as required. All operating systems use files for data storage.

Signal

Signals are useful in interprocess communication in a limited way. They are system messages that are sent from one process to another. Normally, signals are not used to transfer data but are used for remote commands between processes.

Shared Memory

Shared memory is the memory that can be simultaneously accessed by multiple processes. This is done so that the processes can communicate with each other. All POSIX systems, as well as Windows operating systems use shared memory.

Message Queue

Multiple processes can read and write data to the message queue without being connected to each other. Messages are stored in the queue until their recipient retrieves them. Message queues are quite useful for interprocess communication and are used by most operating systems.

Synchronization in Interprocess Communication:

Synchronization is a necessary part of interprocess communication.

It is either provided by the interprocess control mechanism or handled by the communicating processes. Some of the methods to provide synchronization are as follows −

Semaphore

A semaphore is a variable that controls the access to a common resource by multiple processes. The two types of semaphores are binary semaphores and counting semaphores.

Mutual Exclusion

Mutual exclusion requires that only one process thread can enter the critical section at a time. This is useful for synchronization and also prevents race conditions.

Barrier

A barrier does not allow individual processes to proceed until all the processes reach it. Many parallel languages and collective routines impose barriers.

Spinlock

This is a type of lock. The processes trying to acquire this lock wait in a loop while checking if the lock is available or not. This is known as busy waiting because the process is not doing any useful operation even though it is active.

Message Passing Model of Process Communication

Process communication is the mechanism provided by the operating system that allows processes to communicate with each other. This communication could involve a process letting another process know that some event has occurred or transferring of data from one process to another. One of the models of process communication is the message passing model.

Message passing model allows multiple processes to read and write data to the message queue without being connected to each other. Messages are stored on the queue until their recipient retrieves them. Message queues are quite useful for interprocess communication and are used by most operating systems.

Advantage of Message Passing Model

The message passing model is much easier to implement than the shared memory model.

It is easier to build parallel hardware using message passing model as it is quite tolerant of higher communication latencies.

Disadvantage of Message Passing Model

The message passing model has slower communication than the shared memory model because the connection setup takes time.

Terms Used in IPC

The following are a few important terms used in IPC:

Semaphores:A semaphore is a signaling mechanism technique. This OS method either allows or disallows access to the resource, which depends on how it is set up.

Signals: It is a method to communicate between multiple processes by way of signaling. The source process will send a signal which is recognized by number, and the destination process will handle it.

API FOR INTERNET PROTOCOLS

API Stands for "Application Programming Interface." An **API** is a set of commands, functions, **protocols**, and objects that programmers can use to create software or interact with an external system. .

It also provides commands for accessing the file system and performing file operations, such as creating and deleting files.

The characteristics of inter processes communication: Message passing between a pair of processes can be supported by two message communication operations, send and receive, defined in terms of destinations and messages

To communicate, one process sends a message (a sequence of bytes) to a destination and another process at the destination receives the message. This activity involves the communication of data from the sending process to the receiving process and may involve the synchronization of the two processes.

Synchronous and asynchronous communication

A queue is associated with each message destination. Sending processes cause messages to be added to remote queues and receiving processes remove messages from local queues.

Communication between the sending and receiving processes may be either synchronous or asynchronous.

synchronousform of communication,

The sending and receiving processes synchronize at very 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 receives issued by a process (or thread), it blocks until a message arrives.

Asynchronousform of communication,

The use of the send operation is non-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.

MESSAGE DESTINATION

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

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

Reliability

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

A point-to-point message service is described as reliable if messages are guaranteed to be delivered despite

a reasonable number of packets being dropped or lost.
For integrity, messages must arrive uncorrupted and without duplication.

Sockets

Both forms of communication (UDP and TCP) use the socket abstraction, which provides an endpoint for communication between processes.
Sockets originatefrom BSD UNIX but are also present in most other versions of UNIX, including Linux as well as Windows and the Macintosh OS.

UDP DATA GRAM COMMUNICATION

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 DISTRIBUTED SYSTEMS UNIT II 3 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 upto216 bytes, which includes the headers as well as the message .However, most environments impose a size restriction of 8 kilobytes

Blocking

Sockets normally provide non-blocking sends and blocking receives for datagram communication
The send operation returns when it has handed the message to the underlying UDP and IP protocols ,which are responsible for transmitting it to its destination .

Timeouts

The receive that blocks for ever is suitable for use by a server that is waiting to receiver requests from its clients. But in some programs, it is not appropriate that a process that has invoked a receive operation should wait indefined.
To allow for such requirements, timeouts can be set on sockets. Choosing an appropriate timeout interval is difficult, but it should be fairlylarge incomparison with the time required to transmit a message.

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.

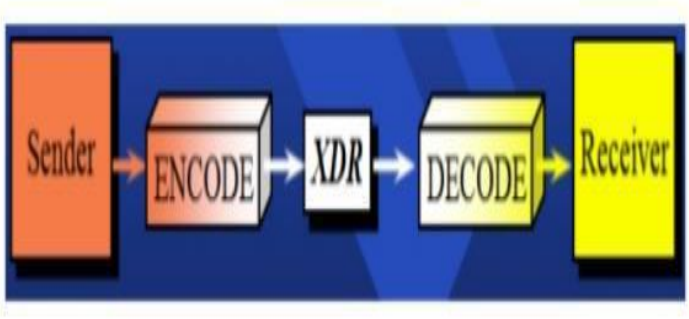
Ordering

: 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.

EXTERNAL DATA REPRESENTATION AND MARSHALLING

External Data Representation (**XDR**) is a standard **data** serialization format, It allows data to be transferred between different kinds of computer systems. In a distributed system information in messages transferring between components consists of sequences of bytes.

To communicate any information these data structures must be converted to a sequence of bytes before transmission. Converting from the local representation to XDR is called ***encoding***. Converting from XDR to the local representation is called ***decoding***.



- At language level data are stored in data structures .
- At TCP/UDP-level data are communicated as ‘messages’ or streams of bytes .
- Different machines have different primitive data types.That are integer,float,char.

External data representation is the data type that act as the intermediate data type in the transmission. XDR, an agreed standard for the representation of data structures and primitive values .E.g: CORBA Common Data Rep (CDR) for many languages, Java object serialization for Java code only.

Marshalling

Marshalling is the process of taking a collection of the data structures to transfer and format them into an external data representation type which suitable for transmission in a message.

Unmarshalling

Unmarshalling is the inverse of this process, which is reformatting the transferred data on arrival to produce the original data structures at the destination.

Marshalling consists of the translation of structured data items and primitive values into an external data representation, unmarshalling consists of the generation of primitive values from their external data representation and the rebuilding of the data structures.

There are some approaches like CORBA’s (Common Object Request Broker Architecture) common data representation, Java’s object serialization and XML (Extensible Markup Language) to external data representation and marshalling.

CORBA’s Common Data Representation

CORBA’s common data representation is the external data representation, which can represent all of the data types that can be used as arguments and return values in remote invocations.And it is represented by sequence of bytes.

For primitive types, CDR (Common Data Representation) defines representation for both big-endian and little-endian ordering. Values are transmitted in sender’s ordering, which is specified in each message.

For constructed types, the primitive values that comprise each constructed type are added to a sequence of bytes in a particular order. The representation for constructed data types are

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

Marshalling in CORBA can be generated automatically, form the specification of the types of data items to be transmitted in message. Types of the data structures and the types of the basic data items are described in CORBA IDL(Interface Definition Language), which provides a notation for describing the types of the arguments and results . The CORBA interface compiler generates appropriate marshalling and unmarshalling operations.

The flattened form represents a Person struct with value:

{‘Smith’,’London’,1934}

index in sequence of bytes	← 4 bytes →
0–3	5
4–7	"Smit"
8–11	"h "
12–15	6
16–19	"Lond"
20-23	"on "
24–27	1934

Java’s Object Serialization

java’s object serialization is concerned with the flattening and external data representation of any single object or tree of objects that may need to be transmitted in a message or stored in a disk.

Serialization refers to the activity of a connected set of objects into a serial form that issuitable for storing on disk or transmitting in a message. Deserialization consists of restoring the state of an object or set of objects from their serialized form. In java RMI, both objects and primitive data values may be passed as arguments and results of method invocations.

To serialize an object: ‹

Its class info is written out: name, version number ‹Types and names of instance variables . If an instance variable belong to a new class, then new

class info must be written out, recursively. Each class is given a handle ‹

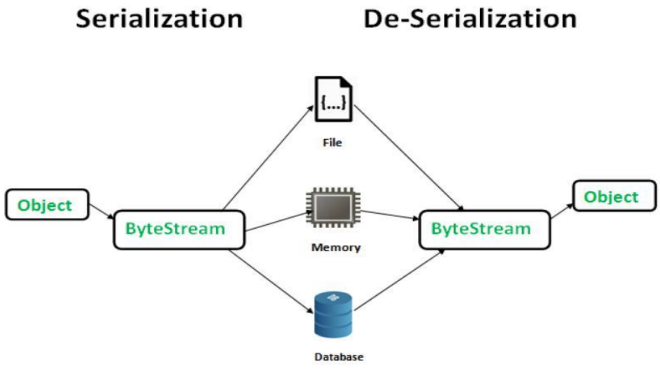
values of instance variables.

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

Serialized values

Person	8-byte version number		h0
3	int year	java.lang.String name	java.lang.String place
1934	5 Smith	6 London	h1

When an object is serialized, all the other objects that it references are serialized together with it to ensure that when the object is reconstructed, all of its references can be full filled at the destination.And these references are serialized as handles.



Serialization and deserialization in java

Serialization and deserialization of the arguments and results of remote invocations are generally carried out automatically by the middle ware. Java supports reflection, the ability to inquire about the properties of a class. Reflection makes it possible to do serialization and deserialization in a completely generic manner. This means that there is no need to generate special marshalling functions for each types of objects, as described for CORBA.

XML (Extensible Markup Language)

XML is a markup language for general use on the web. The term markup language refers to a textual encoding that represents both text and details as to its structure or its appearance.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.

Marshalling and unmarshalling in first two cases are intended to be done by a middle ware layer without any interfere on the part of the application programmer. XML is textual and therefore more accessible to hand encoding, marshalling and unmarshalling software is available for all commonly used platforms and programming environments.

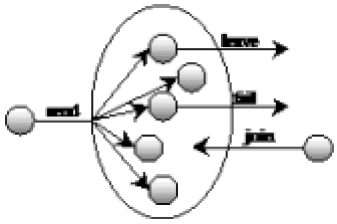
The primitive data types are marshalled into binary form in the first two approaches. But in XML primitive data types are represented textually. Textual representation of data value will generally be longer than the same binary representation. The external data representation although used for the arguments and results of RMIs and RPCs, it has a more general use for representing data structures, objects or structured documents in a form suitable for transmission in messages or storing in files.This is how the XDR and marshalling are performed in different formats.

Group Communication

Introduction

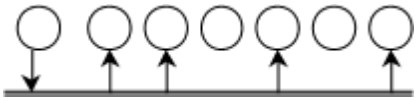
Bemote procedure calls assume the existence of two parties: a client and a server. This, as well as the socket - based communication. Sometimes, we want one-to-many, or group, communication.

They may be created and destroyed. Processes may join or leave groups and processes may belong to multipie groups.
An analogy to group communication is the concept of a mailing list.



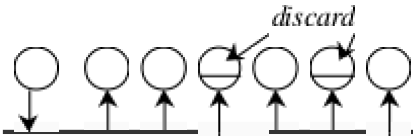
Group Dynamics

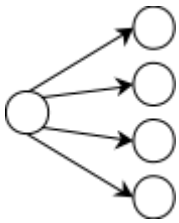
A sender sends a message to one party (the mailing list) and multiple users (members of the list) receive the message. Grays allow processes deal with collections of processes as one abstraction.

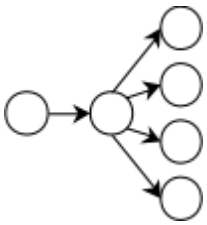


Communication Group Implementing:

Group communication can be imple mented in seve raI ways. Messages sent to the multicast address will be received by all network cards listening on that group (s).







A final implementation option is to simulate multicasting completely in software. In this case, a separate message will be sent to each receiver. This can be implemented in two ways. The sending machine can know all the members of the group and send the same message to each of group member.

Design issues

A number of design alterrnatives for group communication are available. These will affect how the group behave and send messages.

INTER PROCESS COMMUNICATION IN UNIX

Most UNIX systems limit pipes to 5120K (typically ten 512K chunks). The unbuffered system call write() is used to add data to a pipe. Write() takes a file descriptor (which can refer to the pipe), a buffer containing the data to be written, and the size of the buffer as parameters. The system assures that no interleaving will occur between writes, even if the pipeline fills temporarily. To get data from a pipe, the read() system call is used. Read() functions on pipes much the same as it functions on files. However, seeking is not supported and it will block until there is data to be read.

The pipe() system call is used to create unnamed pipes in UNIX. This call returns two pipes. Both support bidirectional communication (two pipes are returned for historical reasons: at one time pipes were unidirectional so two pipes were needed for bidirectional communication). In a fullduplex environment (i.e. one that supports bidirectional pipes) each process reads from one pipe and writes to the other; in a half-duplex (i.e. unidirectional) setting, the first file descriptor is always used for reading and the second for writing.

Pipes are commonly used on the UNIX command line to send the output of one process to another process as input. When a pipe is used both processes run concurrently and there is no guarantee as to the sequence in which each process will be allowed to run. However, since the system manages the producer/consumer issue, both proceed per usual, and the system provides automatic blocking as required

Using unnamed pipes in a UNIX environment normally involves several steps:

Create the pipe(s) needed
Generate the child process(es)
Close/duplicate the file descriptors to associate the ends of the pipe
Close the unused end(s) of the pipe(s)
Perform the communication
Close the remaining file descriptors
Wait for the child process to terminate

To simplify this process, UNIX provides two system calls that handle this procedure. The call popen() returns a pointer to a file after accepting a shell command to be executed as input. Also given as input is a type flag that determines how the returned file descriptor will be used. The popen() call automatically generates a child process, which exec()s a shell and runs the indicated command. Depending on the flag passed in, this command could have either read or write access to the file. The pclose() call is used to close the data stream opened with popen(). It takes the file descriptor returned by popen() as its only parameter.

Named pipes can be created on the UNIX command line using mknod, but it is more interesting to look at how they can be used programatically. The mknod() system call, usable only by the superuser, takes a path, access permissions, and a device (typically unused) as parameters and creates a pipe referred to by the user-specified path. Often, mkfifo() will be provided as an additional call that can be used by all users but is only capable of making FIFO pipes.

Cooperating process require IPC to exchange data and information. Two models:

1.**Shared Memory**: Read and write data in shared region Maximum speed & convenience, within computer e.g. UNIX pipes

Message Passing: Exchange messages between cooperating process Useful for exchanging small

amount of data Implementation ease for intercomputer communication Vs SM e.g. UNIX Sockets

UNIX signals
A UNIX signal is a form of IPC used to notify a process of an event.
generated when event first occurs
delivered when the process takes an action on that signal
pending when generated but not yet delivered. Signals, also called software interrupts, generally occur asynchronously

Signals
Various notifications sent to a process to notify it of important event
They interrupt whatever the process is doing at that time „
Unique integer number and symbolic name (/usr/include/signal.h) „
See the list of signals supported in your system
Each signal may have a signal handler, function that gets called when process receives the signal

Handling Signals „
Used by OS to notify the processes that some event has occurred „
Event notification mechanism for a specific application

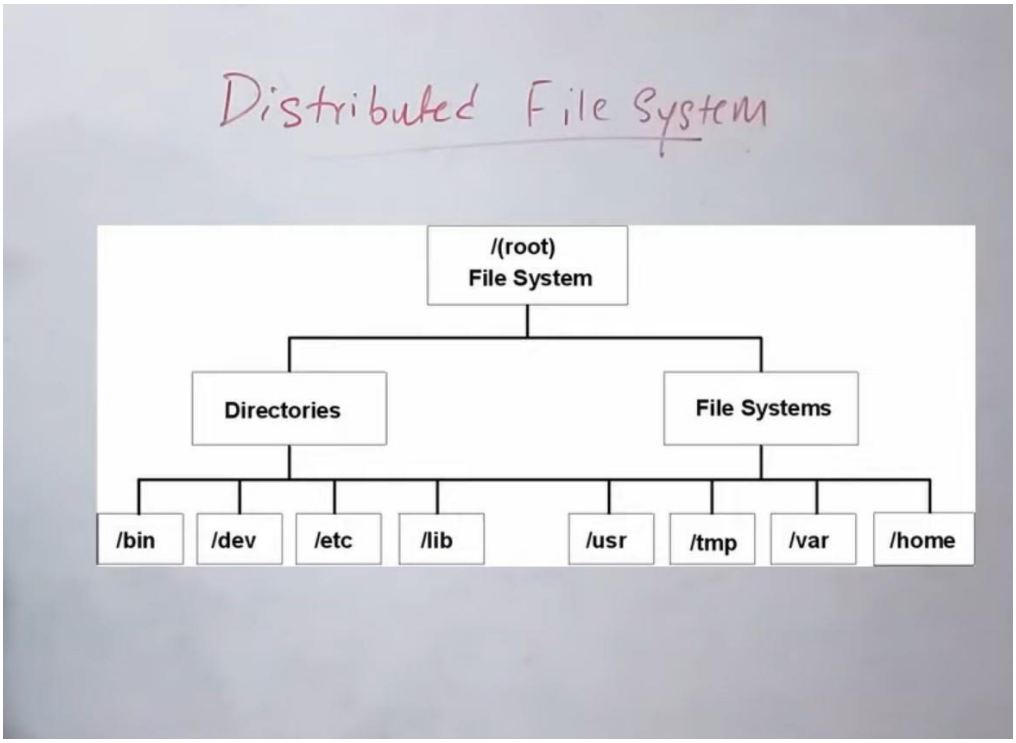
Sending Signals „
One process to another, including itself „ Kernel (OS) to process UNIX signals

Sending Signals Using Keyboard „
Ctrl-C Š-System sends an INT signal (SIGINT) to running process Š
By default – Immediately terminates the running process „
Ctrl-Z -System sends an TSTP signal (SIGSTP) to running process Š
By default – Suspends the execution of running process

Sending Signals Using Command Line
kill <signal><PID>
Signal name or number, e.g. kill – INT 1560, similar to Ctrl-C Š If no Signal name?
fg
Resume the execution of process suspended by Ctrl-Z by sending CONT signal
Raise <signal>
Process sends signal to itself
Signal <signal, SIGARG func>
System Call, A process may declare a function to serve a particular signal as above. When signal is received, Š
Process is interrupted and func is called immediately, resumes once executed
Using the signal() system call, a process can:
Ignore the signal – only two signals, SIGKILL (kill-9 PID) and SIGSTOP (Ctrl-Z) cannot be ignored
Catch the signal – tell the kernel to call a function whenever the signal occurs
Let the default action apply – depending upon the signal, the default action can be:
exit – perform all activities as if the exit system call is requested
core – first produce core image on the disk and then perform the exit activities
stop – suspend the process
ignore – disregard the signal.

DISTRIBUTED FILE SYSTEM

File system:File system which is used to handle our files and folders where ever stored on our disk drive.



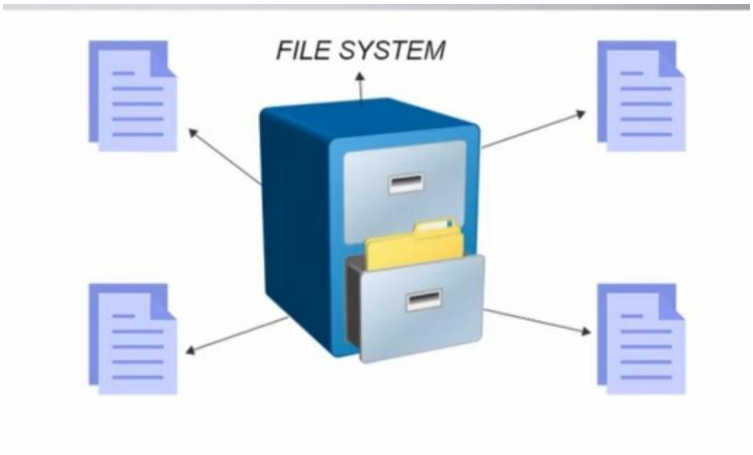
File system uses the lot of compression technique to save files into disk.

Ex: FAT, NTFS, EXFAT (you can see while formatting disk or pendrive). NTFS - New Technology File System (Microsoft Windows NT).

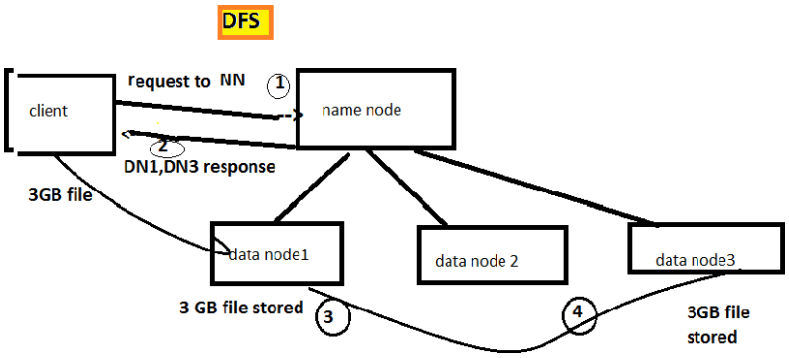
File system is a way in which files are named and where they are stored Or placed logically help us to retrieve the file faster and also efficiently from the memory block or disk.

It also handles the name, location, permission of file.

OS ← FS ← Files and Folders.



A **Distributed File System (DFS)** is a file system that is distributed on multiple file servers or multiple locations. It allows programs to access or store isolated files as they do with the local ones, allowing programmers to access files from any network or computer such as Hadoop file system.
When we want to store the file then it break into small chunks then stored across the machines.



In DFS have two components:
Name node
Data node Name node:
It's like a master node.
It keeps record about the data node.
It also replicate file wherever whenever is needed.
Replication means keep a copy of the file.
It knows information of where the file is stored in the data nodes.
It checks the size of data node, how much free space it has and if machine fails it take decision to move data from that.

Data node:
Which machine going to participate with file storing. It is in some network clients or remote clients.
i.e. it is cluster of computers.

Purpose of DFS:
The main purpose of the Distributed File System (DFS) is to allow users of physically distributed systems to share their data and resources by using a Common File System.
DFS manage the files and folders across multiple computers.
A DFS is executed as a part of the operating system.
In DFS, a namespace is created and this process is transparent for the clients.
DFS is client-server based application
It allows clients to access and process data stored on the server as data are reside own computer.
DFS organize file and directory services.
All files are accessible to all users of global file system and organization is hierarchical and directory based.

DFS has two components:
Location Transparency: Location Transparency achieves through the namespace component.
Redundancy: Redundancy is done through a file replication component.
Replication is a role of service that enables you to efficiently replicate folders between multiple servers (including those referred to by a DFS namespace path)
In the case of failure and heavy load, these components together improve data availability by allowing the sharing of data in different locations to be logically grouped under one folder, which is known as the "DFS root".