Distributed Systems
CT 703

1. Introduction -8
2. Distributed Objects and File System -16
3. Operating System Support
4. Distributed Heterogeneous Applications and CORBA
5. Time and State in Distributed Systems -8
6. Coordination and Agreement -8
7. Replication -8
8. Transaction and Concurrency Control -8
9. Fault Tolerance -8
10. Case Studies (3, 4, 10) -16

1. Introduction

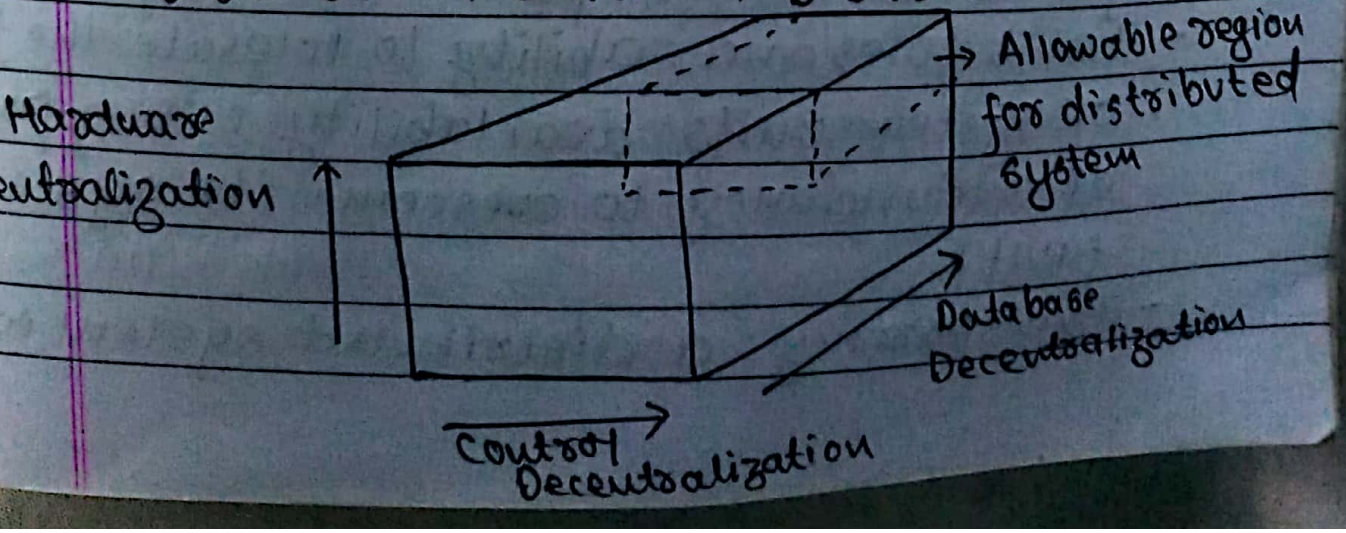
1.1 Introduction to Distributed Systems

- A distributed system is a collection of independent computers that appear to the users of the system as a single computer.
- A distributed system is one in which hardware or software components located at networked computers communicate and coordinate their actions only by message passing.

* Enslow's Definition

- A system can be classified as a distributed system if all three categories (hardware, control and data) reach a certain degree of decentralization

Distributed System = Distributed hardware + Distributed control + Distributed data



1.2 Examples of distributed systems
- Collection of web servers
more precisely, servers implementing the HTTP protocol - that jointly provide the distributed database of hypertext and multimedia documents that we know as WWW (world wide web)
- DNS Servers
- ATM Networks
- Intranet
- Mobile Phone Networks, etc

1.3 Main Characteristics
- Resource Sharing: hardware, software, data
- Openness: openness of main interface of the system, scalability of the current system
- Concurrency: concurrent execution of the processes, high performance, nice rate of price (PC cluster = poor man's supercomputer)
- Fault tolerance: ability to tolerate the fault of system units, availability (using potential redundancy to overcome the system fault)
- Transparency: a distributed system can be looked as one computer (access, position and parallel transparency).

1.4 Advantages and Disadvantages of D.S
* Advantages
- Data Sharing: allow many users to access to a common data base.
- Resource Sharing: expensive peripherals like color printers
- Communication: enhance human to human communication, eg: email, chat.
- Flexibility: spread the workload over the available machines.

* Disadvantages
- Difficult to develop software for distributed systems.
- Network: saturation, lossy transmissions
- Easy access also applies to secrete data, which may arise security problem
- Under distribution of control, hard to detect faults and administration issues.
- Performance
- Interconnect and servers must scale.

1.5 Design Goals

- Transparency

• It is important for a distributed system to hide the location of it's process and resource. A distributed system can only portray itself as a single system only if it is transparent. The various transparencies need to be considered are access, location, migration, relocation, replication, concurrency, failure and pessistence.

- Openness

It offers services according to standard rules that describe the syntax and semantics of those services. Open distributed system must be flexible and extensible making it easy to configure and add new components without affecting existing components.

- Scalable

Scalability in distributed systems are measured along three different dimensions. First, a system can be scalable with respect to it's size which can add more user and resources to a system, second, users and resources can be geographically apart and third, it is possible to manage even if many administrative organizations are spanned.

- Connecting users and resources

The main goal of distributed system is to make it easy for users to access remote resources and to share them with others in a controlled way. Similarly, collaborating and exchanging information can be made easier by connecting users and resources.

1.6 Main Problems

* Design Challenges of Distributed System

- Heterogeneity

Heterogeneous components must be able to interoperate.

- Openness

Interfaces should allow components to be added or replaced.

- Security

The system should only be used in the wa intended.

- Scalability
  System should work efficiently with an increasing number of users.
- Failure Handling
  Failure of a component (partial failure) should not result in failure of the whole system
- Transparency
  Distribution should be hidden from the user as much as possible.

* Pitfalls when developing Distributed systems
  False assumptions made by first time developer
  - The network is reliable
  - The network is secure
  - The network is homogeneous
  - The topology doesn't change
  - Latency is zero.
  - Bandwidth is infinite
  - Transport cost is zero
  - There is one administrator.

1.7 Models of Distributed System
- The main purpose of models is to illustrate or describe common properties and design choices for distributed system in a single descriptive model.
- Three types of models
  - Physical models: capture the hardware composition of a system in terms of computers and other devices and their interconnecting network
  - Architecture models: define the main components of the system, what their roles are and how they interact (software architecture) and how they are deployed in a underlying network of computers (system architecture)
  - Fundamental models: formal description of the properties that are common to architecture models. Three fundamental models are
    - Interaction models
    - Failure models
    - Security models

1. Physical Model

Three generations of distributed system

a) Early distributed systems
- Emerged in the late 1970s and early 1980s because of the usage of local area networking technologies.
- System typically consisted of 10 to 100 nodes connected by a LAN with limited internet connectivity and supported services (eg shared local printer, file servers)

b) Internet-scale distributed systems
- Emerged in the 1990s because of the growth of the internet
- Incorporates a large number of nodes and organizations
- Increasing heterogeneity
- Increasing emphasis on open standards and services and associated middleware such as CORBA and web services.

c) Contemporary distributed systems
- Emergence of mobile computing leads to nodes that are location-independent
- Need to added capabilities such as service discovery and support for spontaneous operation.
- Emergence of cloud computing and ubiquitous computing.

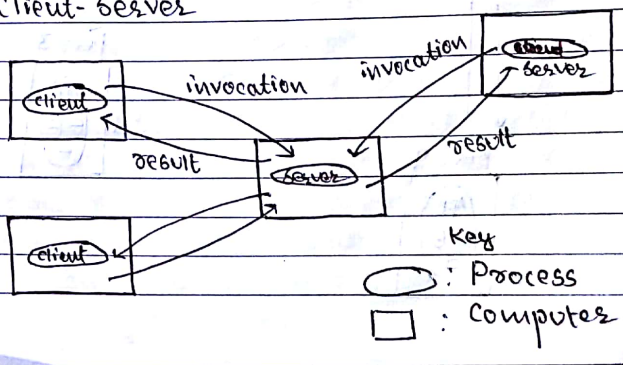
2. Architectural models
- Concern the logical organization of distributed systems into

a) Communicating entities
objects, components and web services

b) Communication paradigms
interprocess communication, remote invocation and indirect communication

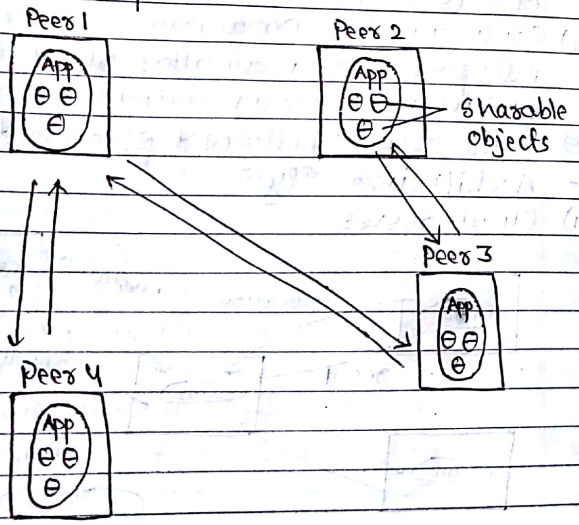
c) Roles, responsibility and placement
- Architectural Styles

1) Client-Server



- Client server offers a direct, relatively simple approach to the sharing of data and other resources but it scales poorly.
- The centralization of service provision and management implied by placing a service at a single address doesn't scale well beyond the capacity of the computer that hosts the service and the bandwidth of its connections.

2) Peer to peer



- composed of large number of peer processes running on separate computers.
- All processes have client and server roles
- patterns of communication between them depends entirely on application requirements.
- Storage and processing and communication loads for accessing objects are distributed across computers and network links
- Each object is replicated in several computers to further distributed the load and to provide resilience in the event of disconnection of individual computers.

3. Fundamental models

1) Interaction model
- Performance of communication channels

a) Latency: delay between the start of a message's transmission from one process and the beginning of its receipt by another
b) Bandwidth: total amount of information that can be transmitted over a computer network in a given time.
c) Jitter: variation in the time to deliver a series of messages.

- Two variants of the interaction model

Synchronous Distributed Systems	Asynchronous Distributed Systems
The following bounds are defined	There are no bounds on
• The time to execute each step of a process has known lower and upper bounds	• Process execution speed
• Each message transmitted over a channel is received within a known bounded time	• Message transmission delays
• Each process has a local clock whose drift rate from real time has known bound.	• Clock drift rate

2) Failure models
- It defines ways in which failure may occur in order to provide an understanding of the effects of failure
- Types of failures
  - Omission failures
    - Fail stop, Crash, Omission, Send-omission and Receive-omission
  - Arbitrary failures (Byzantine failure)
    - Worst possible failure semantics in which any type of error may occur
  - Timing failures
    Clock, Performance,

3) Security model
- The security of a distributed system can be archived by securing the processes and the channels used for their interactions and by protecting the objects that they encapsulate against unauthorized access.
- Access rights are used to specify who is allowed to perform the operations of an object.
- Encryption and authentication are used to build secure channels on top of existing communication services.

1.8 Resource Sharing and the Web Challenges
- Main motivation of DS: resource-sharing
  - hardware: processor, printer, disks

- Software-defined entities: files, database
- Functionality: search engines

- Service
- manage a collection of related resources and present their functionality to users and applications.
- Access resources via a well-defined set of operations

* WWW (world wide web)
- An evolving system for publishing and accessing resources and services across the Internet
- Requires browsers, supported by hypertext linking mechanism to related documents.

1.9 Types of Distributed System

1. cluster
- A cluster is a single logical unit consisting of multiple computers that are linked through a LAN. The networked computers essentially act as a single, much more powerful machine. A computer cluster provides much faster processing speed, larger storage capacity, better data integrity, superior reliability and wider availability of resources.
- Eg: Beowulf cluster, K computer

2. Grid
- A processor architecture that combines computer resources from various domains to reach a main objective. The computers on the network can work on a task together, thus functioning as a ~~compu~~ supercomputer. It is designed to solve problems that are too big for a supercomputer while maintaining the flexibility to process numerous smaller problems.
- Eg: SETI@home, Data Grid at Southern California Earthquake Center

3. Cloud
- Shared pools of configurable computer system resources and higher level services that can be rapidly provisioned with minimal ma-

nagement, effort often over the Internet. Cloud computing relies on sharing of resources to achieve coherence and economies of scale.

- Eg: Google's Gmail, Amazon EC2

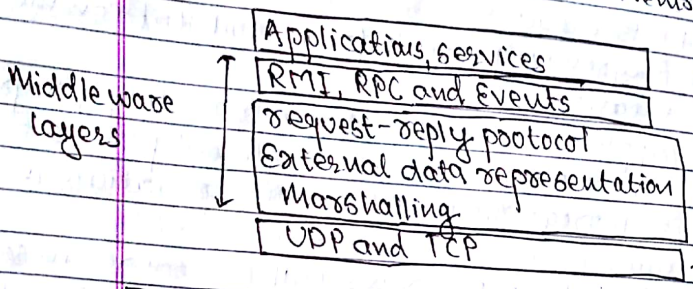
2. Distributed Objects and File System

2.1 Introduction

* Programming models for Distributed communications
- Applications composed of cooperating programs running in several different processes. Such programs need to invoke opesations in other processes.
  - RPC : client programs call procedures in server programs, running in separate and remote computers. Eg: Unix RPC
  - RMI : extensions of object-oriented programming models to allow a local method (of a local object) to make a remote invocation of objects in a remote process. Eg: Java RMI
  - Event-based Programming (EBP) : allows objects anywhere to receive notification of events that occur at other objects in which they have registered interest : Eg: Jini EBP

* Middleware
- A suite of API software that uses underlying processes and communication (message passing)

protocols to provide it's higher level abstracts such as remote invocations and events.



- Transparency Features of Middleware
  - Location Transparency
    In RMI and RPCs, the client calls a procedure /method without knowledge of the location of invoked method/procedure.
  - Transport protocol Transparency
    Request/reply protocol used to implement RPC can use either UPD or TCP
  - Transparency of computer hardware
    They hide differences due to hardware architectures, such as byte-ordering.
  - Transparency of OS
    It provides independency of the underlying operating system.
  - Transparency of programming language used
    Use of programming language independent Interface Definition Language (IDL) such as CORBA IDL.

* Interface Definition Languages (IDL)
  - An IDL is a language that is used to define the interface between a client and server process in a distributed system. It is impossible to specify direct access to variables in remote classes. Hence, access only through specified interface.
  - It is desirable to have language-~~de~~independent IDL that compiles into access methods in application programming language.
  - Interface in distributed system
    - can't access variables directly
    - input argument and output argument
    - pointers can't be passed as arguments or returned results
  - Interface Cases
    - RPC's Service interface
      Specification of the procedure of the server

defining the types of the input and output arguments of each procedure.

- RMI's Remote interface
  Specification of the methods of an object that are available for objects in other processes, defining the types of them.
  may pass objects or remote object references as argument or returned result.
- CORBA IDL Example

```
struct Person{
    string name;
    string place;
    ~~string~~ long year;
};
interface PersonList{
readonly attribute string listname;
void addPerson(in Person p);
void getPerson(in string name, out Person P);
long number();
};
```

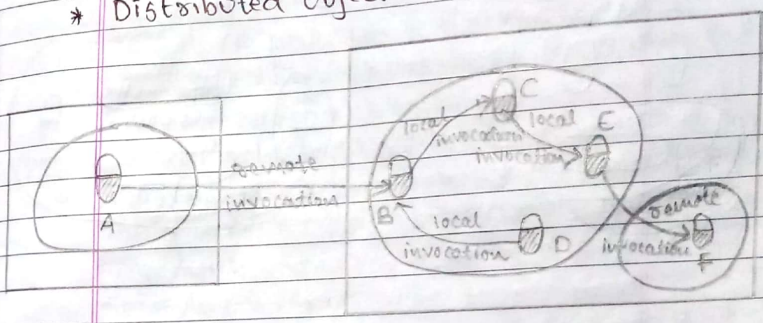
→ Remote interface

remote interface defines methods for RMI

parameters are in, out or inout.

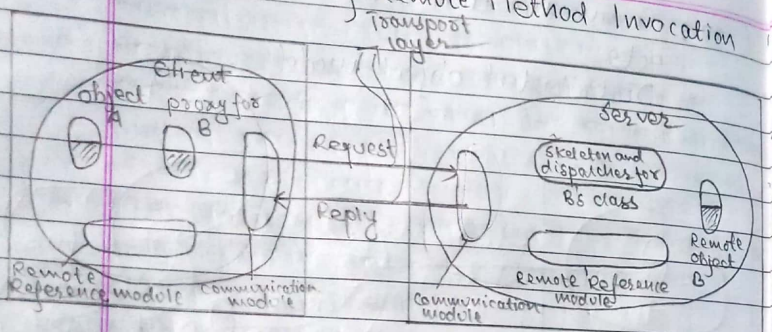
2.2 Communication between distributed objects

* Distributed object model



- each process contains objects, some of which can receive remote invocations, others only local invocations.
- those that can receive remote invocations are called remote objects
- objects need to know the remote object reference of an object in another process in order to invoke its methods.
- the remote interface specifies which methods can be invoked remotely.

* Architecture of Remote Method Invocation



- Communication between object by means of RMI

Extension of local method invocation that allows an object living in one process to invoke the methods of an object living in another process. Method invocations between objects in same process are local method invocation.

- Object modules: consist of collection of interacting objects and have methods and data encapsulation.
- Interfaces: It provides declaration of different method without specifying implementation.
- Exception: Event that occurs during the execution of program that disrupts the normal flow of the program instruction
- Garbage collection: objects no longer needed will be free from memory.
- Proxy: behaves like local object to client, forwards request to remote object, hides the detail of remote object reference, marshalling of arguments and unmarshalling of results and receiving message.
- Dispatcher: receive request, select method and passes on request to skeleton.
- Skeleton: implements method in remote interface, unmarshalles data, invoke remote object, wait for results marshalles it and return reply
- communication module: carry out request-reply protocol
- Uses message type - int (request → 0, reply - 1) request ID → int, remote reference → object type. communication module on server selects the dispatcher for the class of the object to be invoked, passing on its local reference which it

gets from remote module.

- Servents: a servent is instance of class, handles the remote request passed on by corresponding skeleton. The servent lives within server process.
- Remote reference module: responsible for translating between local and remote object references and for creating remote object reference. It has remote object table that records the correspondance between local object reference in that process and remote object reference.
- Remote object reference: object identifiers in distributed system. Must be unique in space and time.

Eg: //host:port/name

2.3 Remote Procedure Call

- RPC is an interprocess communication that allows a computer program to cause a subroutine or procedure to execute in another address space without programmer explicitly coding the details for this remote interaction.

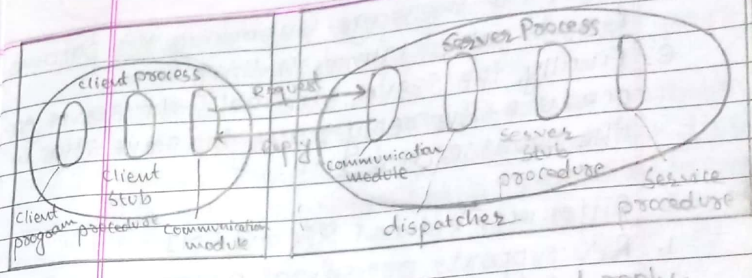


Fig: RPC implementation over request reply protocol

- Sequence of events during RPC

1. The client calls the client stub. The call is a local procedure call, with parameters pushed on to the stack in normal way.
2. The client stub patched the parameter into a message and make a system call to send the message. Packing the parameter is called marshalling.
3. The client's local OS sends the message from the client machine to the server machine.
4. The local os on the server machine passage the incoming packets to the server stub.
5. The server stub unpatched the parameter

from the message. Unpacking the parameters is called unmarshalling.

6. Finally, the server stud calls the server procedure. The reply traces the same steps in the reverse direction.

* Differences between RPC and RMI

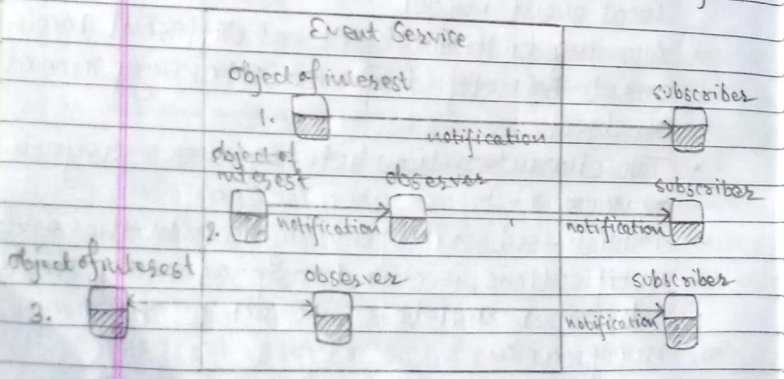
1. RPC supports procedural programming paradigms thus is C based, while RMI supports object-oriented programming paradigms and is java based.
2. The parameters passed to remote procedures in RPC are the ordinary data structures. On the contrary, RMI transits objects as a parameter to the remote method.
3. RPC can only use pass by value method whereas RMI could use pass by value or reference and parameters passed by reference can also be changed.
4. RPC protocol generates more overheads than RMI
5. The parameters passed in RPC must be "in-out" which means that the value passed to the procedure and output value must have the same data types. In constrast, there is no compulsion of passing "in-out" parameters in RMI.
6. In RPC, references could ~~be~~ not be probable because the two processes have the distinct address space, but it is possible in case of RMI.

2.4 Events and Notifications

- Distributed event-based systems extend local event model
- Allowing multiple objects at different locations to be notified of events taking place at an object.
- Two characteristics: heterogeneous, asynchronous
- Publish-subscribe paradigm: publishes sends notifications, ie objects representing events
- Subscriber registers interest to receive notificatians
- The object of interest: where events happen, change of state as a result of its operations being invoked
- Events: occurs in the object of interest

- Notification: an object containing information about an event.
- Subscriber: registers interest and receives notifications
- Publisher: generate notifications, usually an object of interest
- Observer objects: decouple an object of interest from its subscribers

* Architecture for distributed event notification



2.6 Introduction to DFS

- A Distributed File System (DFS) is a classical model of a file system distributed across multiple machines. The purpose is to promote sharing of dispersed files.
- A file system provides a service for clients. The server interface is the normal set of file operations: create, read, etc on files.
- A DFS enables programs to store and access remote files exactly as they do on local ones, allowing users to access files from any computer on the intranet.
- Naming schemes in DFS
  - Files are named with a combination of host and local name.
  - Remote directories are mounted to local directories.
  - A single global name structure spans all the files in the system.

2.7 File Service Architecture

- An architecture that offers a clear separation of the main concerns in providing access to files is obtained by structuring the file

service as three components:
- a flat file service
- a directory service
- a client module

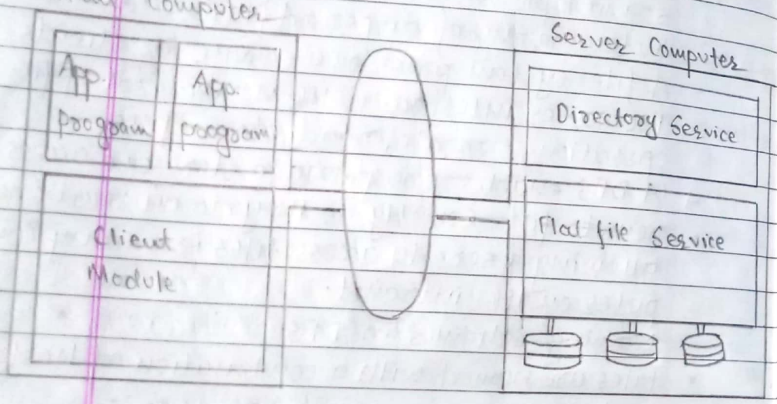


Fig: File Service Architecture

- The client module implements exported interfaces by flat file and directory services on server side.
- Responsibilities of various modules can be defined as follows:

- Flat file Service
  - Concerned with the implementation of operations on the contents of file. Unique File Identifiers (UFIDs) are used to refer to files in all requests for 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.

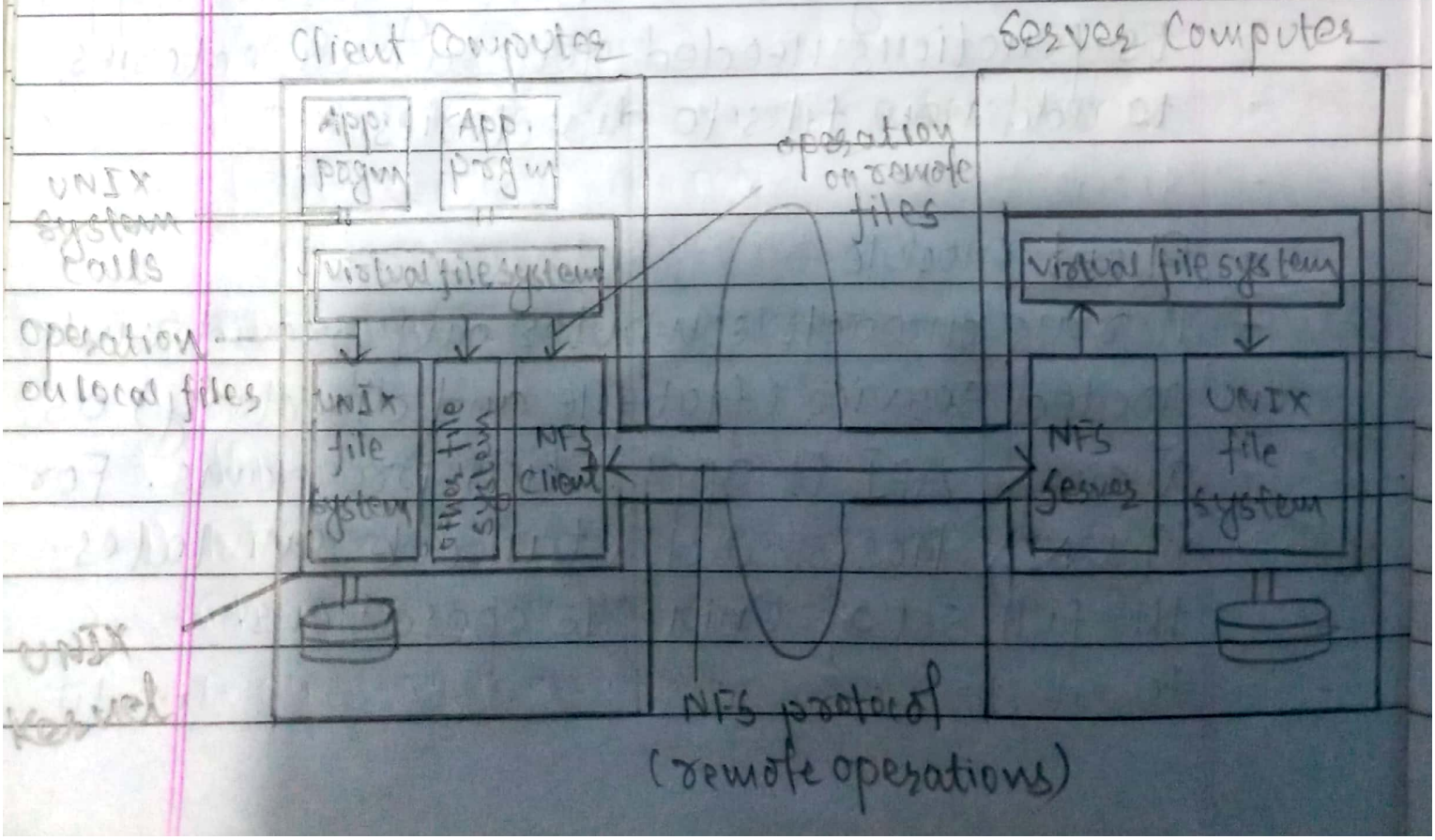
- Client module
  - It runs on each computer and provides integrated service (flat file and directory) as a single API to application programs. For in UNIX hosts, a client module emulates the full set of Unix file operations.

- It hold 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.

2.8 Sun Network File System

- First commericially successful network file system developed by Sun Microsystems for their diskless workstations.
- Designed for robustness and "adequate performance"

* Architecture of NFS



- Three important parts

1) The protocol
- uses the Sun RPC mechanism and Sun eXternal Data Representation (XDR) standard.
- Defined as a set of remote procedures
- Protocol is stateless ie, each procedure call contains all the information necessary to complete the call and server maintains no "between call" information.

2) The server
- Server implements a write-through policy
  - required by statelessness
  - any blocks modified by a write request (including i-nodes and indirect blocks must be written back to disk before the call completes
- File handle consists of
  - Filesystem id identifying disk partition
  - I-node number identifying file within partition
  - Generation number changed every time i-node is reused to store a new file.
- Server will store
  - Filesystem id in filesystem superblock

• i-node generation number in i-node

3) The client side
- Provides transparent interface to NFS
- mapping between remote file names and remote file addresses is done at server boot time through remote mount specified in mount table and which makes a remote subtree appear part of a local subtree.
- New virtual filesystem interface supports
• VFS calls which operate on whole file system
• VNODE calls which operate on individual files
- Treats all files in the same fashion.

* Advantages
- Machine and operating system independent
- Fast crash recovery
- Transparent access
- Reasonable performance

* Disadvantages
- NFS root file system cannot be shared.
- Client can mount any remote subtree any way they want.
- NFS has no file locking
- NFS tries to preserve UNIX open file semantics but does not always succeed.

2.9 Introduction to Name Services
- A name is human-readable value (usually a string) that can be resolved to an identifier or address. Eg internet domain name, file pathname, etc.
- A name service stores a collection of one or more naming contexts - set of bindings between textual names and attributes for objects such as users, computers, services and remote objects.
- The major operation is to resolve a name ie, to look up attributes from a given name.
-

2.10 Name Services and DNS
- Name services returns the information about the resource when the name of the resource is given. Eg: Consider the white pages given the name of a person you get the address/telephone number of that person.

* DNS (Domain Name System)
- It is a hierarchical decentralized naming system for computers, services or other resources connected to the internet or a private network.
- It translates more readily memorized domain names to the numerical IP addresses needed for locating and identifying computer services and devices with the underlying network protocols.
- Services provided by DNS
  - Host aliasing: pointing multiple domain to same machine. Eg facebook.com and fb.com
  - Mail server aliasing:
  - Load distribution: load distribution among replicated web servers. Eg cnn.com → replicated over multiple servers with each servs running on different ip address.
  - DNS caching: faster response, fastes access

* DNS Query
1) Recursive Query: resolves and gives IP address
2) Iterative: It will not go and fetch the complete answer for your query but will give back a referral to other DNS servers, which might have the answer

* DNS Resource Record
- A record: Resolves a host name to IP address
- PTR record: Resolves an IP address to a host name
- SOA record: The first record in any zone file.
- NS record: Identifies the DNS server for each zone.

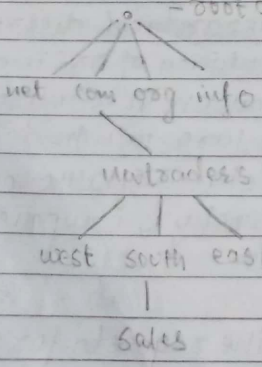
* DNS is a hierarchical distributed database
- The implementation of DNS is done in a "distributed manner". It is implemented as a collection of large number of servers, organized hierarchically all over the world.
- The DNS hierarchy is comprised of the following elements
1) Root level
It answers the requests for records in the

2.11 ~~Directory and~~ root zone and other requests by providing a list of authoritative name servers for the appropriate TLD.

2) Top level Domain
It recognizes a certain element regarding the associated website such as it's objective, owner or the geographical area from which it originated.

3) Second level Domain
It is specific part of a website, page domain name or URL address that complements a TLD.

4) Sub-domain
It is used to create a more memorable web address for specific or unique content with a website.



2.11 Directory and Discovery Services
- Directory services find a service or resource that matches the description given by the user. It is more powerful than naming as attributes can help. For eg: consider the yellow pages: when you want to rent a car, it may give a list of car rental agencies.
- UDDI (universal directory and discovery service) provides both white pages and yellow pages services ie, provide information about organizations and the web services they provide.
- Discovery service: a special case of a directory service for services provided by devices in a spontaneous networking environment
  - automatically updated as the network configuration changes.
  - discovers services required by a client

3. Operating System Support

3.1 The operating system layer

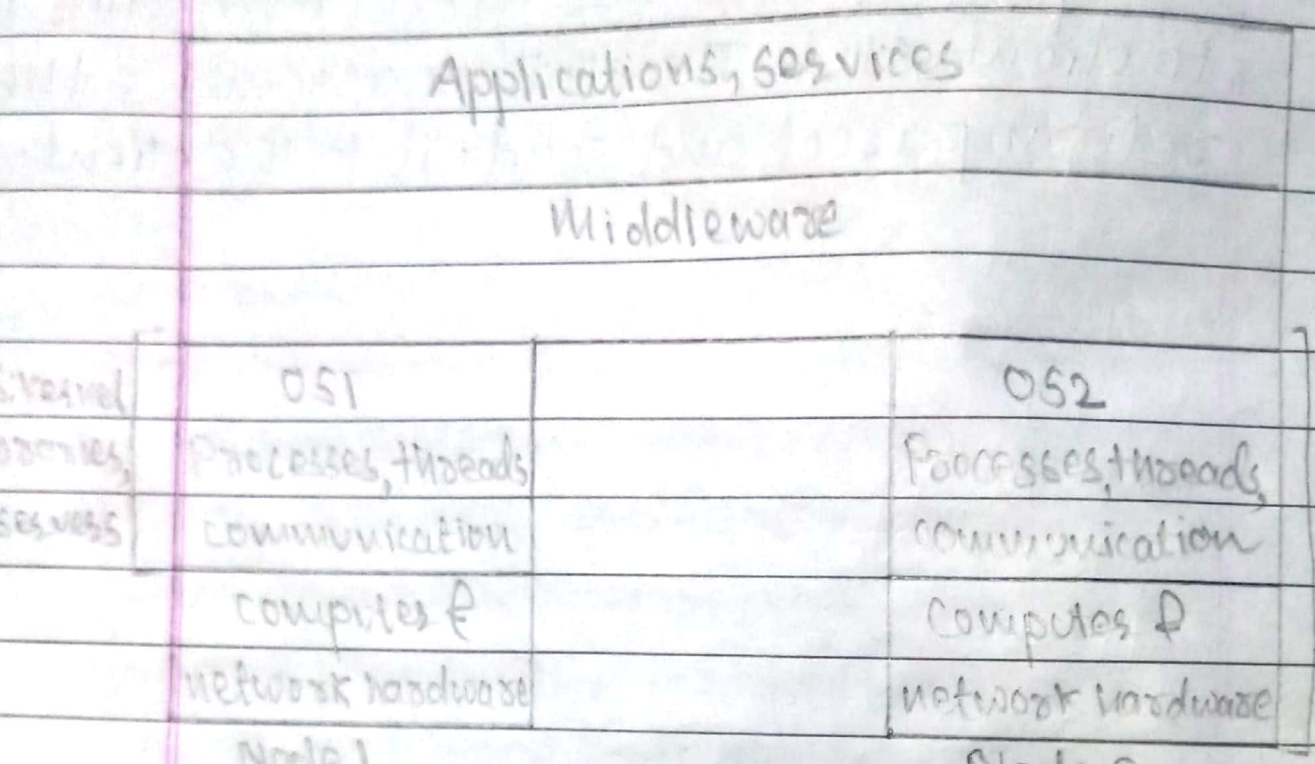
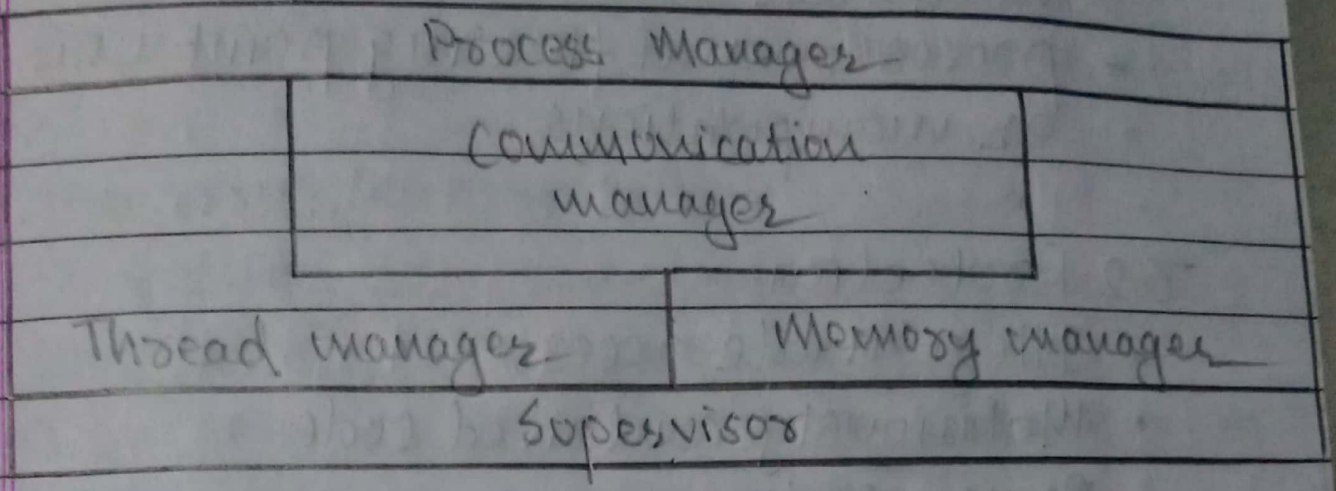


Fig: System layers

- The above figure shows how the operating system at each of two nodes supports a common middleware layer in providing a distributed infrastructure for applications and services.
- Kernels and server processes are the components that manage resources and present clients with an interface to the resources.

- The OS facilitates: Encapsulation, protection and concurrent processing.
- The core OS components are shown in the figure below



- ~~3.2~~ ~~Protection~~ Process Manager
- Handles the creation of and operations upon processes.
- Thread Manager
- Thread creation, synchronization and scheduling
- Communication Manager
- Communication between threads attached to different processes on the same computer
- Memory Manager
- Management of physical and virtual memory

- Supervisor
  - Dispatching of interrupts, system call traps and other exceptions
  - control of memory management unit and hardware caches
  - processor and floating point unit register manipulations.

3.2 Protection
- Illegitimate access
- Maliciously contrived code
- Benign code
  - contains a bug
  - have unanticipated behavior
- Example : read and write in File System
  - Illegal user vs access right control
  - Access the file pointer variable directly (set File Pointer Randomly) vs type safe language
    Type-safe language - Java
    Non-type safe language - C++, C

* Kernel and Protection
- Kernel always runs and complete access privileges for the physical resources.

- Different execution mode
  - supervisor mode (kernel process / user mode (user process)
  - Interface between kernel and user processes : system call trap.
- Kernel design is a good choice for protection
- The price for protection
  - switching between different processes take many processor cycles.
  - a system call trap is a more expensive operation than a simple method call.

3.3 Process and threads
- Process
  - A process consists of an execution environment together with one or more threads.
  - A thread is the operating system abstraction of an activity
  - An execution environment is the unit of resource management ie, a collection of local kernel managed resources to which its threads have access.

- An execution environment consist of
→ an address space
→ thread synchronization and communication resources (eg: semaphores, sockets)
→ Higher-level resources (eg: file systems)

• Threads
- Threads are schedulable activities attached to process
- The aim of having multiple threads of execution is
→ to maximize degree of concurrent execution between operations.
→ to enable the overlap of computations with input and output.
→ to enable concurrent processing on multiprocessors.
- Threads can be helpful within servers
→ concurrent processing of client's requests can reduce the tendency for servers to become bottleneck. Eg: one thread can process a client's request while a second thread serving another request waits for a disk access to complete.

* Differences between process and thread

Basis for comparison	Process	Thread
Basic	Program in execution	Lightweight process of part of it
Memory Sharing	completely isolated and do not share memory	Shares memory with each other
Resource consumption	more	less
Efficiency	less efficient in the context of communication.	Enhances efficiency in context of communication
Time req. for creation	more	less
Context Switching Time	Takes more time	consumes less time.
Uncertain Termination	Results in loss of process.	A thread can be reclaimed.
Time req. for termination	more	less

3.4 Communication and invocation

• Communication primitives and protocols

- Communication primitives

→ TCP (UDP) Socket in Unix and Windows

-

- Protocols and openness

→ provide standard protocols that enable internal working between middleware.

→ integrate low level protocols without upgrading their application

→ Static Stack: new layer to be integrated statically as a driver

→ Dynamic Stack: protocol stack be composed on the fly.

• Invocation Performance

- Invocation costs are influenced by the factor such as synchronous / asynchronous, domain transition, communication across a network, thread scheduling and switching.

- Invocation over the network

→ Delay: the total RPC call time experienced by a client.

→ Latency: the fixed overhead of an RPC, measured by null RPC

→ Throughput: the rate of data transfer between computers in a single RPC.

→ Main components accounting for remote invocation delay

1) Marshalling and unmarshalling
2) Data copying
3) Packet initialization
4) Thread scheduling and context switching
5) Waiting for acknowledgements
6) Network transmission.

3.5 Operating system architecture

1. Monolithic Kernels

- A monolithic kernel can contain some server processes that execute within its address space, including file servers and some networking.
- The code that these processes execute is part of the standard kernel configuration.
- Kernel is massive: perform all basic operating system functions, megabytes of code and data.

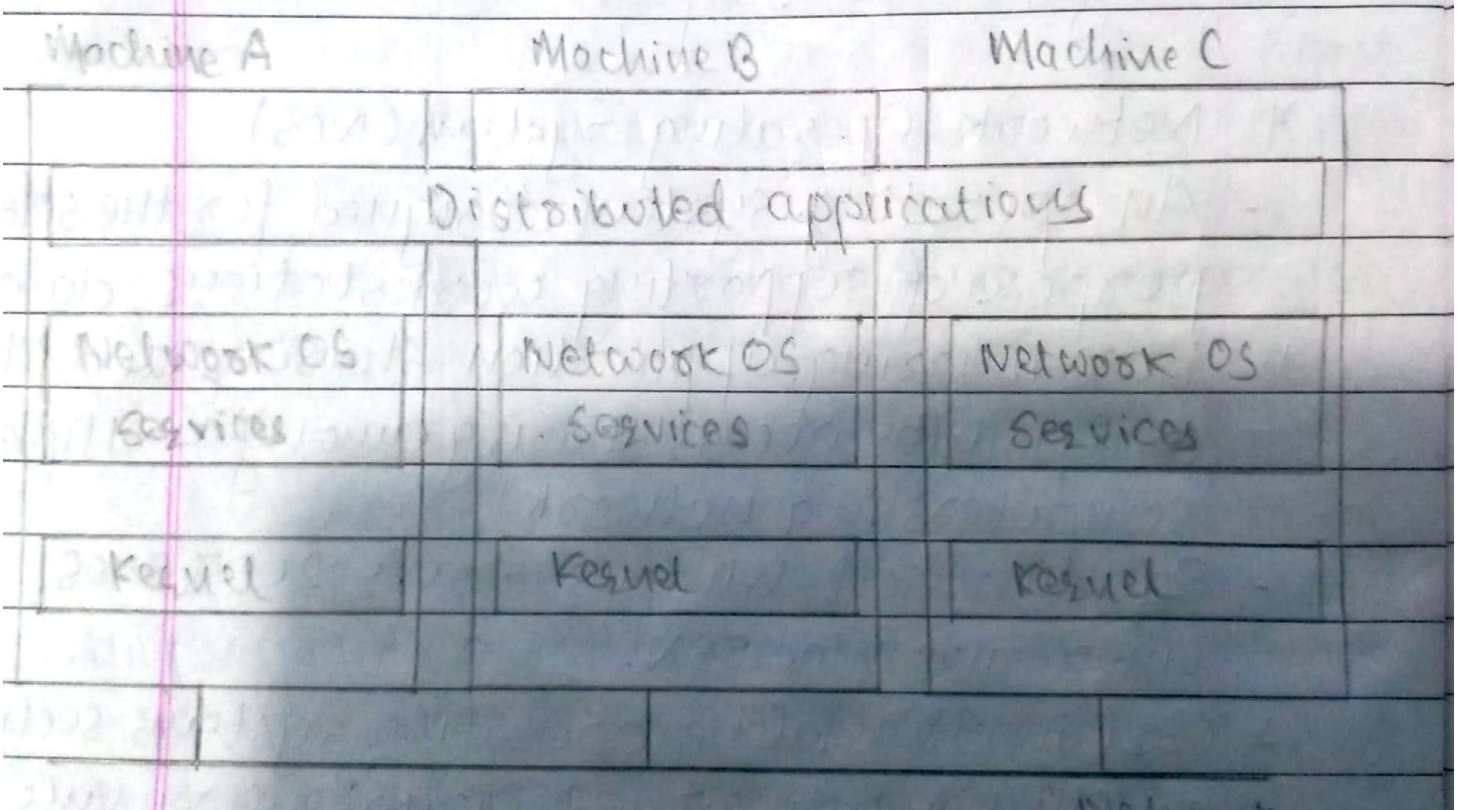
- Kernel is undifferentiated: coded in a monolithic way.
- Eg: Unix
- Pros: efficiency
- Cons: lack of structure

2. Microkernel
- The microkernel appears as a layer between hardware layer and a layer consisting of major systems.
- If performance is the goal, rather than portability, then middleware may use the facilities of the microkernel directly.
- Kernel provides only the most basic abstractions: address spaces, threads and local interprocess communication.
- All other system services are provided by servers that are dynamically loaded.
- Eg: VM of IBM 370
- Pros: extensibility, modularity, free of bugs
- Cons: relatively inefficiency

3. Hybrid Approaches
- Pure microkernel operating system such as Chorus and Mach have changed over a time to allow servers to be loaded dynamically into the kernel address space or into a user level address space.
- In some OS such as SPIN, the kernel and all dynamically loaded modules grafted onto the kernel execute within a single address space.

* Network Operating System (NOS)
- An operating system designed for the sole purpose of supporting workstations, database sharing, application sharing and file and printer access sharing among multiple computers in a network.
- Eg: Microsoft Windows Server 2003, 2008, Linux and Mac OSX.
- Certain standalone operating systems such as MS Windows NT and Digital's Open VMS come with multi purpose capabilities and can also act as Network operating System

* Characteristics of NOS
- extension of centralized operating systems
- offer local services to remote clients
- each processor has own OS.
- User owns a machine but can access others (eg. rlogin, telnet)
- no global naming of resources
- system has little fault tolerance.



Loosely-coupled operating system for heterogeneous multi-computers (LAN and WAN)
Weak transparency

* Distributed operating system (DOS)
- It is the logical aggregation of operating system software over a collection of independent, networked, communicating and physically separate computational nodes.
- A model where distributed applications are running on multiple computers linked by communications.
- This system looks to it's users like an ordinary centralized operating system but runs on multiple, independent CPUs.
- Individual nodes each hold a specific software subset ~~of the~~ which contains two distinct service provisioners.
. Kernel - that directly controls the node's hardware
. System management components: that coordinate the node's individual and collaborative activities
- Eg: IRIX OS, DYNIX OS, OSF/1 OS

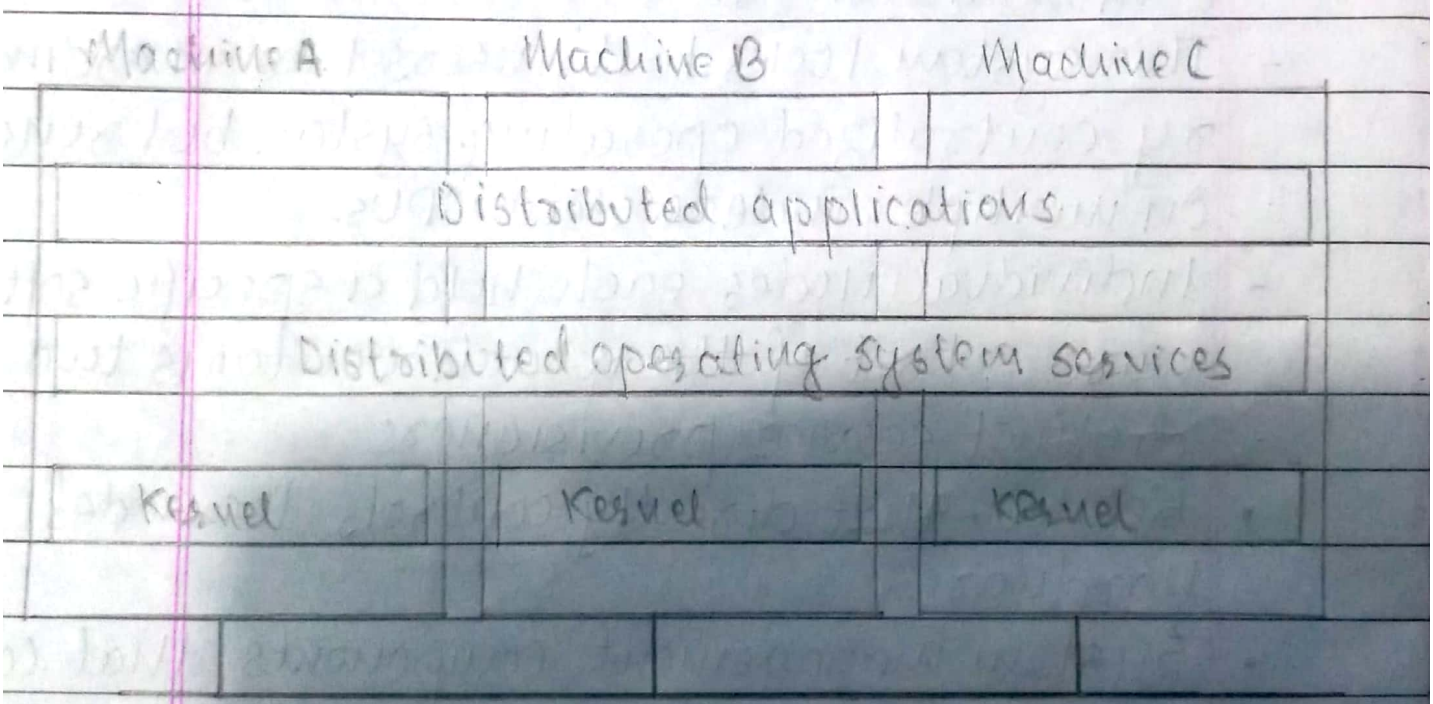
*

* Characteristics of DOS
- Allows a multiprocessor or multicomputer

strong transparency

network resources to be integrated as a single system image.

- Hide and manage hardware and software resources
- provides transparency support
- provides heterogeneity support
- global addressing and naming
- synchronization and deadlock avoidance



Tightly-coupled operating system for multi-processors and homogeneous multi-computers

Network

* Why NOS is preferred over DOS?
- Users have much invested in their application software, they will not adopt a new OS that will not run their applications.

- Users tend to prefer to have a degree of autonomy for their machines even in an organization.

* Differences between NOS and DOS

NOS	DOS
NOS also referred to as the Dialogue.	DOS also refferred to as the Middleware.
NOS are loosely coupled OS for heterogeneous multi-computers whose goal is to offer the local services to remote client.	DOS is tightly-coupled OS for multi processors and homogenous multi-computers whose goal is to hide and manage hardware resources
In NOS, users can access remote resources by either logging in to remote machine or transferring data from remote machine.	In DOS, users access remote resources in the same way they access local resources.
The System will have no operation when any of the terminals will fail or loose message.	Processing unit failures are independent.

NOS doesn't reside on every computer, the client only has enough software to boot the hardware and contact the server. All the subsequent operations are done on the server and client relay the i/p and o/p b/w server and user.	DOS is a middleware shared on all system that knows which computers are overloaded and which ones are idle It would then balance the tasks available so that each computer in the group is sharing equal load.

4. Distributed Heterogeneous Applications and CORBA

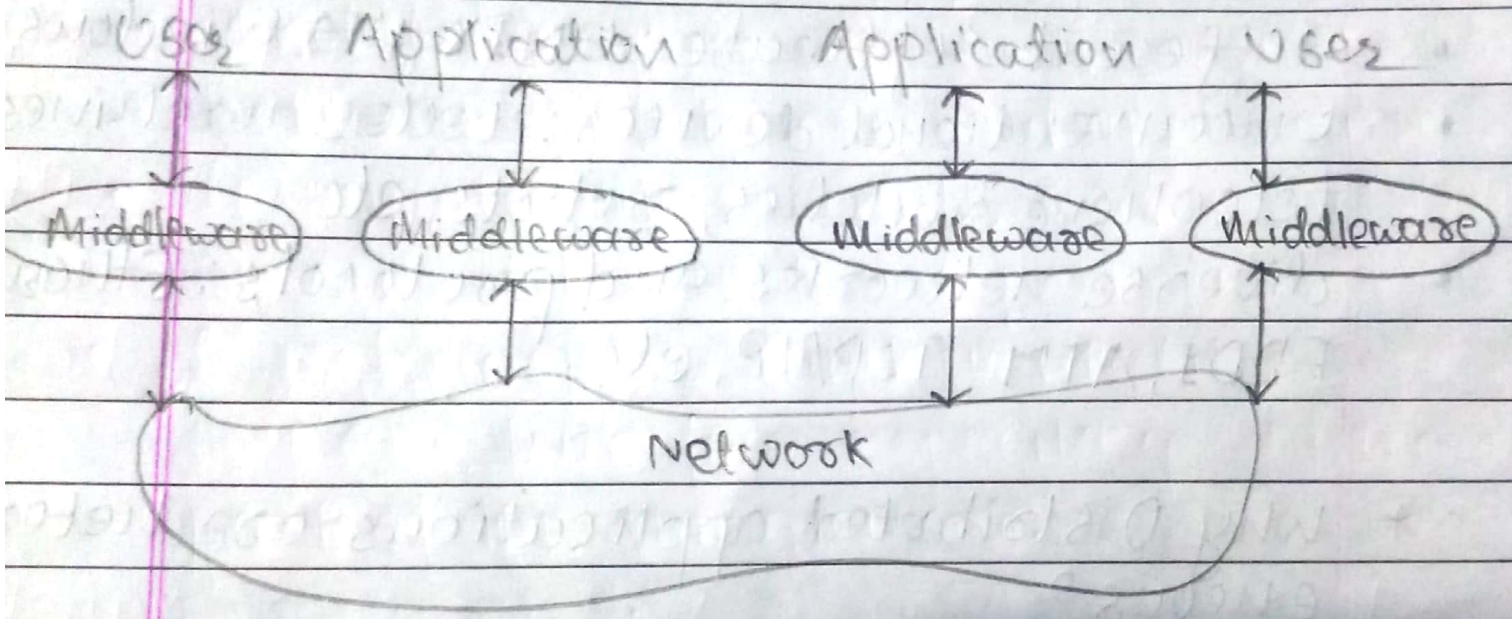
4.1 Heterogeneity in Distributed Systems

- Distributed applications are typically heterogeneous
  - different hardware: mainframe, workstations, PCs, server, etc.
  - different software: UNIX, MS windows, etc
  - unconventional devices: teller machines, telephone switches, robots, etc
  - diverse networks and protocols: Ethernet, FDDI, ATM, TCP/IP, etc.

* Why Distributed applications are heterogeneous?
  - Different hardware/software solutions are considered to be optimal for different parts of the system.
  - Different users which have to interact are deciding for different hardware/software solutions/vendors
  - Outdated systems also known as legacy systems

4.2 Middleware

- Middleware is a set of services that enable applications and end users to interact with each other across a heterogeneous distributed system.
- It resides above the network and below the application software.



* Functions

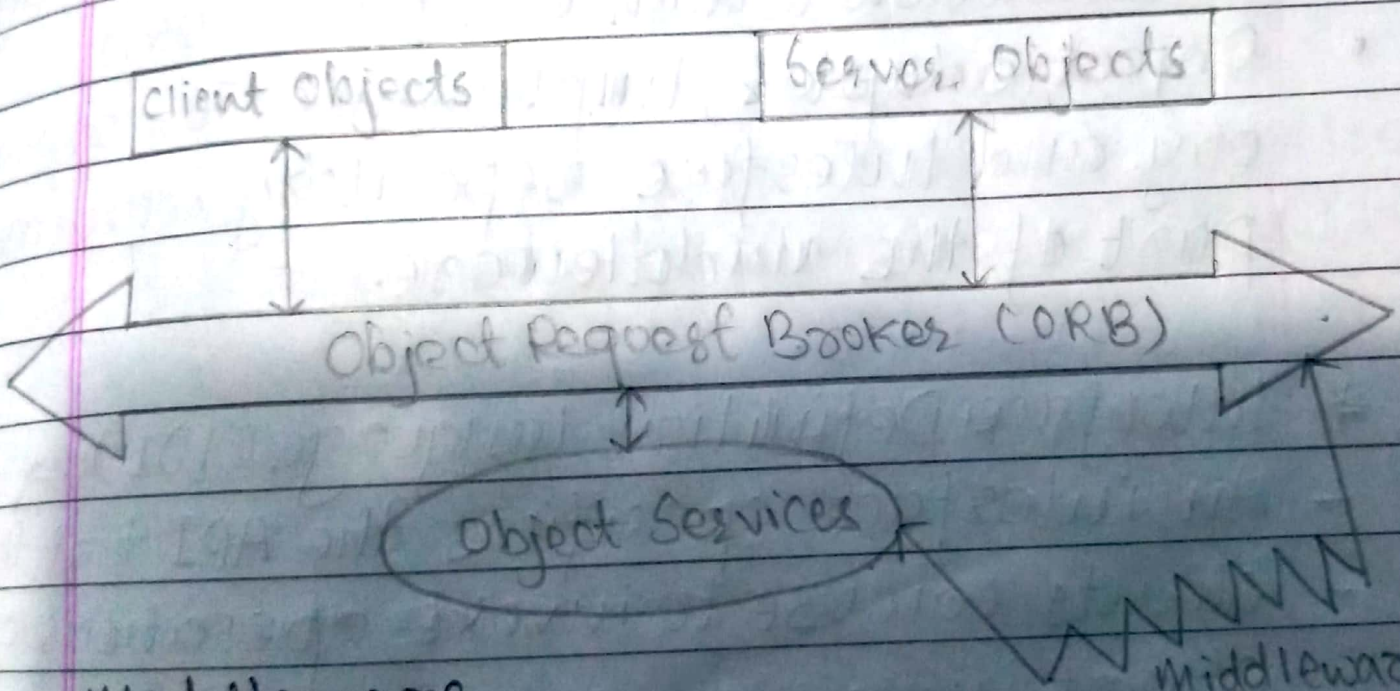
- Middleware should make the network transparent to the applications and end-users
- Middleware should hide the details of computing hardware, OS, software components across networks.

- Example: CORBA, web browsers and FTP and

email to some extent.

4.3 Objects in Distributed Systems

- A distributed application can be viewed as a collection of objects (user interfaces, databases, application modules, customers)
- objects are data surrounded by code, each one has it's own attributes and methods which define the behavior of the object where objects can be clients, servers or both



- Middleware

• Object Request Broker (ORB) allow objects to find each other in a distributed system and interact with each other over a network. They are the backbone of the distributed

object-oriented approach.
- object services allow to create, name, move, copy, store delete, restore and manage objects.

- If we take RMI into consideration, we can recognize some components
  - client and server objects
  - Skeleton and proxy which are on the borders between middleware and application
  - Communication module and remote reference module which are part of ORB
  - object adapter, Implementation repository and Interface repository which are part of the middleware.

* Interface Definition language (IDL)
- An interface specifies the API that the clients can use to invoke operations on objects.
  - set of operations
  - the parameters needed to perform the operations

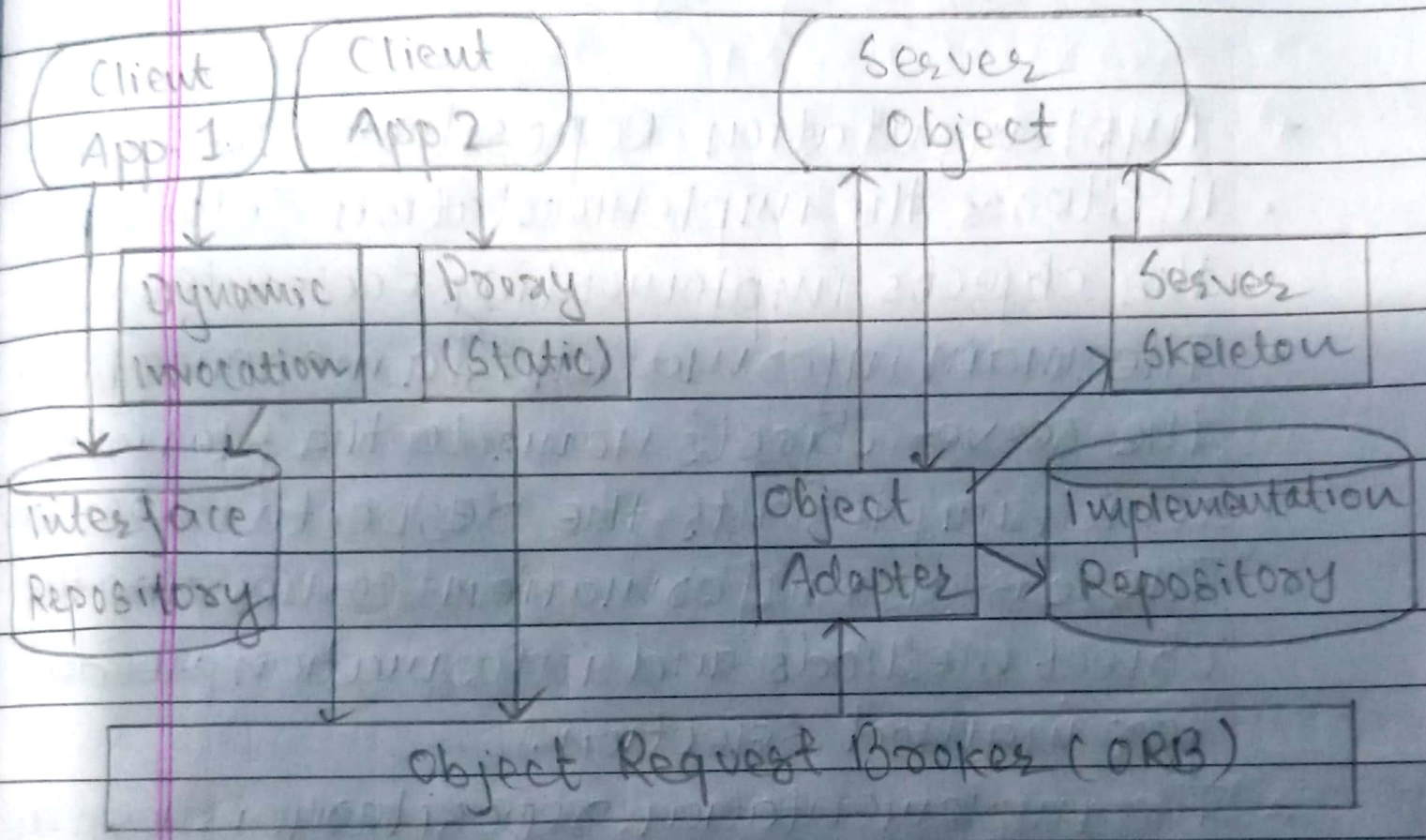
- Interfaces are defined by using IDL.
- Middleware products (CORBA) provide interface compilers that parse the IDL description of the interface. Such a compiler produces the code which represents
  - the classes corresponding to the proxies (in the language of the client).
  - the classes corresponding to the skeletons (in the language of the server).
  - If the client or server are not in an object oriented language, the compiler generates a client stub and server stub respectively.
- IDLs are declarative language ie, they do not specify any executable code but only declarations.
- IDLs should be implementation language independent but language mappings have
  - to be defined which allow to compile the IDL interface and to generate proxies and skeletons.

4.4 The CORBA approach

- Object Management Group (OMG) a non-profit industry consortium with the goal to develop, adopt and promote standards for the development of distributed heterogeneous applications.
- CORBA is the OMG specification of an object Request Boker.
- The CORBA specification details the interfaces and characteristics of the ORB. It practically specifies the middleware functions which allow application objects to communicate with one another no matter where they are located, who has designed them and in which language they are implemented.
- CORBA supports static as well as dynamic binding.
- The interface represents the contract between client and server, and an IDL has been defined for CORBA.
- CORBA objects do not know the underlying implementation details. Object adapter maps the generic model to a specific

implementation.

* Components of CORBA environment

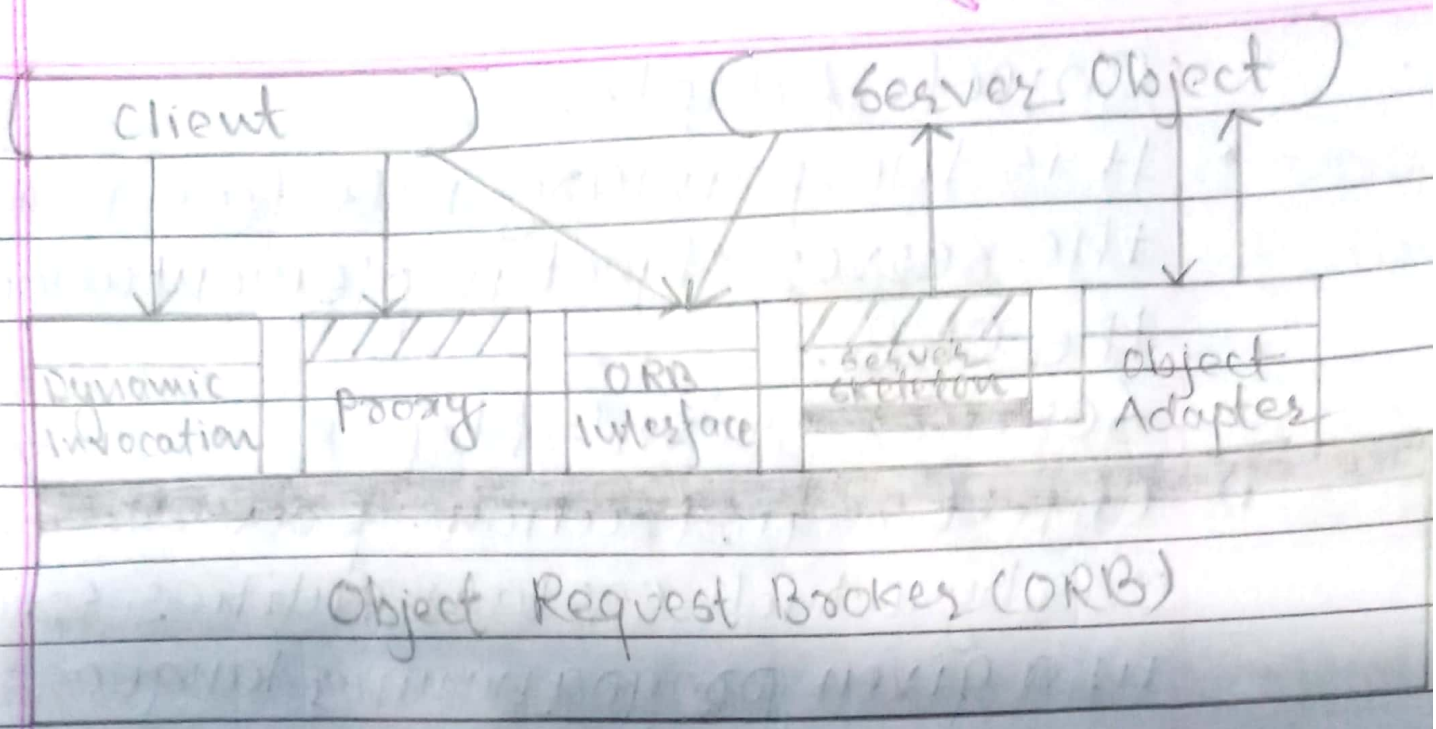


- Interface Repository
- It provides a standard representation of available object interfaces for all objects in the distributed environment. It corresponds to the server object's IDL specification.
- The clients can access the interface repository to learn about the server objects,

determine the types of operations which can be invoked and the corresponding parameters. This is used for dynamic invocation of objects.

- Implementation Repository
  - It stores the implementation details for the objects implementing each interface.
  - The main information is a mapping from the server object's name to the file name which implements the respective service. There is also information concerning the object methods and information needed for method selection.
  - The implementation repository is used by the object adapter (OA) inorder to solve an incoming call and activate the right object method (via a server skeleton)

- Object Request Broker (ORB)
  - The ORB through its interfaces, provides mechanisms by which objects transparently interact with each other



Object Request Brokers (ORB)

ORB implementation dependent interface

Interface identical for all ORB implementations

Proxies and skeletons for each server interface

- Issuing of a request from a client is performed through the proxies (client stubs) or the dynamic invocation interface.
- Invocation of a specific server method is performed by the server skeleton which gets the request forwarded from the object adapter.
- ORB interface can be accessed directly by clients and object implementations for certain services. Eg: directory services, services connected to naming.

- Object Adapter
  - It is the primary interface between the server object implementation and the ORB.
  - Services provided by OA

i) Object registration: provides operations by which certain entities specified in a given programming language are registered as CORBA objects.

ii) Object reference generation: generates object references to CORBA objects

iii) Object upcalls: dispatches incoming requests to the corresponding registered objects.

iv) Server process and object activation: If required, it starts up server processes and activates objects as result of incoming invocations.

* Static and Dynamic Invocation
  - CORBA allows both static and dynamic invocation of objects. The choice is made depending on how much information, concerning the server object, is available at compile time.

* Static Invocation

- Static invocation is based on compile time knowledge of the ~~serv's~~ server's interface specification. This specification is formulated in IDL and is compiled into a proxy (client stub), corresponding to the programming language in which the client is encoded.
- For the client, an object invocation is like a local invocation to a proxy method. The invocation is then automatically forwarded to the object implementation through the ORB, the OA and the skeleton.
- Static invocation is efficient at run time, because of the relatively low overhead.

* Dynamic Invocation

- Dynamic invocation allows a client to invoke requests on an object without having compile time knowledge of the object's interface.
- The object and it's interface (methods, parameters, types) are detected at run-time. CORBA provides through the dynamic invocation interface, the mechanisms in order to inspect the interface repository, to dynamically construct

invocations and provide argument values corresponding to the server's interface specification.
- Once the request has been constructed and arguments placed, it's invocation has the same effect as a static invocation.
- The execution overhead of a dynamic invocation is huge.

4.5 CORBA Services

- Naming and Trading Services
  - The basic way an object reference is generated is at creation of the object when the reference is returned.
  - Object references can be stored together with associated information (names and properties).
  - The naming service allows clients to find objects based on names.
  - The trading service allows clients to find objects based on their properties.
- Transaction Management Service provides two phase commit coordination among recoverable components using

transactions.

- Concurrency control service
provides a lock manager that can obtain and free locks for transactions or threads.
- Security service
protects components from unauthorized users. It provides authentication, access control lists, confidentiality.
- Time service
provides interface ~~by~~ for synchronizing time, provides operations for defining and managing time-triggered events.

* How do we solve interaction between objects which are running on different CORBA implementations?
- CORBA 2.0 defines protocols for interaction between ORBs implemented by different vendors.
- General Inter-ORB Protocol (GIOP) and Internet Inter-ORB Protocol (IIOP) are protocols specified in CORBA 2.0 which specifies a set of message formats and common data representations for interaction between ORBs.