Module 2

Message passing: interprocess communication requires info sharing between two or more processes on the same or on different systems. Two ways for such: 1. Shared data or shared memory (usually not the case as computers on the same network dont share memory) or 2. Copied data or message passing, i.e. interprocess communication between sender and receiver.

Features of Good IPC:

1. Simplicity: simple to construct and send messages, i.e. construct new applications by commands
2. Uniform semantics: so that syntax of message passing on local communication (process on same node) should be as similar as possible to global communication.
3. Efficiency: not expensive, tried by avoiding setting up and terminating connections after each message, and not sending acknowledgement directly.
4. Correctness: Atomicity- message reaches to all users or none, ordered delivery, and survivability
5. Reliability: Message will reach, acknowledgements and duplicates used, and duplicates should also be correctly identified.
6. Flexibility: User should be able to choose the degree of correctness, like in routing, where message lost is better than delayed delivery, or first come first serve, where atomicity is not required.
7. Security: authentication of receiver by server and server by receiver, and end to end encryption.
8. Portability: Message passing system and the application created by its primitives should be portable. Design of different primitives for different systems on heterogeneous networks should be hidden from user.

Parts of Message:

1. Address: sender, receiver
2. Sequence number: to handle lost and duplicate message
3. Content: further has a section called type, which specifies whether message is present there itself or only has a pointer to some other part where actual content is stored.

Synchronization:

Types of primitives:

Blocking send: sender is blocked until execution

Non blocking send: sender can resume execution as soon as it nows message has been copied to buffer

Blocking receive: receiver is blocked until receive message

Non blocking receive: Receiver resumes execution when it knows message has been copied to buffer

A good message passing system has all types of primitives.

Blocking send+blocking receiver=synchronous communication. Otherwise, asynchronous communication.

To prevent deadlock, or indefinite wait due to message pass, a timeout (user defined or default) can be defined in blocking send/receive. Synchronous reduces concurrency, may cause indefinite wait or deadlock, and may need unnecessary acknowledgements, multithreading helps with this.
In non blocking receive, receiver can know message has been put up in buffer by 2 ways:

1. Polling: a test primitive is used to regularly poll buffer to check message received.
2. Interrupt: receiver resumes execution, and interrupt is made when message is posted in buffer. Difficult to code. Provides max parallelism

BUFFERING
A buffer may be introduced in kernel's memory or receiver's memory to store sender's messages. Associated with synchronous communication.

1. Null buffering: No buffer space allocated. Messages are dealt with on 2 ways:

a. Message remains in sender, and sending is blocked until receiver puts out receiver receive. Then sender sends.
b. Sender send automatically, and is blocked. If receiver is free, it immediately acknowledges, otherwise simply discards. After timeout, sender sends again. Sender gives up after trying some predefined number of times.

2. Single message buffer: requires 2 copy operations. A buffer, possible to hold only single message, in kernel's space or receiver's space. In most synchronous systems, there will be only 1 outstanding message at a time, built on this concept.
Null buffer is no use as it takes so much time for packet to be sent across network, only to be discarded. This is a waste of resources.
3. Unbounded buffer: Not practically possible to construct. Can support asynchronous communication, as all messages sent will be stored in buffer.
4. Finite size buffer: Requires 2 copy operations. Size is specified by receiver by create-buffer instruction. If made in kernel's space, allocated accordingly to message sizes. Otherwise made in receiver's space. Helps in better concurrency, but difficult to manage, as protection etc will have to handled. Deals with buffer overflow in 2 ways:

a. Unsuccessful communication: Sends back error message if message given could not be kept. Makes message passing unreliable as there is chance it will not be accepted.
b. Flow control: Blocks send until few messages have been accepted by receiver, making some space, then unblocks send and allows to send messages. Drawback: is not completely asynchronous as it blocks send.

MULTIDATAGRAM MESSAGES:
There is a maximum bound on data that can be transferred at a time, called MTU(Maximum Transfer Unit). Messages smaller than this size are single datagram messages, others are multidatagram messages, and have to be broken down into fragments, sequentially. All fragments have a sequence number and al, have some data and some control message. Responsibility of assembling and reassembling at sender and receiver is responsibility of message passing system.

Dealing with missing or out of sequence packets in multidatagram messages is by following ways:

1. Stop and wait protocol: separate acknowledgement for every in sequence, rest discarded, waste of resources.
2. Blast protocol: Multiple acknowledgements at once, out of sequence discarded
3. Selective repeat: Header os each packet has 2 fields, no. of packets, and this packet's sequence number. No. of packets helps allocate an appropriate buffer, and a bitmap is created, and out of sequence packets are stored in buffer according to sequence in buffer. Message is sent to sender about missing packet numbers, so only those are sent again.

GROUP COMMUNICATION
Three types:
1. One to many
2. Many to one
3. Many to many

ONE-TO-MANY
One sender, many receivers. In this case, we can create groups of the receivers we need, and form a multicast address, which sends to all receivers of the group. Special case of multicast is broadcast addressing, where the system has a set broadcast address, usually 0, where sending info to this sends info to all in network. If packet does not pertain to that node, it is simply discarded.
Groups can be open or closed.
Close: no outside node can send message to all at once. They can send one by one to all in the group, but cannot access group itself. Helpful if it related to an operation that only that group is handling together.
Open: Any outside node can access the entire group. Useful if replicated servers need help.

To remove, or add or any other group function can be implemented by implementing a group server, which manages all requests, but single node of contention can cause reliability failures. A replicated group server adds overhead keeping requests consistent.

A group has a high level address, independent of group constitutes, which calls server. A low level address contains info of constituents and is kept with group server, alongwith a mapping of processes identifiers of each. Low level address is also called multicast address.

In a system which uses broadcast addressing for group address, all groups a have same low level name, the broadcast address.
A system should allow for multicast, broadcast both. If not multicast, then bo=roadcast is used. If not any, then sender has to individually send message to all receivers. Here for n receivers, n messages have to be sent, while earlier only 1 needed to be sent. Latter is helpful for distributed LAN networks (multiple LANs, connected by gateways), and is better than broadcast for those.

SENDING TO RECEIVER

The sender sends content to high level address, which can be the group server's address, and then the kernel contains a list of low level address mapping and process identifiers. The message is the passed onto individual nodes. Each node checks from process identifier list if it contains the process needed, else discards the message.

If the receiver does not have any buffer, it accepts the message if it can or discards it. If it has buffer, it will store the message in buffer.

Two ways for one-to-many communications:

1. Send-to-all: message is sent to all members of group, and buffered by all receivers if cannot accept at the moment.
2. Bulletin board: message is passed through channel and put up on bulletin board. The process which are idle and have the process needed can check up on board and react to it. This solves the problems: 1. Relevance of the message to the receiver, receiver will only take message from board if it has the appropriate process, and 2. Value decided by sender can be taken care of, for example, a message where only the first response is required.

Flexible reliability: messages can require responses to be not all. So limited reliability is also acceptable. There are 4 types:

1. 0-reliable: no response from any receiver required
2. 1-reliable: only 1 response required, message expires after first response
3. M-out-of-n-reliable: m(1<m<n) responses required
4. All-reliable: all receivers need to respond.

Atomic multicast: if a process is sent to multiple receivers, it needs to be accepted by all or none of them. If a process fails, it gets kicked out of group. It can only rejoin after. Message passing system should allow both atomic and non atomic messaging. One way to implement atomic is messages sent with reliability being all-reliable.

Here, kernel sends message to group members, and receives acknowledgements. After timeout, it retransmits to those where acknowledgements have not been received. It continues this until all acknowledgements have been received.

This does not account for sender failure in sending one of them or receiver failure in receiving. One way out of this is a header section that denotes the message is an atomic multicast message.

When the receiver receives one such message, it will first discard on the basis of it being a new message or old. Then, it checks if it is an atomic multicast message, if it is, then it send to all members of multicast. So, all nodes send to all members. However, this is expensive=ve because of the amount of message being sent, and messages should strive to be non-atomic.

GROUP PRIMITIVES

Although it is possible for same semantics to be used while sending message to a single receiver or a group (swap receiver address to group address), it is recommended a separate primitive like send_group to be used. This defines that the sender is going to get a list of process identifier and low level mapping from group server. Send_group being a primitive

separately also can introduce a new header input where sender can define reliability, or one that can define atomicity.

Many-to-one communication: Many senders, one receiver. Here, receiver may be selective or non selective. Selective means receiver defines that messages will only be taken from a single sender. Non-selective means receiver will accept message from a set of senders. As it is not known which sender will send first in non selectiveness, this introduces non determinism in order of message. Guarded command can be used to control non-determinism.

Many-to-many communication: many senders, many receivers.
This is a mix of one-to-many and many-to-one. In one-to-many, receivers send acknowledgement, and so they receive messages in order as sender sends, or it is possible to verify out-of-order packers by sequence numbers. In many-to-one, ordering depends on how messages reach receiver, and ordering will be handled by receiver. So ordering is not a problem.
In many-to-many, messages from different senders reach using different routes taking different times, and ordering becomes issue. If more than one receiver is handling a db, and are sent requests from more than one receiver, the intended order if reversed will lead to race conditions. So ordering is an issue in many to many, and is handled by absolute, consistent and casual ordering.
ABSOLUTE ORDERING
Here, it is assumed that all systems have clocks and all clocks are synchronized. All messages are set with timestamp before departing. The receiver has sliding window of length appropriate of the time taken for message to reach from any node and time limit of some timestamp, and messages in that time frame are put into the queue. Messages after the limit are not with the possibility that some remaining messages before the limit might reach. So messages are received by receiver in the same order as they were sent.

CONSISTENT ORDERING
Clock synchronization is difficult to achieve, and not necessary. Consistent ordering is when messages are sent to receiver in the same order, not necessarily the order in which they left the sender. 2 ways to implement this:

1. One common receiver, called sequencer, is sent all messages, who attaches a sequence number to them. If they are in order, they are dispatched to their destination receivers, otherwise, waited on until messages that fill the gap arrive.

The above method gives single point of failure and reliability, so not practised. Second method:
ABCAST method:
The sender sends a temporary sequence number, larger than any sent so far, attached message to all embers. The members calculate a sequence number by this:
max(F,P) +1+i/N
I, is the member i,
N=total number of members
F=max seq num decided so far
P=max proposed seq num proposed by this member.

This final decided number is sent to the sender by each member, sender selects the largest of them and sets it as new seq num in a commit message. Members then use this num, and send the committed processes with new seq. Num to their processes. The seq num calculation and assignment is a part of runtime, not user process.

CASUAL ORDERING
Basic funda is: messages that can be affected by their sequence are related, and should arrive in order, but messages which are not related to each other or other set of related messages can arrive in any order.
CBCAST algo:
Each member has a vector of size n, (n=num of members), and value of vector[i] is the latest seq num sent by member i to this member.
Then the member increments its own seq num and sends message to receiver.
Receiver upon receiver checks 2 conditions. S[i]=R[i]+1, to ensure that there is no message left, and S[j]>=R[j], j!=i, to check if there are any remaining packets not received from other members.
If both conditions are fulfilled, then message is given to receiver, else it remains in buffer.
This is checked for every message.

RPC
To achieve transparency and hide details of message passing and non local implementation, we use RPC (Remote Procedure Call).
Basic idea: allowing machines to execute procedures on other machines. Process on A calls procedure on B, and calling process on A gets suspended until it gets a result. Procedure executes on B, sends result back to A. No message passing is visible to programmer.
Normal local procedure execution: Call made from main initializes a stack where parameters and local variables are pushed, last one first. Procedure executed, return value pushed onto register and control transferred back to main.
RPC: calling procedure should not know that execution is taking place on other machine and vice versa.
Steps:

1. Client procedure calls client stub in normal way.
2. Client stub calls build message and send to local OS.
3. Local OS passes message to server OS
4. Server OS passes onto server stub
5. Server stub unpacks parameters and send to server
6. Server executes and return answer to server stub
7. Server stub packs answer in message and sends to local OS
8. Local OS ends message to client OS
9. Client OS gives message to client stub
10. Client stub unpacks and gives answer to client procedure.

Calling procedure on client remains blocked until it receives an answer. Server stub, after sending answer, puts out receive message to answer any incoming requests.

PASSING PARAMETERS
Packaging parameters into a message is called parameter marshaling.
Passing parameters by value:
As long as the client and server machines are identical and all parameters and results are scalar types, the model works fine.
If client and server machines are of different types, it could be they store character differently (in ASCII and EBCDIC codes), causing them to interpret characters incorrectly. They may also assign numbering from right or left (big endian and little endian notations). Also, without additional info about what is the data type of each parameter, simply reversing doesn't help, if one parameter is interpreted correctly by both machines and other is not, as all will be reversed the same.

Passing parameters by reference:
We pass address in this, and as client and server uses different address to same bit, we cannot send the address directly. If client knows size of vector to be sent, one way is to copy and send vector entirely, and server will interpret the address to change by its own. If the stubs know which parameter is input and output to the server, mechanism becomes twice efficient. If arrays are input to the server, they need not be copied back, and if they are an output parameter, they need not be sent over in first place.
One method is by passing a pointer to the server stub and generating a special code in server procedure to use pointers, where request may be sent back to client to send referenced data.

Another way of passing by value is sending a deatoed message of value format with the parameters, so they can be unambiguously interpreted.
The client and server also need to agree on the protocol to be used, TCP, UDP, etc.
Implementation of stubs is simple, stubs for same protocol but different procedures only differ in interfaces, which is a collection of procedures that can be called by client and implemented by server. Interfaces are defined in Interface Definition language (IDL) which simplifies client-server applications.

Extended RPC models
IPC is faster than networking where client server models exist in the same system. RPC provides an extended functionality called door for such. A door is a generic name for a procedure in the address space of a server process that can be called by processes colocated with the server. Server first has to register a door with a unique identifier with command create_door. A client calls the door with its unique identifier and passes parameters. The OS then does an upcall to the server process that registered the door. The results after invoking the door are returned to client process through system call door_return.
Benefit: allow use of single mechanism, procedure calls, for communication in a distributed system.

Disadvantage: applications developers still need to be aware whether a call is done within current process (local stack), local to different process on same machine(door) or remote process (RPC).

Asynchronous RPC: in normal RPC, caller process is blacked until return value is given. This causes waste of client process time specially in calls where no return is required, such as transferring money, updating db, etc. For this, asynchronous RPC is allowed where server gives quick acknowledgement before even processing RPC , and client can continue execution.

If a return value is required, but client does not want to wait for it, 2 asynchronous RPC can be used. Client sends RPC, server sends acknowledgement, client continues execution. Server computes, send result back to client, client gives acknowledgment. This is called deferred synchronous RPC.
One way RPC is when client does not wait for acknowledgment, sends RPC and continues execution. However, here client has no way to know if request will be processed if reliability is not guaranteed.

ROI (Remote Object Invocation)
Object encapsulates data, also called state, and that data can only be altered using methods of the object. Methods are available by interface. An object can have multiple interfaces.
A distributed object has the entire object on one system, and interfaces on many.
Proess:
Client binds to distributed object, and so an interface loads on client machine, called proxy (like a client stub). Proxy marshals method invocations into a message, send it to main object, and unmarshals the reply back to client. Incoming invocation are first passed to server stub, also known as skeleton on the server machine where object resides, and performs the same thing server stub does at RPC. object state is not distributed, entire object resides on a single machine. Such object is also called remote object.
Objects can be created at compile time by a language, using class, where object is instance of a class, but they are dependent on the language.
Objects can be created on runtime, where implementation appears as an object whose methods can be invoked from a remote machine. An object adapter is used, which is a wrapper around the implementation to give it appearance of object. Defining objects in terms of interfaces makes wrapping easy. An implementation of interface can then be registered at adapter, which makes interface available for remote invocations.

Persistent objects: continues to exist even if currently not contained n server's address space, can be stored in secondary storage until another invocation call.
Transient object: object ceases to exist once server exits.

Binding client to object: object references can be freely passed between processes on different machines. When a process holds an object reference, it must first bind to the reference before invoking any method. There are 2 ways: 1. Implicit binding: client is offered a simple mechanism

that allows it to directly income methods using only a reference to an object. So object is bound the moment the reference is resolved to an actual object.
2. Explicit binding: a client should call a special function to bind to the object before it can invoke the methods.

Object reference must contain address of machine where object resides, endpoint identifying the server that manages the object, an identifier of object to bind to it.
If server machine crashes and it is given a different endpoint, all references become invalid. For this, a local list is kept in every machine which keeps track of server-to-endpoint, and all local lists update from a global list where servers update.
However with this a server cannot move to another machine without invalidating all references to the object it manages, so we should keep a location server that keeps track of the machine where object's server is currently running.

If the server accepts more than one protocols, it is client's responsibility to get proxy implementation in one of the protocols. We can also create a template like implementation handle, which is a complete implementation of proxy that client can download, install. Then client need not worry if it has an implementation of specific protocol available. It also gives object developer freedom to create object specific proxies.

Invoking an object's methods through its proxy is called RMI (remote method invocation). RMI is given support by specifying object's interface in interface definition language. Using predefined interface definitions is referred to as static invocation. To compose a method invocation at runtime is called dynamic invocation.
Dynamic invocation form: invoke(object,method,input parameters,output parameters)
Dynamic invocation can be used for batch processing service when invocation request can be handed alongwith a time when invocation should be done.
References to remote object and local object is treated differently. When it is a remote object, reference is copied and passed as value parameter, so literally passed by reference. When it is a local object, referred object is copied as whole and passed along with invocation, i.e. passed by value.

Side effect of invoking a method with object reference as parameter is that we may be copying the object.

MESSAGE ORIENTED COMMUNICATION
rMI and RPC assume both side applications are active. MOC assumes all other cases.
Persistence synchronicity: buffer is available at each host and communication server. Eg. mail

1. Persistent communication: it is not necessary for receiving application to be running, and sending application to continue execution as message is stored by communication server.
2. Transient communication: communication server stores message if both sender, receiver are executing, else message is discarded.
3. Asynchronous: sender continues execution after submitting message

4. Synchronous: sander is blocked until message is stored in local buffer of destination host.

Combinations:

1. Persistent asynchronous:
2. Persistent synchronous
3. Transient asynchronous: eg. one way RPC
4. Receipt based transient synchronous:
5. Delivery based transient synchronous: eg. asynchronous RPC
6. Response based transient synchronous: eg. RPC, RMI

Message oriented transient communication:
Eg. berkley sockets

1. Sockets: primitive created by server to create new communication endpoint for particular transport endpoint.
2. Bind: primitive used to server to bind IP address of machine and port number to created socket.
3. Listen
4. Accept: call to this primitive blocks the caller until connection request arrives
5. Connect
6. Send
7. Receive
8. close
   Message passing interface
    1. MPI_bsend
    2. MPI_send
    3. MPI_ssend
    4. MPI_sendrecv

Message oriented persistent communication: messages are forwarded through many communication servers, each application has its own private queue. System wide unique name is used as address. Primitives offered: put, get, poll, notify
Architecture:
Source queue is on the local machine of sender or same LAN. queue managers manage queues, interact with applications and work as routers or relays.
Message brokers
Message nodes convert messages into formats understandable to destination applications. Some info oss is expected.

Distributed system definition:
A distributed system is defined as a set of autonomous computers that appear to the user as a single coherent system. Its components are located on different computers in network and communicate and coordinate by passing messages. Main characteristics:

1. Users should not know of heterogeneity between systems.
2. Internal organization is also hidden
3. Interaction between user and system is consistent and identical
4. Scaling should be allowed
5. Failure handling should be hidden from users and applications
6. System should always be available

Main issues:

1. Concurrency: cooperating applications should run in parallel and coordinate by utilizing the resources at different machines
2. Globally clocks cannot be synchronized
3. An individual component failure should not affect computation.

To offer single system view of distributed system in heterogenous environment, a middleware layer is placed between higher layer that comprises of user and applications and lower layer comprising of OS and communication facilities.

Examples of distributed system

1. Internet
2. Intranet: can connect to internet. Firewalls protect ingoing and outgoing messages
3. Mobile computing: good portability

Issues of Distributed System

1. Heterogeneity: there can be different implementations of internet, different languages in which system is implemented, different data representation types, different data structure, all such should be hidden from user
2. Making resources accessible: remote resources should be available everywhere, and the security risk involved with increase in connectivity should also be dealt with. Unnecessary communication, like spam, should also be dealt with.
3. Distribution transparency: the distributed system should feel like a single cohesive system. Following types of distribution should be hidden: access transparency(dissimilarities of data across different systems), location transparency(no info about location of storage of info), migration transparency(no change in access if info location changed), replication transparency(hides the fact that copies are stores in many places), concurrency transparency(hides sharing of resource between many users), failure transparency(hides failure and recovery), performance transparency(hides system actions taken to improve performance), scaling transparency(hides steps of scaling the system).
4. Scalability: three factors for scalability: 1. Scalable wrt size: centralization causes bottleneck, 2. Geographically scalable: users can faraway may access, but centralization causes a long time to access from faraway parts and is a waste of resources. 3. Administratively scalable: resources can be accessed with authorization everywhere, but conflicts in policies may occur. Scaling techniques are: 1. Hiding communication latencies: implementing asynchronous communication, so no time is wasted in getting

responded (RPC can be implemented) 2. Distribution: DNS server types distribution in different zones. 3. Replication: creating different copies for faster access but consistency and stale copy issues can occur.

5. Openness: processes use interfaces to communicate with each other. An open DS should allow configuring the system out of different components from different developers. Users should be implement their own policy as a component that can be plugged in

TYPES OF DISTRIBUTED SYSTEMS

1. Distributed computing system: a. Cluster computing that consists of similar hardware closely connected through high speed network for high performance computing tasks. Another example is grid computing, where systems have high degree of heterogeneity, and resources from different organizations are brought together for collaboration of a group of people or institutions.
2. Distributed Info system: client server applications, where client send request and server runs the application. Eg. 1. Transaction processing systems that follow ACID properties and nested transactions
3. Enterprise Application Integration: to decouple applications from databases and integrate applications independent from their databases. Eg. RPS, ROI, and message oriented middleware
4. Distributed Pervasive Systems: above systems are stable and nodes are fixed into it. This system has mobile and embedded systems which makes it unstable, where nodes are connected through wireless connections and are battery powered. There is absence of human administrative control. These devices also discover their environment automatically.

DISTRIBUTED SYSTEM MODELS

ARCHITECTURAL MODELS: Concerned about placement of different components and relationship between them.

1. Software layers
2. System architecture: main types of models : 1. Client-server model 2. Services by multiple servers(replication, to improve performance availability and fault tolerance) 3. Proxy servers and caches 4. Peer processes (all processes are considered equal and cooperate with each other to achieve a task)
3. Variations in basic client-server model: 1. mobile code and mobile agent 2. Network computers 3. Clients 4. Mobile devices and spontaneous networking
4. Interface and objects
5. Design requirements: responsiveness (achieved by less software layers and less amount of data transfer), throughput, balancing loads(by migraying jobs to other machine and replicate info across servers), caching and replication, dependability (correctness, security and fault tolerance) and reliability, security and performance.

FUNDAMENTAL MODELS: allow users to be more specific about characteristics, risk of failure and security.

1. Interaction Model: server process will interact with client, and peer process will interact with each other. The factors that affect are: communication performance, ordering of events, machine clocks and timing events. Two variations of interaction models are: 1. Synchronous (lower and upper bounds for each process are known in advance) 2. asynchronous(does not consider timing bounds on speed of process execution, delays in message transmission and drift rate of clock)
2. Failure model: it suggests different ways of failure that may occur and tries to solve the same: 1. Omission failure(failure of process or communication channel- could be process failure or communication failure) 2. Arbitrary failure 3. Timing failure(clock may drift) 4. Masking failure
3. Security model: it provide methods such as: 1. Protecting objects(access rights with each invocation should be provided) 2. Securing process and interactions 3. adversary(server cannot accept request without verifying it is from a legitimate sender) 4. Use of security models (encryptions or access right techniques applied).

HARDWARE CONCEPTS
Machines are divided into 2 groups: multiprocessors(share the same memory) and multicomputers(individual memory space)
Both multicomputers and multiprocessors can be bus-based or switch based

1. Bus based: machines connected by single cable, and share memory by communicating through bus
2. Switch based: machines connected by different writing patterns ad messages are routed by switching switch.

Multi Computers can be homogenous(same technology for interconnection network, all processors similar), heterogenous (different architecture for different machines, connected using different technology).
Multiprocessors: multiple CPUs and memory modules connected through bus. Memory should be coherent. Cache used when overloaded . processors and memory modules connected by crossbar-switch allow multiple access to different locations at once, but access to same location multiple times simultaneously takes time.
Multicomputer: homogenous: nodes connected through high speed networks. Cluster of workstations is an example. Heterogeneous: performance varies at different locations because of different technologies. There is a requirement of software to build distributes applications for heterogeneous multicomputers.

SOFTWARE CONCEPTS

OS can be categorized as : 1. Tightly coupled systems (provides single global view of the resources its manages, eg. DOS), 2. Loosely coupled systems(each computer has its own OS, this system is known as heterogeneous)

Uniprocessor OS

Single CPU. permits users and applications to share different resources , simultaneously several applications executing on same machine gets required resources. OS protects data from one application from another application. In kernel mode, execution of all int=structions are allowed, set of all registers are available. In user mode, there is restriction on memory and register access. If only kernel mode exists then it is a monolithic architecture.

MUltiprocessor OS

Offers multiple processors to access shared memory. Consistency offered by eait and signal semaphore and monitors.

Multicomputer OS

Communication is by message passing. Software layer offers complete software implementation of hared memory

Distributed memory systems (DSMs)

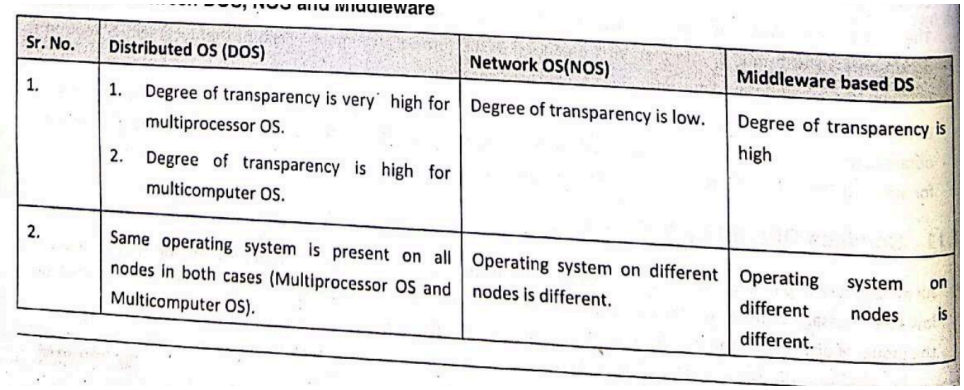
Shared memory on multicomputers offers a virtual shared memory machine. All read only pages are replicated so access performance improves. Before editing any, other copies are invalidated. The OS repetitively transferring page between processors that need a page is called false sharing.

Network Operating system (NOS)

NOS assumes underlying hardware as heterogenous. NOS allows remote login from their terminal to other terminal. File system is implemented by file servers. And user is on a client machine. This system does not allow complete encapsulation as login is still required.

MIDDLEWARE

Middleware is an additional layer that hides heterogeneity of underlying computer networks but get better distribution transparency. Layer is placed between applications and NOS.
Middleware treats everything like a file. Supports file distribution systems. Treats objects as remote entity too, and allows RPC.
Middleware offers high level communication facilities to hide low level message transmitting. Offers naming, persistence, distributed transactions and security.



Sr. No.	Distributed OS (DOS)	Network OS(NOS)	Middleware based DS
1.	1. Degree of transparency is very high for multiprocessor OS. 2. Degree of transparency is high for multicomputer OS.	Degree of transparency is low.	Degree of transparency is high
2.	Same operating system is present on all nodes in both cases (Multiprocessor OS and Multicomputer OS).	Operating system on different nodes is different.	Operating system on different nodes is different.

Sr. No.	Distributed OS (DOS)	Network OS(NOS)	Middleware based DS
3.	1. As multiple processors present in single machine, number of copies of multiprocessor OS is one. 2. As multiple machines are present multiple (n) copies of multicomputer OS are required.	As multiple machines are present multiple (n) copies of NOS are required.	Middleware is installed on the top of NOS on every machine. Hence multiple machines are preset so multiple copies of OS are required.
4.	1. In multiprocessor OS basis for communication is shared memory. Processors share the memory to communicate with each other. 2. In case of multicomputer OS, Many machines communicate with each other by passing the messages. Message passing is basis for communication	In network OS, basis for communication is files.	In middleware based DS, basis for communication is model specific.
5.	1. In multiprocessor OS resource management is global and central. 2. In multicomputer OS resource management is global and distributed	In network OS resource management is per node. Hence it is easy to scale the system.	Resource management is per node
6.	1. In multiprocessor OS there is no scalability as single machine in which multiple processors are presents 2. Multicomputer OS supports for moderate scalability..	Network operating system offers the scalability.	In middleware based distributed system scalability varies.
7.	Both multiprocessor OS and multicomputer OS are not open. These OS are developed to optimize the performance.	Network operating system is openness. Modern operating systems are built with microkernel design.	It is also open.

Client server Model- server implements a particular type of service and client requests to server.
Connectionless protocol can be built between client and server.
Application layering:

1. User interface level- implemented on clients. Input medium GUI
2. Processing level: huge database at backend.
3. Data level-programs that maintain data on which applications perform operation.

Client server architectures

1. Multi Tiered: terminal dependent part on client machine and db on server.
2. Modern: client and server split into logical parts, each part operates on its own share of data set to balance load. Request forwarded in round robin fashion.

Models for Distributed Algorithms

- Topology: Completely connected, Ring, Tree etc.
- Communication: Shared memory / Message passing (reliable? Delay? FIFO/Causal? Broadcast/multicast?)
- Synchronous/asynchronous
- Failure models: Fail stop, Crash, Omission, Byzantine…
- *An algorithm needs to specify the model on which it is supposed to work*

Complexity Measures

- Message complexity: no. of messages
- Communication complexity / Bit Complexity: no. of bits
- Time complexity:
  - For synchronous systems, no. of rounds
  - For asynchronous systems, different definitions are there.