Tulsiramji Gaikwad-Patil College of Engineering & Technology, Nagpur
Department of CSE
8th Sem
Distributed Operating System (BECSE406T)

Models

The different models that are used for building distributed computing systems can be classified as :

1. Minicomputer Model
2. Workstation Model
3. Workstation Server Model
4. Processor Pool Model and
5. Hybrid Model

1.Minicomputer Model

The minicomputer model is a simple extension of the centralized time-sharing system. It consists of a many minicomputers interconnected by a communication network. Each minicomputer has multiple users logged on to it simultaneously. Many interactive terminals are connected to each minicomputer. Each user can have remote access to other minicomputers. The network allows a user to access remote resources that are available on some other machines in the network of distributed system. The minicomputer model was suitable when resource sharing with remote users is desired. Example : Early ARPA net.

2.Workstation Model

This model consists of many workstations interconnected by a communication network. In Some applications it is required to have several workstations located at different locations and communicate with each other with the help of communication network. Where each workstation is has its own disk & serves as a single-user computer.

This model was designed to utilize the power of lightly loaded or idle workstations. For some amount of time a significant proportion of the workstations are idle. If the idle workstations are not utilized properly it results in the waste of large amounts of CPU time.

Therefore, the idea of the workstation model is to interconnect all these workstations by a high-speed LAN so that idle workstations may be used to process jobs of users who are logged onto other workstations & do not have sufficient processing power at their own workstations to get their jobs processed efficiently. Example:Sprite system by Xerox PARC.

3.Workstation–Server Model

In the workstation model there is a network of personal workstations. Where each workstation has its own disk & a local file system (Diskful Workstation).

A workstation with its own local disk is usually called a diskful workstation & a workstation without a local disk is called a diskless workstation. Diskless workstations have become more popular in network environments than diskful workstations, making the workstation-server model more popular than the previous model for building distributed computing systems.

A distributed computing system based on the workstation-server model consists of a few minicomputers & multiple workstations interconnected by a communication network. In this model, a user logs onto a workstation called his or her home workstation. Special

requests for services provided by special servers are sent to a server providing that type of service that performs the user's requested activity.
Therefore, in this model, the user's processes need not migrated to the server machines for getting the work done by those machines. Example:The V-System.
4.Processor–Pool Model:
The processor-pool model is based on the assumption that most of the time a user needs limited amount of computing power but sometimes he may need a very large amount of processing power for a short period. A single processor could not be enough to perform the task.
Therefore, here in processor-pool model the processors are pooled together. The power of multiple processors can be utilized as and when its required by the user.
The pool of processors consists of a large number of microcomputers & minicomputers attached to the network.Each processor in the pool has its own memory to load & run a system program or an application program of the distributed computing system.
In this model no home machine is present & the user does not log onto any machine. This model has better utilization of processing power & greater flexibility.
Example:Amoeba & the Cambridge Distributed Computing System.
5.Hybrid Model: The workstation-server model has a large number of computer users only performing simple interactive tasks &-executing small programs.
In a working environment that has groups of users who often perform jobs needing massive computation, the processor-pool model is more attractive & suitable.
The Advantages of workstation-server & processor-pool models can be obtained by implementing the hybrid model for a distributed system.
The processors in the pool can be allocated dynamically for computations that are too large or require several computers for execution.
The hybrid model gives guaranteed response to interactive jobs allowing them to be more processed in local workstations of the users.

Issues in DOS
1. Openness
The openness of a computer system is the characteristic that determines whether the system can be extended and re-implemented in various ways.The openness of distributed systems is determined primarily by the degree to which new resource-sharing services can be added and be made available for use by a variety of client programs.
2. Security
Many of the information resources that are made available and maintained in distributed systems have a high intrinsic value to their users.Their security is therefore of considerable importance. Security for information resources has three components: confidentiality, integrity, and availability.
3. Scalability
Distributed systems operate effectively and efficiently at many different scales, ranging from a small intranet to the Internet. A system is described as scalable if it will remain effective when there is a significant increase in the number of resources and the number of users.
4. Failure handling

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

5. Concurrency

Both services and applications provide resources that can be shared by clients in a distributed system. There is therefore a possibility that several clients will attempt to access a shared resource at the same time. Object that represents a shared resource in a distributed system must be responsible for ensuring that it operates correctly in a concurrent environment. This applies not only to servers but also to objects in applications. Therefore any programmer who takes an implementation of an object that was not intended for use in a distributed system must do whatever is necessary to make it safe in a concurrent environment.

6. Transparency

Transparency can be achieved at two different levels. Easiest to do is to hide the distribution from the users. The concept of transparency can be applied to several aspects of a distributed system.

a) Location transparency: The users cannot tell where resources are located

b) Migration transparency: Resources can move at will without changing their names

c) Replication transparency: The users cannot tell how many copies exist.

d) Concurrency transparency: Multiple users can share resources automatically.

e) Parallelism transparency: Activities can happen in parallel without users knowing.

7. Quality of service

Once users are provided with the functionality that they require of a service, such as the file service in a distributed system, we can go on to ask about the quality of the service provided. The main nonfunctional properties of systems that affect the quality of the service experienced by clients and users are reliability, security and performance. Adaptability to meet changing system configurations and resource availability has been recognized as a further important aspect of service quality.

8. Reliability

One of the original goals of building distributed systems was to make them more reliable than single-processor systems. The idea is that if a machine goes down, some other machine takes over the job. A highly reliable system must be highly available, but that is not enough. Data entrusted to the system must not be lost or garbled in any way, and if files are stored redundantly on multiple servers, all the copies must be kept consistent. In general, the more copies that are kept, the better the availability, but the greater the chance that they will be inconsistent, especially if updates are frequent.

9. Performance

Always the hidden data in the background is the issue of performance. Building a transparent, flexible, reliable distributed system, more important lies in its performance. In particular, when running a particular application on a distributed system, it should not be appreciably worse than running the same application on a single processor. Unfortunately, achieving this is easier said than done.

Lamport-Vector Logical Clocks

The concept of time is fundamental to our way of thinking about ordering of events in a system. Since physical clocks in a distributed system can drift among each other, it will be very difficult to enforce total ordering among all events in a distributed system across a set of processes.

As a first step, let's try to move away from a physical clock for measuring order in a distributed system. Instead let's try to create a logical clock that assigns an unique id to events in a distributed system, such that it captures the casual order among events in a distributed system. Let's define the distributed system as set of processes. Each process execution is modeled as a sequence of events. The processes communicate with each other via messages.

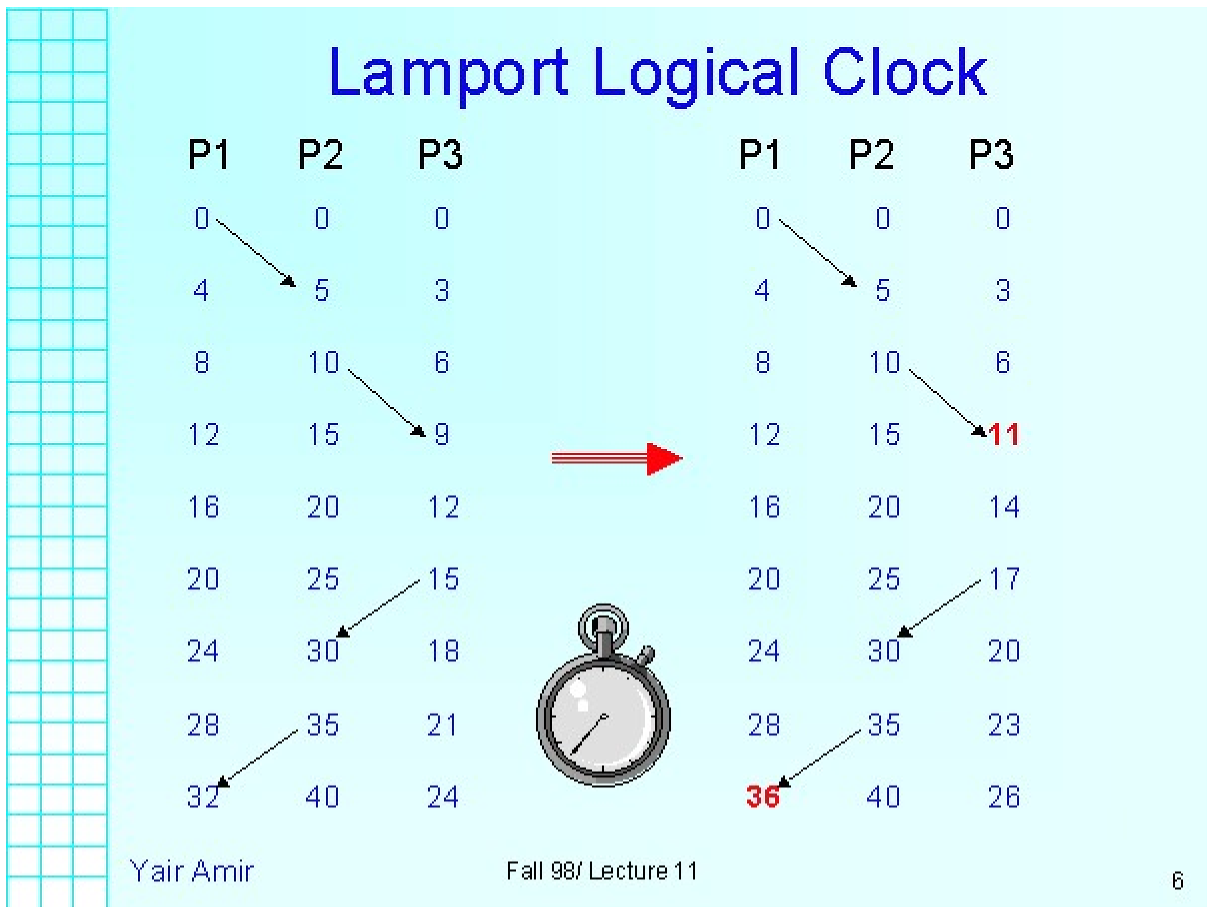
The happens before relation (<) is defined as follows.

1. If a, b are events in the same process, if a occurs before b, then a < b
2. if a denotes sending of message in process-a and b denotes receipt of message in process-b then a < b
3. The relation is transitive, a < b and b < c => a < c

To capture the happens before relation, Lamport introduced a form of clock by name Lamport clocks.

Lamport clocks are defined as follows.

1. It's a counter within a single process that increments from 1. Events within a process is assigned an unique id based on the counter value. Thus all compute and messaging events within a process are assigned an unique value
2. When sending a message from process-a, we set C(a) = C(a) + 1, then pass on C(a) as part of the message.
3. On recipient of message in process-b we set C(b) = max(C(b) + 1, C(a))



Lamport Logical Clock

P1	P2	P3
0	0	0
4	5	3
8	10	6
12	15	9
16	20	12
20	25	15
24	30	18
28	35	21
32	40	24

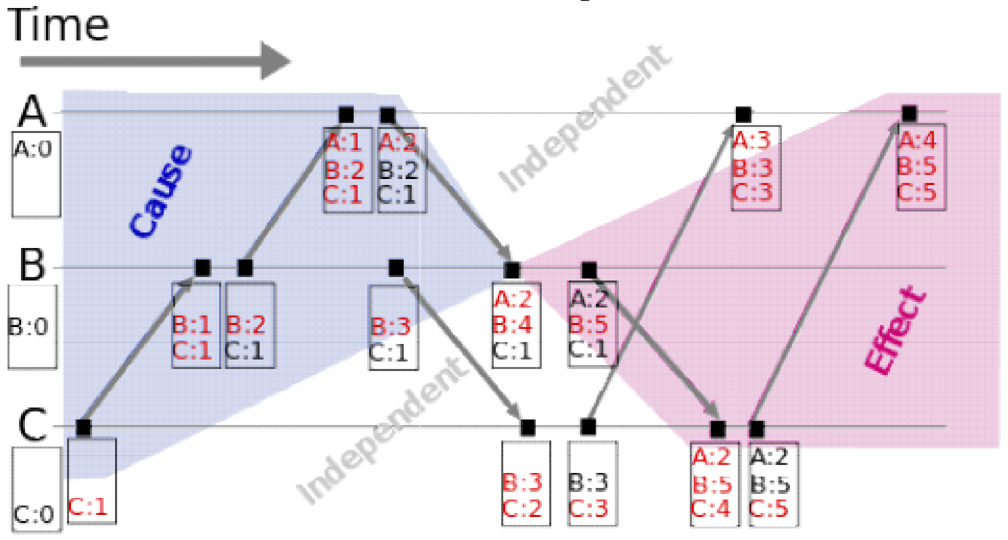
P1	P2	P3
0	0	0
4	5	3
8	10	6
12	15	11
16	20	14
20	25	17
24	30	20
28	35	23
36	40	26

Yair Amir Fall 98/ Lecture 11 6

Lamport clocks can guarantee that if a < b then C(a) < C(b). However it can’t guarantee, that if C(a) < C(b) then event a happened before b.

A casuality relation states that if an event e1 possibly influences the generation of event e2, then e1 < e2. What’s better way to establish such a relation than relying on message passing among the processes. Message passing establishes a happens before order on events based on the communication pattern. For example if a process receives a message as event A, processes the message as B and sends out a new message to another process as event C then it’s assumed event A influenced the generation of event C and hence A & C are casually related, A < C.

Then comes the vector clocks into this picture.



In a system with N processes, each process keeps a vector timestamp TS[N]

1. In Process i,
a. TS[j] is logical time of process j as process i knows about it.
b. TS[i] is the lamport clock of process i.
2. When a new event is generated, TS[i] = TS[i] + 1
3. When sending a message we copy the vector clock to the message
4. On recipient of message with vector timestamp MTS we set the TS of process i as,
TS[k] = max(TS[k], MTS[k]) for k = 1 to N
The happened before ordering in vector clocks is defined as follows.
e1 < e2 iff TS(e1)[k] <= TS(e2)[k] for k = 1 to N
** atleast one of the value of indices of e1 should be less than e2's value*
The timestamp of an event would tell us what all events among all processes would have influenced the generation of that particular event. So if e1 < e2, then e2 would have witnessed all the events that has been witnessed by e1. With vector clocks we can ascertain if TS(e1) < TS(e2) then e1 < e2.

Causal Ordering of Messages

Explanation of Causal Ordering of Messages:
The purpose of causal ordering of messages is to insure that the same causal relationship for the "message send" events correspond with "message receive" events. (i.e. All the messages are processed in order that they were created.)

Birman-Schiper-Stephenson Protocol
There are three basic principles to this algorithm:

1. All messages are time stamped by the sending process.
   [Note: This time is separate from the global time talked about in the previous sections. Instead each element of the vector corresponds to the number of messages sent (including this one) to other processes.]
2. A message can not be delivered until:
   - All the messages before this one have been delivered locally.
   - All the other messages that have been sent out from the original processs has been accounted as delivered at the receiving process.
3. When a message is delivered, the clock is updated.

This protocol requires that the processes communicate through broadcast messages since this would ensure that only one message could be received at any one time (thus concurrently timestamped messages can be ordered).
[Note: There may be other reasons that broadcast messages are required.]

Schiper-Eggli-Sandoz Protocol

Instead of maintaining a vector clock based on the number of messages sent to each process, the vector clock for this protocol can increment at any rate it would like to and has no additional meaning related to the number of messages currently outstanding.

Sending a message:

1. All messages are timestamped and sent out with a list of all the timestamps of messages sent to other processes.
2. Locally store the timestamp that the message was sent with.

Receiving a message:

- A message cannot be delivered if there is a message mentioned in the list of timestamps that predates this one.
- Otherwise, a message can be delivered, performing the following steps:
    1. Merge in the list of timestamps from the message:
        - Add knowledge of messages destined for other processes to our list of processes if we didn't know about any other messages destined for one already.
        - If the new list has a timestamp greater than one we already had stored, update our timestamp to match.
    2. Update the local logical clock.
    3. Check all the local buffered messages to see if they can now be delivered.

Global state recording Chandy Lamport Algorithm

The assumptions of the algorithm are as follows:

- There are no failures and all messages arrive intact and only once
- The communication channels are unidirectional and FIFO ordered
- There is a communication path between any two processes in the system
- Any process may initiate the snapshot algorithm
- The snapshot algorithm does not interfere with the normal execution of the processes
- Each process in the system records its local state and the state of its incoming channels

The algorithm works using marker messages. Each process that wants to initiate a snapshot records its local state and sends a marker on each of its outgoing channels. All the other processes, upon receiving a marker, record their local state, the state of the channel from which the marker just came as empty, and send marker messages on all of their outgoing channels. If a process receives a marker after having recorded its local state, it records the state of the incoming channel from which the marker came as carrying all the messages received since it first recorded its local state.

Some of the assumptions of the algorithm can be facilitated using a more reliable communication protocol such as TCP/IP. The algorithm can be adapted so that there could be multiple snapshots occurring simultaneously.

The Chandy-Lamport algorithm works like this:

1. The observer process (the process taking a snapshot):
    1. Saves its own local state
    2. Sends a snapshot request message bearing a snapshot token to all other processes
2. A process receiving the snapshot token *for the first time* on *any* message:
    1. Sends the observer process its own saved state

   2. Attaches the snapshot token to all subsequent messages (to help propagate the snapshot token)
3. When a process that has already received the snapshot token receives a message that does not bear the snapshot token, this process will forward that message to the observer process. This message was obviously sent before the snapshot "cut off" (as it does not bear a snapshot token and thus must have come from before the snapshot token was sent out) and needs to be included in the snapshot.

From this, the observer builds up a complete snapshot: a saved state for each process and all messages "in the ether" are saved.

Cuts in Distributed Computing

There are two formal models of distributed systems: synchronous and asynchronous.
Synchronous distributed systems have the following characteristics:

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

Asynchronous distributed systems, in contrast, guarantee no bounds on process execution speeds, message transmission delays, or clock drift rates. Most distributed systems we discuss, including the Internet, are asynchronous systems.
Generally, timing is a challenging an important issue in building distributed systems. Consider a couple of examples:

- Suppose we want to build a distributed system to track the battery usage of a bunch of laptop computers and we'd like to record the percentage of the battery each has remaining at exactly 2pm.
- Suppose we want to build a distributed, real time auction and we want to know which of two bidders submitted their bid first.
- Suppose we want to debug a distributed system and we want to know whether variable x_1 in process p_1 ever differs by more than 50 from variable x_2 in process p_2.

In the first example, we would really like to synchronize the clocks of all participating computers and take a measurement of absolute time. In the second and third examples, knowing the absolute time is not as crucial as knowing the order in which events occurred.

Clock Synchronization

Every computer has a physical clock that counts oscillations of a crystal. This hardware clock is used by the computer's software clock to track the current time. However, the hardware clock is subject to *drift*-- the clock's frequency varies and the time becomes inaccurate. As a result, any two clocks are likely to be slightly different at any given time. The difference between two clocks is called their *skew*.
There are several methods for synchronizing physical clocks. *External synchronization* means that all computers in the system are synchronized with an external source of time (e.g., a UTC signal). *Internal synchronization* means that all computers in the system are synchronized with one another, but the time is not necessarily accurate with respect to UTC.
In a synchronous system, synchronization is straightforward since upper and lower bounds on the transmission time for a message are known. One process sends a message to another

process indicating its current time, t. The second process sets its clock to $t + (max+min)/2$ where max and min are the upper and lower bounds for the message transmission time respectively. This guarantees that the skew is at most $(max-min)/2$.
Cristian's method for synchronization in asynchronous systems is similar, but does not rely on a predetermined max and min transmission time. Instead, a process p_1 requests the current time from another process p_2 and measures the RTT (T_{round}) of the request/reply. Whenp_1 receives the time t from p_2 it sets its time to $t + T_{round}/2$.
The Berkeley algorithm, developed for collections of computers running Berkeley UNIX, is an internal synchronization mechanism that works by electing a master to coordinate the synchronization. The master polls the other computers (called slaves) for their times, computes an average, and tells each computer by how much it should adjust its clock.
The Network Time Protocol (NTP) is yet another method for synchronizing clocks that uses a hierarchical architecture where he top level of the hierarchy (stratum 1) are servers connected to a UTC time source.

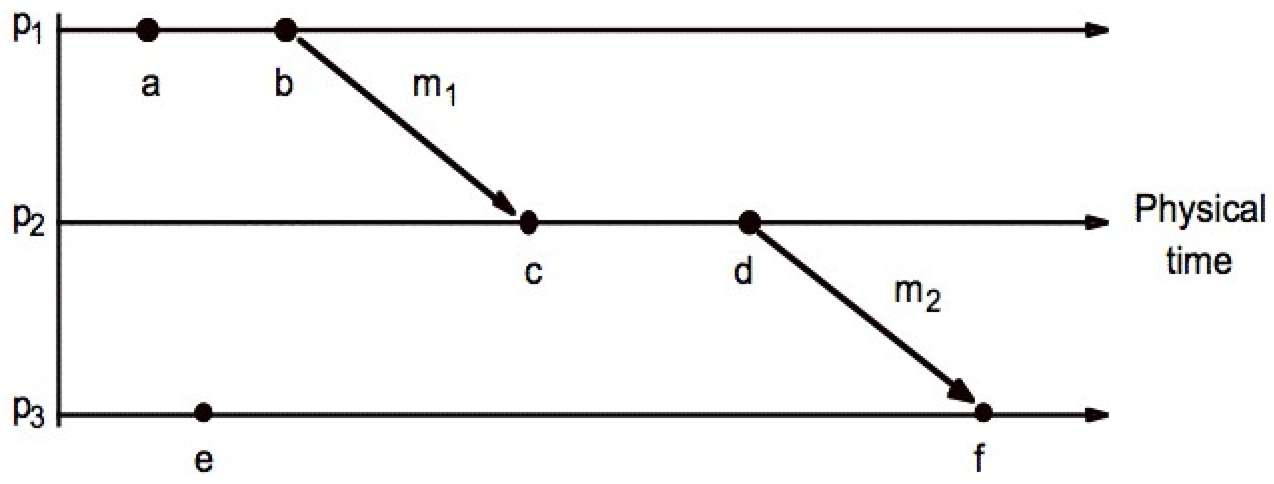
Logical Time

Physical time cannot be perfectly synchronized. Logical time provides a mechanism to define the *causal order* in which events occur at different processes. The ordering is based on the following:

- Two events occurring at the same process happen in the order in which they are observed by the process.
- If a message is sent from one process to another, the sending of the message happened before the receiving of the message.
- If e occurred before e' and e' occurred before e'' then e occurred before e''.

"Lamport called the partial ordering obtained by generalizing these two relationships the *happened-before* relation." ($\rightarrow$)

Figure 11.5
Events occurring at three processes

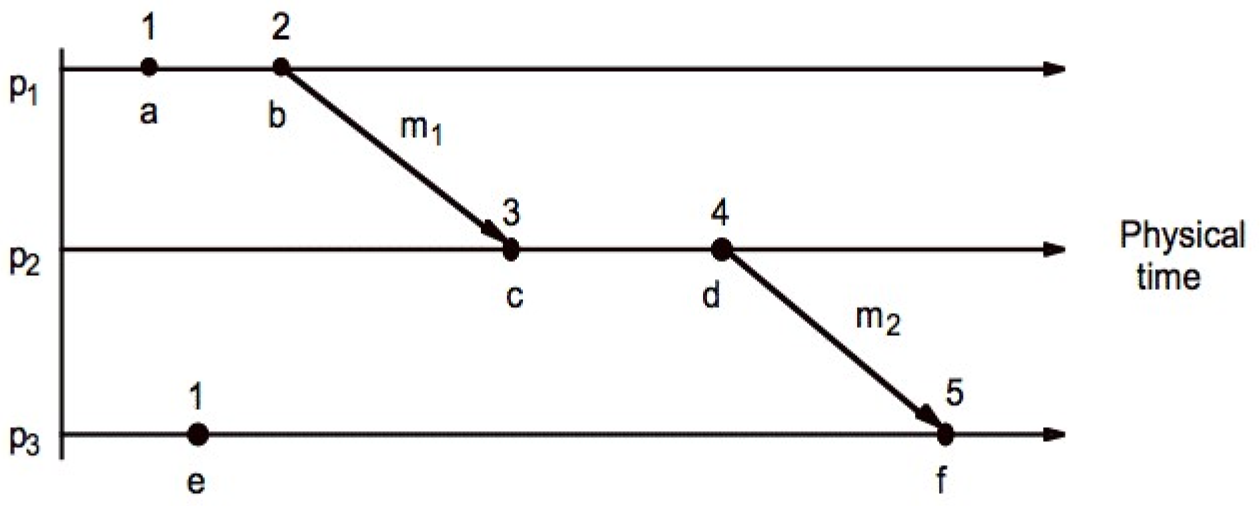


In the figure, a → b and c → d . Also, b → c and d → f , which means that a → f . However, we cannot say that a → e or vice versa; we say that they are *concurrent* ($a \| e$).
A Lamport logical clock is a monotonically increasing software counter, whose value need bear no particular relationship to any physical clock. Each process p_i keeps its own logical clock, L_i, which it uses to apply so-called *Lamport timestamps* to events.
Lamport clocks work as follows:

- LC1: L_i is incremented before each event is issued at p_i.
- LC2:
    -
        - When a process p_i sends a message m, it piggybacks on m the value $t = L_i$.
        - On receiving (m, t), a process p_j computes $L_j := max(L_j, t)$ and then applies LC1 before timestamping the event *receive(m)*.

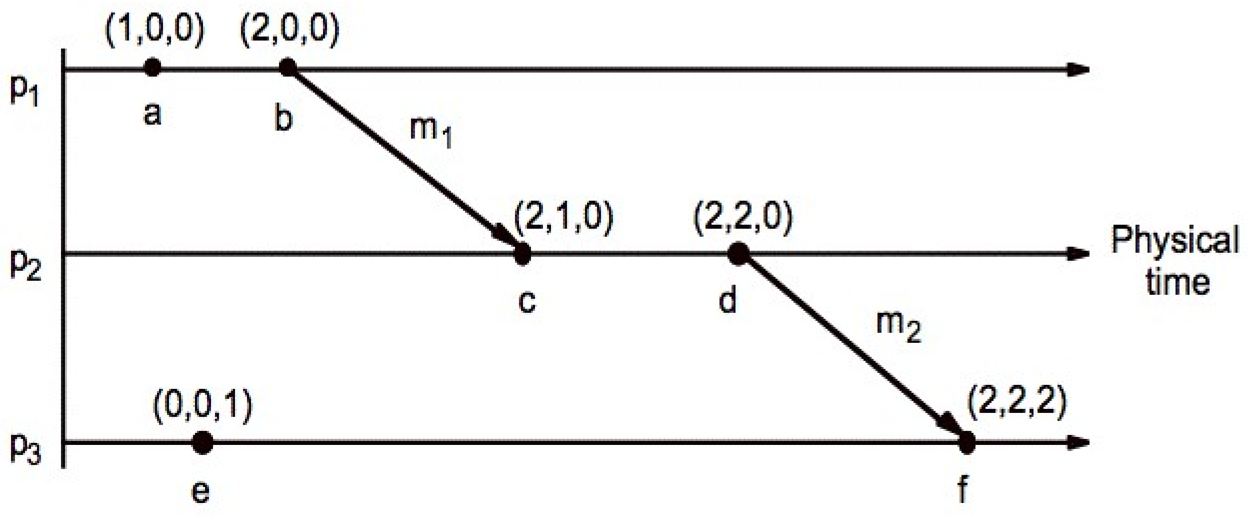
An example is shown below:

Figure 11.6
Lamport timestamps for the events shown in Figure 11.5



If $e \rightarrow e'$ then L(e) < L(e'), but the converse is not true. Vector clocks address this problem. "A vector clock for a system of N processes is an array of N integers." Vector clocks are updated as follows:
VC1: Initially, $V_i[j] = 0$ for i, j = 1, 2, ..., N
VC2: Just before p_i timestamps an event, it sets $V_i[i]:=V_i[i]+1$.
VC3: p_i includes the value $t = V_i$ in every message it sends.
VC4: When p_i receives a timestamp t in a message, it sets $V_i[j]:=\max(V_i[j], t[j])$, for 1, 2, ...N. Taking the componentwise maximum of two vector timestamps in this way is known as a merge operation.
An example is shown below:

Figure 11.7
Vector timestamps for the events shown in Figure 11.5



Instructor's Guide for Coulouris, Dollimore and Kindberg Distributed Systems: Concepts and Design Edn. 4
© Pearson Education 2005

Vector timestamps are compared as follows:
V=V' iff V[j] = V'[j] for j = 1, 2, ..., N
V <= V' iff V[j] <=V'[j] for j = 1, 2, ..., N
V < V' iff V <= V' and V != V'
If e → e ' then V(e) < V(e') and if V(e) < V(e') then e → e ' .

Global States

It is often desirable to determine whether a particular property is true of a distributed system as it executes. We'd like to use logical time to construct a global view of the system state and determine whether a particular property is true. A few examples are as follows:

- Distributed garbage collection: Are there references to an object anywhere in the system? References may exist at the local process, at another process, or in the communication channel.
- Distributed deadlock detection: Is there a cycle in the graph of the "waits for" relationship between processes?
- Distributed termination detection: Has a distributed algorithm terminated?
- Distributed debugging: Example: given two processes p_1 and p_2 with variables x_1 and x_2 respectively, can we determine whether the condition $|x_1-x_2| > \delta$ is ever true.

In general, this problem is referred to as *Global Predicate Evaluation*. "A global state predicate is a function that maps from the set of global state of processes in the system ρ to {True, False}."

- Safety - a predicate always evaluates to false. A given undesirable property (e.g., deadlock) never occurs.
- Liveness - a predicate eventually evaluates to true. A given desirable property (e.g., termination) eventually occurs.

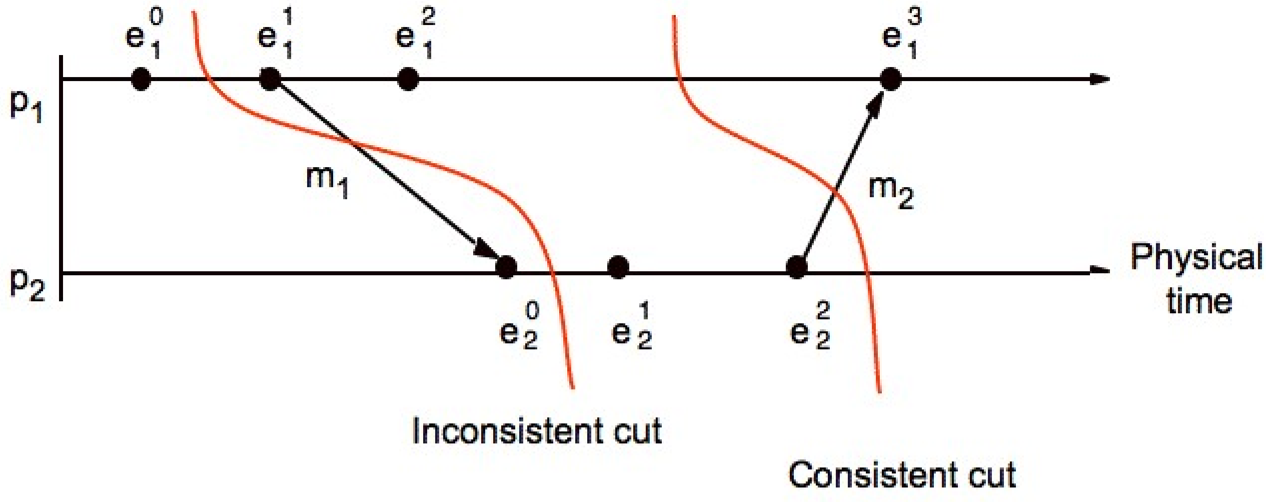
Cuts of a Distributed Computation

Because physical time cannot be perfectly synchronized in a distributed system it is not possible to gather the global state of the system at a particular time. Cuts provide the ability to "assemble a meaningful global state from local states recorded at different times".

Definitions:

- ρ is a system of N processes p_i (i = 1, 2, ..., N)
- history(p_i) = h_i = < e i 0 , e i 1 ,...>
- h i k =< e i 0 , e i 1 ,..., e i k > - a finite prefix of the process's history
- s i k is the state of the process p_i immediately before the kth event occurs
- All processes record sending and receiving of messages. If a process p_i records the sending of message m to process p_j and p_j has not recorded receipt of the message, then m is part of the state of the channel between p_i and p_j.
- A *global history* of ρ is the union of the individual process histories: $H = h_0 \cup h_1 \cup h_2 \cup ... \cup h_{N-1}$
- A *global state* can be formed by taking the set of states of the individual processes: $S = (s_1, s_2, ..., s_N)$
- A *cut* of the system's execution is a subset of its global history that is a union of prefixes of process histories (see figure below).
- The *frontier* of the cut is the last state in each process.
- A cut is *consistent* if, for all events *e* and *e'*:
  - $(e \in C \text{ and } e' \rightarrow e) \Rightarrow e' \in C$
- A *consistent global state* is one that corresponds to a consistent cut.

Figure 11.9
Cuts



Instructor's Guide for Coulouris, Dollimore and Kindberg Distributed Systems: Concepts and Design Edn. 4
© Pearson Education 2005

Distributed Debugging

To further examine how you might produce consistent cuts, we'll use the distributed debugging example. Recall that we have several processes, each with a variable x_i. "The safety condition required in this example is $|x_i-x_j| <= \delta$ (i, j = 1, 2, ..., N)."

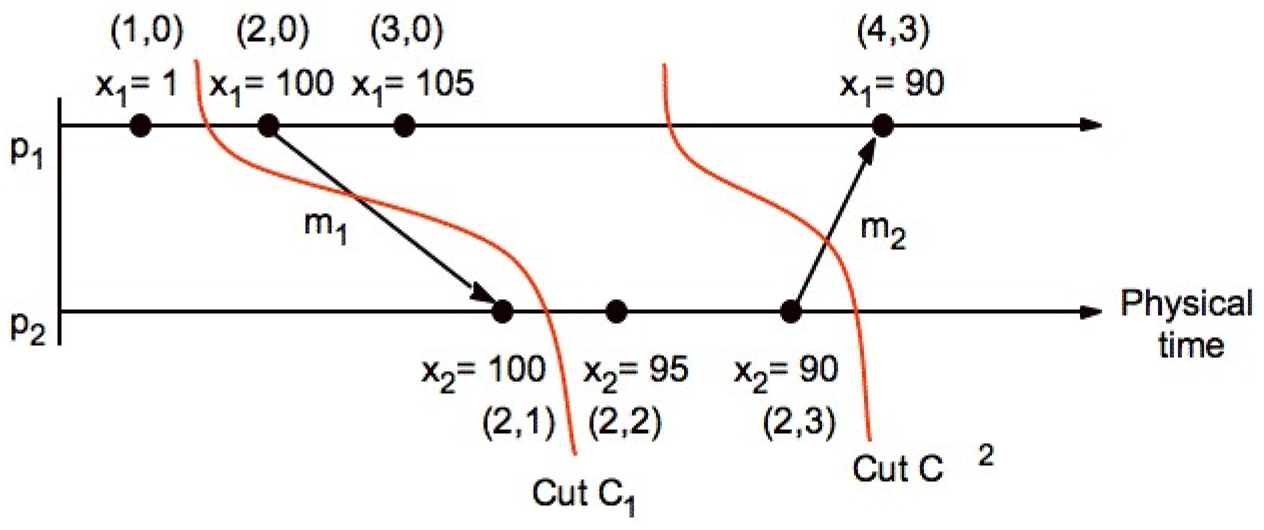
The algorithm we'll discuss is a centralized algorithm that determines post hoc whether the safety condition was ever violated. The processes in the system, p_1, p_2, ..., p_N, send their states to a passive monitoring process, p_0. p_0 is not part of the system. Based on the states collected, p_0 can evaluate the safety condition.

Collecting the state: The processes send their initial state to a monitoring process and send updates whenever relevant state changes, in this case the variable x_i. In addition, the processes need only send the value of x_i and a vector timestamp. The monitoring process maintains a an ordered queue (by the vector timestamps) for each process where it stores the state messages. It can then create consistent global states which it uses to evaluate the safety condition.

Let S = (s_1, s_2, ..., S_N) be a global state drawn from the state messages that the monitor process has received. Let $V(s_i)$ be the vector timestamp of the state s_i received from p_i. Then it can be shown that S is a consistent global state if and only if:

$V(s_i)[i] >= V(s_j)[i]$ for i, j = 1, 2, ..., N

Figure 11.14
Vector timestamps and variable values for the execution of Figure 11.9



Termination Detection
Huang's algorithm is an algorithm for detecting termination in a distributed system. The algorithm was proposed by Shing-Tsaan Huang in 1989 in the Journal of Computers.
Termination detection

The basis of termination detection is in the concept of a distributed system process' state. At any time, a process in a distributed system is either in an active state or in an idle state. An active process may become idle at any time but an idle process may only become active again upon receiving a computational message.
Termination occurs when all processes in the distributed system become idle and there are no computational messages in transit.
Algorithm
Huang's algorithm can be described by the following:

- Initially all processes are idle.
- A distributed task is started by a process sending a computational message to another process. This initial process to send the message is the "controlling agent".
  - The initial weight of the controlling agent is {\displaystyle w} (usually 1).
- The following rules are applied throughout the computation:

- A process sending a message splits its current weight between itself and the message.
- A process receiving a message adds the weight of the message to itself.
- Upon becoming idle, a process sends a message containing its entire weight back to the controlling agent and it goes idle.
- Termination occurs when the controlling agent has a weight of {\displaystyle w} and is in the idle state.

Some weaknesses to Huang's algorithm are that it is unable to detect termination if a message is lost in transit or if a process fails while in an active state.

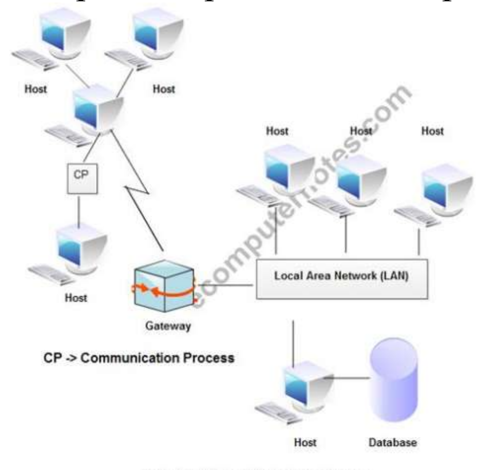
What does Distributed System mean? Explain the concept of DOS?

A distributed system is a network that consists of autonomous computers that are connected using a distribution middleware. They help in sharing different resources and capabilities to provide users with a single and integrated coherent network.

Concept of Distributed Operating System

Distributed Operating System is a model where distributed applications are running on multiple computers linked by communications. A distributed operating system is an extension of the network operating system that supports higher levels of communication and integration of the machines on the network.

This system looks to its users like an ordinary centralized operating system but runs on multiple, independent central processing units (CPUs).



A Typical View of Distributed System

These systems are referred as loosely coupled systems where each processor has its own local memory and processors communicate with one another through various communication lines, such as high speed buses or telephone lines. By loosely coupled systems, we mean that such computers possess no hardware connections at the CPU - memory bus level, but are connected by external interfaces that run under the control of software.

The Distributed Os involves a collection of autonomous computer systems, capable of communicating and cooperating with each other through a LAN / WAN. A Distributed Os provides a virtual machine abstraction to its users and wide sharing of resources like as computational capacity, I/O and files etc.

The structure shown in fig contains a set of individual computer systems and workstations connected via communication systems, but by this structure we can not say it is a distributed system because it is the software, not the hardware, that determines whether a system is distributed or not.
The users of a true distributed system should not know, on which machine their programs are running and where their files are stored. LOCUS and MICROS are the best examples of distributed operating systems.
Using LOCUS operating system it was possible to access local and distant files in uniform manner. This feature enabled a user to log on any node of the network and to utilize the resources in a network without the reference of his/her location. MICROS provided sharing of resources in an automatic manner. The jobs were assigned to different nodes of the whole system to balance the load on different nodes.
Below given are some of the examples of distributed operating systems:
l. IRIX operating system; is the implementation of UNIX System V, Release 3 for Silicon Graphics multiprocessor workstations.
2. DYNIX operating system running on Sequent Symmetry multiprocessor computers.
3. AIX operating system for IBM RS/6000 computers.
4. Solaris operating system for SUN multiprocessor workstations.
5. Mach/OS is a multithreading and multitasking UNIX compatible operating system;
6. OSF/1 operating system developed by Open Foundation Software: UNIX compatible.
Distributed systems provide the following advantages:
1 Sharing of resources.
2 Reliability.
3 Communication.
4 Computation speedup
Distributed systems are potentially more reliable than a central system because if a system has only one instance of some critical component, such as a CPU, disk, or network interface, and that component fails, the system will go down. When there are multiple instances, the system may be able to continue in spite of occasional failures. In addition to hardware failures, one can also consider software failures. Distributed systems allow both hardware and software errors to be dealt with.
A distributed system is a set of computers that communicate and collaborate each other using software and hardware interconnecting components. Multiprocessors (MIMD computers using shared memory architecture), multicomputers connected through static or dynamic interconnection networks (MIMD computers using message passing architecture) and workstations connected through local area network are examples of such distributed systems.
A distributed system is managed by a distributed operating system. A distributed operating system manages the system shared resources used by multiple processes, the process scheduling activity (how processes are allocating on available processors), the communication and synchronization between running processes and so on. The software for parallel computers could be also tightly coupled or loosely coupled. The loosely coupled software allows computers and users of a distributed system to be independent each other but having a limited possibility to cooperate. An example of such a system is a group of computers connected through a local network. Every computer has its own

memory, hard disk. There are some shared resources such files and printers. If the interconnection network broke down, individual computers could be used but without some features like printing to a non-local printer.

Features of Distributed Systems:

Basic features of distributed system

1. 1. Basic featuresof distributedsystem Transparency Transparency or single-system image refers to the ability of an application to treat the system on which it operates without regard to whether it is distributed and without regard to hardware or other implementation details. Many areas of a system can benefit from transparency, including access, location, performance, naming, and migration. The consideration of transparency directly affects decision making in every aspect of design of a distributed operating system. Transparency can impose certain requirements and/or restrictions on other design considerations. Inter-process communication Inter-Process Communication (IPC) is the implementation of general communication, process interaction, and dataflow between threads and/or processes both within a node, and between nodes in a distributed OS. The intra-node and inter-node communication requirements drive low- level IPC design, which is the typical approach to implementing communication functions that support transparency. In this sense, Inter process communication is the greatest underlying concept in the low-level design considerations of a distributed operating system. Process management Process management provides policies and mechanisms for effective and efficient sharing of resources between distributed processes. These policies and mechanisms support operations involving the allocation and de-allocation of processes and ports to processors, as well as mechanisms to run, suspend, migrate, halt, or resume process execution. While these resources and operations can be either local or remote with respect to each other, the distributed OS maintains state and synchronization over all processes in the system. As an example, load balancing is a common process management function. Load balancing monitors node performance and is responsible for shifting activity across nodes when the system is out of balance. One load balancing function is picking a process to move. The kernel may employ several selection mechanisms, including priority-based choice. This mechanism chooses a process based on a policy such as 'newest request'. The system implements the policy Resource management Systems resources such as memory, files, devices, etc. are distributed throughout a system, and at any given moment, any of these nodes may have light to idle workloads. Load sharing and load balancing require many policy-oriented decisions, ranging from finding idle CPUs, when to move, and which to move. Many algorithms exist to aid in these decisions; however, this calls for a second level of decision making policy in choosing the algorithm best suited for the scenario, and the conditions surrounding the scenario. Reliability
2. 2. or a process must establish exclusive access to a shared resource. Improper synchronization can lead to multiple failure modes including loss of atomicity, consistency, isolation and durability, deadlock, livelock and loss of serializability.[citation needed] Flexibility Flexibility in a distributed operating

system is enhanced through the modular and characteristics of the distributed OS, and by providing a richer set of higher-level services. The completeness and quality of the kernel/microkernel simplifies implementation of such services, and potentially enables service providers greater choice of providers for such services.[citation needed]• one or more processes must wait for an asynchronous condition in order to continue, • one or more processes must synchronize at a given point for one or more other processes to continue, •Distributed OS can provide the necessary resources and services to achieve high levels of reliability, or the ability to prevent and/or recover from errors. Faults are physical or logical defects that can cause errors in the system. For a system to be reliable, it must somehow overcome the adverse effects of faults. The primary methods for dealing with faults include fault avoidance, fault tolerance, and fault detection and recovery. Fault avoidance covers proactive measures taken to minimize the occurrence of faults. These proactive measures can be in the form of transactions, replication and backups. Fault tolerance is the ability of a system to continue operation in the presence of a fault. In the event, the system should detect and recover full functionality. In any event, any actions taken should make every effort to preserve the single system image. Availability Availability is the fraction of time during which the system can respond to requests. Performance Many benchmark metrics quantify performance; throughput, response time, job completions per unit time, system utilization, etc. With respect to a distributed OS, performance most often distills to a balance between process parallelism and IPC.[citation needed] Managing the task granularity of parallelism in a sensible relation to the messages required for support is extremely effective.[citation needed] Also, identifying when it is more beneficial to migrate a process to its data, rather than copy the data, is effective as well.[citation needed] Synchronization Cooperating concurrent processes have an inherent need for synchronization, which ensures that changes happen in a correct and predictable fashion. Three basic situations that define the scope of this need:

3. 3. Operating systems are there from the very first computer generation. Operating systems keep evolving over the period of time. Following are few of the important types of operating system which are most commonly used. Batch operating system The users of batch operating system do not interact with the computer directly. Each user prepares his job on an off-line device Reduces CPU idle time. Disadvantages of Timesharing operating systems are following.• Avoids duplication of software. • Provide advantage of quick response. • Difficult to provide the desired priority. Time-sharing operating systems Time sharing is a technique which enables many people, located at various terminals, to use a particular computer system at the same time. Time-sharing or multitasking is a logical extension of multiprogramming. Processor's time which is shared among multiple users simultaneously is termed as time-sharing. The main difference between Multiprogrammed Batch Systems and Time-Sharing Systems is that in case of Multiprogrammed batch systems, objective is to maximize processor use, whereas in Time-Sharing Systems objective is to minimize response time. Multiple jobs are executed by the CPU by switching between them, but the switches occur so frequently. Thus, the user can receives an immediate response. For example, in a transaction processing, processor execute

each user program in a short burst or quantum of computation. That is if n users are present, each user can get time quantum. When the user submits the command, the response time is in few seconds at most. Operating system uses CPU scheduling and multiprogramming to provide each user with a small portion of a time. Computer systems that were designed primarily as batch systems have been modified to time-sharing systems. Advantages of Timesharing operating systems are following • CPU is often idle, because the speeds of the mechanical I/O devices is slower than CPU. • Lack of interaction between the user and job. •like punch cards and submits it to the computer operator. To speed up processing, jobs with similar needs are batched together and run as a group. Thus, the programmers left their programs with the operator. The operator then sorts programs into batches with similar requirements. The problems with Batch Systems are following.

4. 4. Dependency on a central location for most operations.• High cost of buying and running a server. • Remote access to servers is possible from different locations and types of systems. The disadvantages of network operating systems are following. • Upgrades to new technologies and hardwares can be easily integrated into the system. • Security is server managed. • Centralized servers are highly stable. • Reduction of delays in data processing. Network operating System Network Operating System runs on a server and and provides server the capability to manage data, users, groups, security, applications, and other networking functions. The primary purpose of the network operating system is to allow shared file and printer access among multiple computers in a network, typically a local area network (LAN), a private network or to other networks. Examples of network operating systems are Microsoft Windows Server 2003, Microsoft Windows Server 2008, UNIX, Linux, Mac OS X, Novell NetWare, and BSD. The advantages of network operating systems are following. • Reduction of the load on the host computer. • Better service to the customers. • If one site fails in a distributed system, the remaining sites can potentially continue operating. • Speedup the exchange of data with one another via electronic mail. • With resource sharing facility user at one site may be able to use the resources available at another. • Problem of data communication. Distributed operating System Distributed systems use multiple central processors to serve multiple real time application and multiple users. Data processing jobs are distributed among the processors accordingly to which one can perform each job most efficiently. The processors communicate with one another through various communication lines (such as high-speed buses or telephone lines). These are referred as loosely coupled systems or distributed systems. Processors in a distributed system may vary in size and function. These processors are referred as sites, nodes, computers and so on. The advantages of distributed systems are following. • Question of security and integrity of user programs and data. • Problem of reliability. •
5. 5. Regular maintenance and updates are required. Real Time operating System Real time system is defines as a data processing system in which the time interval required to process and respond to inputs is so small that it controls the environment. Real time processing is always on line whereas on line system need not be real time.

The time taken by the system to respond to an input and display of required updated information is termed as response time. So in this method response time is very less as compared to the online processing. Real-time systems are used when there are rigid time requirements on the operation of a processor or the flow of data and real-time systems can be used as a control device in a dedicated application. Real-time operating system has well-defined, fixed time constraints otherwise system will fail.For example Scientific experiments, medical imaging systems, industrial control systems, weapon systems, robots, and home-applicance controllers, Air traffic control system etc. There are two types of real-time operating systems. Hard real-time systems Hard real-time systems guarantee that critical tasks complete on time. In hard real-time systems secondary storage is limited or missing with data stored in ROM. In these systems virtual memory is almost never found. Soft real-time systems Soft real time systems are less restrictive. Critical real-time task gets priority over other tasks and retains the priority until it completes. Soft real-time systems have limited utility than hard real- time systems.For example, Multimedia, virtual reality, Advanced Scientific Projects like undersea exploration and planetary rovers etc.•

Limitations of Distributed Operating System

- The added complexity required to ensure proper co-ordination among the sites, is the major limitation. This increased complexity takes various forms :
- Software Development Cost : It is more difficult to implement a distributed database system; thus it is more costly.
- Greater Potential for Bugs : Since the sites that constitute the distributed database system operate parallel, it is harder to ensure the correctness of algorithms, especially operation during failures of part of the system, and recovery from failures. The potential exists for extremely subtle bugs.
- increased Processing Overhead : The exchange of information and additional computation required to achieve intersite co-ordination are a form of overhead that does not arise in centralized system.
- Absence of global clock... so that no synchronization among processes.
- Absence of dynamic memory.... So at a particular time a process can only get partial & coherent state Or complete & incoherent state of the distributed system.
- Coherent:- recorded state of all the processes at any given time