B. E.

Eighth Semester Examination, May-2005

DISTRIBUTED SYSTEM

Note : Attempt any five questions. All questions carry equal marks.

Q. 1. (a) What is a distributed system? What were the causes behind evolution of distributed computing system?

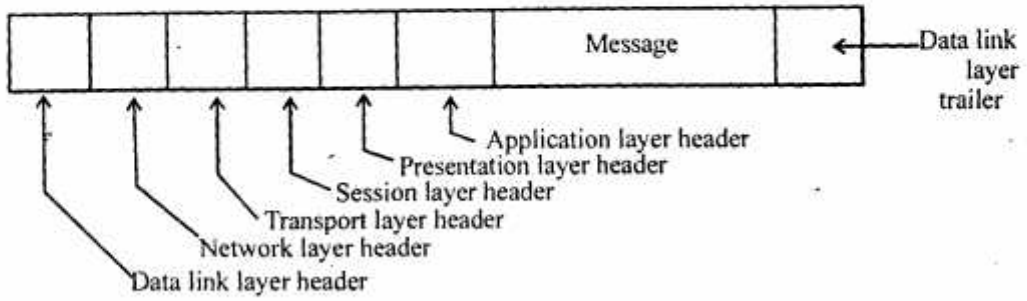
Ans. A distributed system is a collection of independent computers that appear to the users of the system as a single computer. eg. Consider a network of workstations in a university or company department. In addition to each user's personal workstation, there might be a pool of processors in the machine room that are not assigned to specific users but are allocated dynamically as needed. Such a system might have a single file system, with all files accessible from all machines in the same way and using the same path name. Furthermore, when a user types a command the system could look for the best place to execute that command, possibly on the user's own workstation, possibly on an idle workstation belonging to someone else, and possibly on one of the unassigned processors in the machine room. If the system as a whole looked and acted like a classical single-processor timesharing system, it would qualify as a distributed system. The causes behind evolution of distributed computing system are :

1. Historically there were two paths of evolution of computer. Firstly to build a single computer with huge processing capacity like mainframe system. Secondly the goal was achieved by increasing the capacity by using multiple small computers like microprocessors. The later can be termed as distributed system and offered a better price and performance than mainframe. According to principle of economy of scale the price of things made in bulk is less than the those which are fewer in number. And in case of distributed system, we have many microprocessor rather than a single mainframe so they cost less.
2. Distributed systems are designed to allow many users to work together, and parallel system, whose only goal is to achieve maximum speedup on a single problem.
3. A next reason for building a distributed system is that some applications are inherently distributed. A supermarket chain might have many stores, each of which gets goods delivered locally, makes local sales and makes local decisions about which vegetables are so old or rotten than they must be thrown out. If therefore makes sense to keep track of inventory at each store on a local computer rather than centrally at corporate headquarters. After all, most queries and updates will be done locally. Even though, from time to time, top management may want to find out how many rutabagas it currently owns. One way to accomplish this goal is to make the complete system look like a single computer to the application programs, but implement decentrally, with one computer per store. This would then be a commercial distributed system.
4. Another reason for evolution of a distributed system was higher reliability. By distributing the workload over many machines, a single chip failure will bring down atmost one machine, leaving the rest infact. Ideally, if 5 percent of the machine are down at any moment, the system should be able to continue to work with a 5 percent loss in the performance.
5. Finally incremental growth is also potentially a big plus. often, a company will bug a mainframe with

the intention of doing all work on it. If the company prospers and the workload grows, at a certain point the mainframe will no longer be adequate. The only solutions are either to replace the mainframe with a larger one or to add a second mainframe. Both of these can wreak major havoc on the company's operations. In constrastl, with a distributed system, it may be possible simply to add more processors to the system, thus allowing it to expand gradually as the need arises.

Q. 1. (b) Explain the Client-Server Model.

Ans. Whenever any sender wants to send some message, it sets up a connection with the receiver, and then pumps the bits (message) in which arrive without error, in order, at the receiver. Consider the figure below :



The message has to pass through all these layers and the existence of all those headers generates a considerable amount of overhead. Every time a message is sent it must be processed by about half a dozen layers, each one generating and adding a header on the way down or removing & examining a header on the way up. All of this work takes time.

For a LAN-based distributed system, the protocol overhead is often substantial. So much CPU time is wasted running protocols that the effective throughput over the LAN is often only a fraction of what LAN can do. As a consequence, most LAN-based distributed system do not use layered protocols at all.

Thus the client-server model is used. The idea behind this model is to structure the operating system as a group of co-operating processes, called servers, that offer services to the users, called clients. The client & server machines normally all run the same microkerned, with both the clients & servers running as user processes. A machine may run a single process or it may run multiple clients, multipie servers, or a mixture of the two.

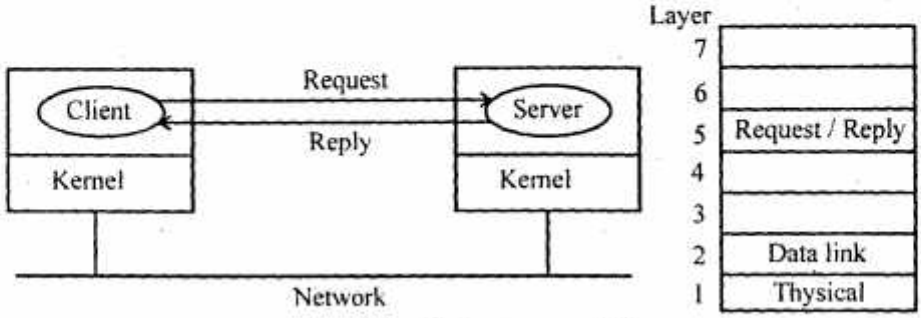


Fig. 1. The client-server model

To avoid the considerable overhead of the connection-oriented protocols such as OSI or TCP/IP, the

client server model is usually based on a simple connectionless request/reply protocol. The client sends a request message to the server asking for some service. The server does the work & returns the data requested or an error code indicating why the work could not be performed as shown in Fig. (1) (a). The client sends a request gets an answer. No connection has to be established before use or form down afterward. The reply message serves as the acknowledgment to the request. The primary advantage is the simplicity. Secondly comes the efficiency. The protocol stock is shorter & thus more efficient. Assuming that all the machines are identical, only three levels of protocols are need as shown in fig. 1 (b). The physical & data link protocols take care of getting the packets from client to server and back. These are always handled by the hardware, for eg.

Ethernet or token ring chip. No routing is needed and no connections are established, so layers 3 and 4 are not needed. Layer 5 is the request/reply protocol. It defines the set of legal requests & the set of legal replies to these requests.

Due to this simple structure, the communication services provided by the (micro) kernel can, for eg. be reduced to two system calls, one for sending messages and one for receiving them.

Q. 2. (a) What do you mean by Synchronization? Justify its need for distributed systems.

Ans. In order to know at what time of day events occur at the processes in the distributed system-for eg, for accountancy, purposes—it is necessary to synchronize the processes' clocks C_i with an authoritative, external source of time. This is external synchronization. And if the clocks C_i are synchronized with one another to a known degree of accuracy, then we can measure the interval between two events occurring at different computers by appealing to their local clocks-even though they are not necessarily synchronized to an external source of time. This is internal synchronization.

The need of synchronization : Consider an example about the implications of the lack of global time on the UNIX make program Normally, in UNIX, large programs are split up into multiple source files, so that a change to one source file only requires one file to be recompiled not all the files. If a program consists of 100 files, not having to recompile everything because one file has been changed greatly increases the speed at which programmers can work.

When the programmers has finished changing all the source files, he starts make, which examines the times at which all the source & object files were last modified. If the source file input C has time 2151 7 the corresponding object file input. O has time 2150, make knows that input C has been changed since input O was created, and thus input. C must be recompiled. On the other hand, if output. C has time 2144 and output. O has time 2145, no compilation is needed here. Thus make goes through all the source files to find out which one's need to be recompiled and calls the compiler to recompile them.

Now imagine distributed system in which there is no global agreement on time. Suppose that output. O has time 2144 as above, and shortly thereafter output. C is modified but is assigned time 2143 because the clock on its machine is slightly shown as shown below :

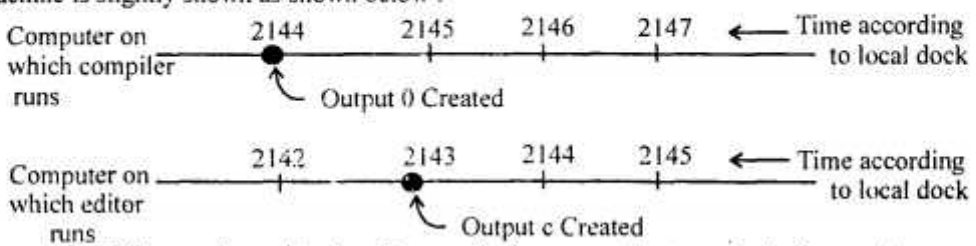


Fig. When each machine has its own clock, an event that occured after another event may nevertheless be assigned an earlier time

Make will not call the complier. The resulting executable binary program will then contain a mixture of object files from the old sources & the new sources. It will probably not work & the programmer will not understand what is wrong with the code.

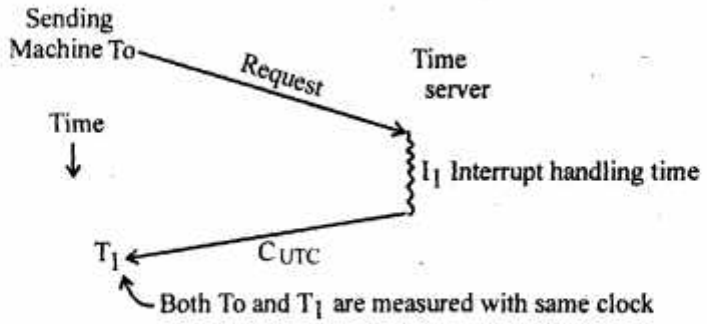
Thus time is so basic thing that all the clocks synchronized can be so dramatic.

Q. 2. (b) Write an algorithm that decides whether a given set of clocks are synchronized or not. What input parameters are needed in you algorithm?

Ans. If one machine has a wwv receiver, the goal becomes keeping all the other machines synchronized to if. If no machines have wwv receivers, each machine keeps track of its own time, and the goal is to keep all the machines together as well as possible.

Cristian's Algorithm :

In this one machine has a wwv receiver and the goal is to have all the other machines stay synchronized with it. Let us call the machine with the wwv receiver a time server. Periodically, certainly no more than every s/ 2p seconds, each machine sends a message to the time server asking it for the current time, C_{UTC} as shown in figure below :



Getting the current time from a time server

As a first approximation, when the sender gets the reply, it can just set its clock to C_{UTC}. However there are two problems, one major and one minor. The major problem is that time must never run backward. If the sender's clock is fast, C_{UTC} will be smaller than the sender's current value of C. Just taking over C_{UTC} could cause serious problems, such as an object file compiled just after the clock change having a time earlier than the source which was modified just before the clock change. Such a change must be introduced gradually. One way is as follows. Suppose that the timer is set to generate 100 interrupts per second. Normally each interrupt would add 10 msec to the time. When slowing down, the interrupt routine adds only 9 msec each time until the correction has been made. Similarly, the clock can be advanced gradually by adding 11 msec at each interrupt instead of jumping it forward all at once.

The minor problem is that it takes a non zero amount of time for the time server's reply to get back to the sender. Worse yet, this delay may be large and vary with the network load. Cristian's way of dealing with it is to attempt to measure it. It is simple enough for the sender to record accurately the interval between send in the request to the time server and the arrival of the reply. Both the starting time, T_0, and the ending time, T_1, are measured using the same clock, so the interval will be relatively accurate, even if the sender's clock is off from UTC by a substantial amount.

In the absence of any other information, the best estimate of the message propagation time is $(T_1-T_0)/2$. When the reply comes in, the value in the message can be increased by this amount to give an estimate of the server's current time. If the theoretical minimum propagation time is known, other properties of time estimate can be calculated.

This estimate can be improved if it is known approximately how long it takes the time server to handle the interrupt and process the incoming message. Let us call the interrupt handling time I. Then the amount of the interval from T_0 to T_1 that was devoted to message propagation is T_1-T_0-I, So the best estimate of the one-way propagation time is half this. Systems do exist in which messages from A to B systematically take a different route than messages from B to A, and thus a different propagation time.

To improve the accuracy, cristian suggested making not one measurement, but a series of them. Any measurements in which T_1-T_0 exceeds some threshold value are discarded as being victims of network congestion and thus unreliable.

Q. 3. (a) Write the goal of process management. How can we achieve this goal?

Ans. In a distributed system, the main goal of process management is to make the best possible use of the processing resources of the entire system by sharing them among all processes. Three important concepts are used in distributed operating systems to achieve this goal.

1. Processor allocation deals with the process of deciding which process should be assigned to which processor.
2. Process migration deals with the movement of a process from its current location to the processor to which it has been assigned.
3. Threads deals with fine-grained parallelism for better utilization of the processing capability of the system.

Process migration is the relocation of a process from its current location (the source node). A process may be migrated either before it starts executing on its source node or during the course of its execution. The former is known as non-preemptive process migration and the latter is known as preemptive process migration.

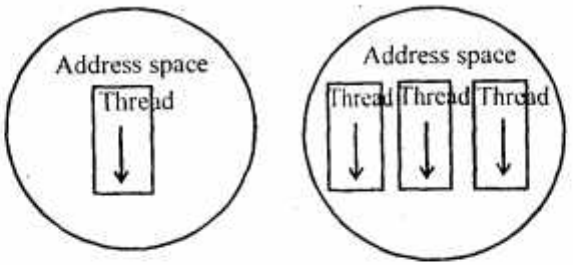
In the operating systems with threads facility, the basic unit of CPU utilization is a thread. In such systems, a process consists of an address space and one or more threads of control. Each threads of a process has its own program counter, its own register states and its own stack. But all the threads of a process share the same address space.

Lastly a resource manager schedules the processes in a distributed system to make use of the system resources in such a manner that resource usage, response time, network congestion, and scheduling overhead are optimized.

Q. 3. (b) What are the threads? Why are these required?

Ans. Threads are a popular way to improve application performance thorough parallelism. In operating systems with thread facility, the basic unit of CPU utilization is a thread. In such systems, a process consists of an address space and one or more threads of control as shown in Fig.

Each thread of a process has its own program counter, its own register states and its own stack. But all the threads of a process share the same address space. Hence they also share the global variables. Threads are often referred to as lightweight processes and traditional processes are referred to as heavyweight processes in which each process was the basic unit of CPU utilizations.



(a) (b)

(a) Single–Threaded

(b) Multiple-threaded.

A single threaded process corresponds to a process of a traditional operating system.

Motivations for using threads :

1. The overheads involved in creating a new process are in general considerably greater than those of creating a new thread within a process.
2. Switching between threads sharing the same address space is considerably cheaper than switching between processes that have their own address space.
3. Threads allow parallelism to be combined with sequential execution and blocking system calls. Parallelism improves performance and blocking system calls make programming easier.
4. Resource sharing can be achieved more efficiently and naturally between threads of a process than between processes because all threads of a process share the some address space.

Q. 3. (c) Describe need of scheduling. Discuss desirable features of a good Global Scheduling Algorithm.

Ans. Consider an example in which processes A and d' B run on one processor and processes C and D run on another. Each processor is time-shared in say 100- msec time slices, with A and C running is the even sliceks and B and D running in the odd ones as shown below–in fig 1. (a).

Time slot	Processor 0	Processor 1
0	A	C
1	B	D
2	A	C
3	B	D
4	A	C
5	B	D

(a)

Time slot	Processor 0	1	2	3	4	5	6	7
0	X				X			
1			X			X		
2		X			X		X	
3	X					X		
4		X'						X
5			X		X			

(b)

1. (a) Two jobs running out of phase with each other.

1. (b) Scheduling matrix for eight processors, each with six time slots. The XS indicated allocated slots.

Suppose that A sends many messages or makes many remote procedure calls to D. During time slice O, A starts up and immediately calls D, which unfortunate is not running because it is now C's turn. After 100 m sec, process switching takes place, and D gets A's message, carries out the work & quickly replies. Because B is now running, it will be another 100 m sec before A gets the reply and can proceed. The net result is one message exchange every 200 m sec. What is needed is a way to ensure that processes that communicate frequently run simultaneously. i.e. scheduling.

One way to achieve this is the use of a conceptual matrix in which each column is the process table for one processor as shown in fig. 1(b). Thus column 4 consists of all the processes that run on processor 4. Row 3 is the collection of all processes that are in slot 3 of same processor, starting with the process in slot 3 of same processor, starting with the process in slot 3 of processor 0, then the process in slot 3 of processor 1 and so on. By putting all the members of a process group in the same slot number, but on different processors, one has the advantage of N-fold parallelism, with a guarantee that all the processes will be run at the same time to maximize communication throughout. Thus in fig. 1 (b), four processes that must communicate should be put into slot 3, on processors 1, 2, 3 and 4 for optimum performance.

Desirable Features of a Good Global Scheduling Algorithm are :

1. No a priori Knowledge about the processes : A good process scheduling algorithm should operate with absolutely no a priori knowledge about the processes to be executed. Scheduling algorithms that operate based on the information about the characteristics and resource requirements of the processes normally pose an extra burden upon the users.

2. Dynamic in Nature : It is intended that a good process-scheduling algorithm should be able to take care of the dynamically changing load (or status) of the various nodes of the system. i.e. process assignment decisions should be based on the current load of the system & not on some fixed static policy.

3. Quick Decision-Making Capability : Quick decisions about the assignment of processes to processors must be taken.

4. Balanced System Performance and Scheduling Overhead : Several global scheduling algorithms collect global state information & use this information in making process assignment decisions. The information regarding the state of the system is typically gathered at a higher cost than in a centralized system. Hence algorithms that provide near-optimal system performance with a minimum of global state information gathering overhead are desirable.

5. Stability : A scheduling algorithm is said to be unstable if it can enter a state in which all the nodes of the system are spending all of their time migrating processes without accomplishing any useful work in an attempt to properly schedule the processes for better performance. This type of fruitless migration of processor is known as process thrashing. This is not desirable.

6. Scalability : A scheduling algorithm should be capable of handling small as well as large networks. An algorithm that make scheduling decisions by first inquiring the workload from all the nodes & then selecting the most lightly loaded nodes as the one for receiving the process(es) has poor scalability factor. Such an algorithm may work fine for small networks but gets crippled when applied to large networks. A simple approach to make an algorithm scable is to probe only M of N nodes for selecting a host.

7. Fault Tolerance : A good scheduling algorithm should not be disabled by the crach of one or more nodes of the system. At any instance of time, it should continue functioning for nodes that are up at that time. Algorithms that have decentralized decision-making capability and consider only available nodes in their decision-making approach have better fault tolerance capability.

8. Fairness of Service : While the average quality of service provided is clearly an important performance index, how fairly service is allocated is also a common concern. For eg., two users simultaneously initiating equivalent processes expect to receive about the same quality of service. What is desirable is a fair strategy that will improve response time without unduly affecting the other process.

Q. 4. Describe the various components of distributed file system. Also explain the implementation issue of Distributed file system.

Ans. DFS provides name transparency to disparate server volumes and shares. Through DFs, an administrator can build a single hierarchical file system whose contents are distributed throughout hyour organization's WAN. In short, DFS can be thought of as a share of other shares.

Domain DFS root : It is published in Active Directory, can be replicated, and can be on any windows 2000 server. Files and directories must be manually replicated to other servers or windows 2000 must be configured to replicate files & directories. Configure the domain DFS root, then the replicas when configuring automatic replication. Links are automatically replicated. There may be up to 31 replicas. Domain DFS root directories can be accessed using the following syntax :

\\ domain \ DFS name

DFS link : A pointer to another shared directory. There can be upto 1000 DFS links for a DFS root. Historically with the universal naming convention (UNC), a user or application was required to specify the physical server & share in order to access file information (that is, the user or application had to specify)

\\ Server \ Share \ Path \ Filename). Even though UNCS can be used directly, a UNC is typically mapped to a drive letter where X : might be mapped to \\ Server \ Share. From that point, a user had to navigate beyond the redirected drive mapping to the date he or she wishes to access (for eg., copy X : \ Path \ More - Path \ .....\ Filename.). As networks continue to grow in size and as organizations begin to use existing storage-both internally and externally - For purposes such as internets, mapping a single drive letter to individual shares scales poorly. Further, although users can use UNC names directly, these users can be overwhelmed by the number of places where data can be stored.

DFS solves these problems by permitting the linking of servers & shares into a simpler, more meaningful name space. This new DFS volume permits shares to be hierarchically connected to other windows shares. Since DFs maps the physical storage into a logical representation, the net benefit is that the physical location of data becomes transparent to users & applications.

DFS Components :

*** DFS root :** A shared directory that can contain other shared directories, files, DFS links and other DFS roots. One root is allowed per server. Types of DFS roots :

Stand alone DFS root : Not published in the Active Directory, cannot be replicated, and can be on any windows 2000 server. This provides no fault tolerance with the DFS topology stored on one computer. A DFS can be accessed using the following syntax :

\\ Server \ DFS name.

Implementation Issues :

1. Transparency : They following four types of transparencies are desirable :

(i) Structure transparency : Although not necessary, for performance, scalability and reliability reasons, a distributed file system normally uses multiple file servers. Each file server is normally a user process or sometimes a Kernel process that is responsible for controlling a set of secondary storage devices of the node

on which it runs. In multiple file servers, the multiplicity of file servers should be transparent to the clients of distributed file system.

(ii) Access transparency : Both local and remote files should be accessible in the same way i.e. the file system interface should not distinguish between local and remote files, and file system should automatically locate an accessed file and arrange for the transport of data to the client's site.

(iii) Naming Transparency : The name of a file should give no hint as to where the file is located. Furthermore, a file should be allowed to move from one node to another in a distributed system without having to change the name of the file.

(iv) Replication Transparency : If a file is replicated on multiple nodes, both the existence of multiple copies and their locations should be hidden from the clients.

2. User mobility : In a distributed system, a user should not be forced to work on a specific node but should have the flexibility to work on different nodes at different time. The performance characteristics of the file system should not discourage users from accessing their files from workstations other than the one at which they usually work.

3. Performance : The performance of a file system is usually measured as the average amount of time needed to satisfy client requests. In a DFS, this time also includes network communication overhead when the accessed file is remote. Although acceptable performance is hard to quantify, it is desirable that the performance of a DFS should be comparable to that of a centralized file system. Users should never feel the need to make explicit file placement decisions to improve performance.

4. Scalability : It is inevitable that a distributed system will grow with time since expanding the network by adding new machines or interconnection two networks together is common place. Therefore, a good DFS should be designed to easily cope with the growth of nodes and users in the system. That is, such growth should not cause serious disruption of service or significant loss of performance to users.

5. Simplicity and ease of use : Several issues influence the simplicity and ease of use of a DFS. The most important issue is that the semantics of the DFS should be easy to understand. This implies that the user interface to the file system must be simple & the number of commands should be as small as possible.

6. High availability : A DFS should continue to function even when partial failures occur due to failure of one or more components, such as a communication link failure, a machine failure, or s storage device crash. When partial failures occur, the DFS may show degradation in performance, functionality or both. High availability & stability are mutually related properties. Both call for a design in which both control and data are distributed. A highly available and scalable DFS should have multiple & independent file servers controlling multiple & independent storage devices. Replication of files at multiple servers is the primary mechanism for providing high availability.

7. High reliability : In a good DFS, the probability of loss of stored data should be minimized as far as practicable. That is, users should not feel compelled to make backup copies of their files because of the unreliability of the system. Rather, the file system should automatically generate backup copies of critical files that can be used in the event or loss of the original ones. Stable storage is a popular technique used by several file systems for high reliability.

8. Data integrity : A file is often shared by multiple users. For a shared file, the file system must guarantee the integrity of data stored in it. i.e. concurrent access requests from multiple users who are competing to access the file must be properly synchronized by the use of some form of concurrence control mechanism. Atomic transactions are a high-level concurency control mechanism after provided to the users by a file system

for data integrity.

9. Security : A DFS should be secure so that its users can be confident of the privacy of their data. Necessary security mechanisms must be implemented to protect information stored in a file system against unauthorized access. Further more, passing rights to access a file should be performed safely i.e. the receiver of rights should not be able to pass them further if he/she is not allowed to do that.

10. Heterogeneity : As a consequence of large scale, heterogenity becomes inevitable in distributed system. Hetrogeneous distributed systems provide the flexibility to their users to use different computer platforms for different applications. eg. a user may use a super computer for simulations, a macintosh for document processing and a UNIX workstation for program development. Easy access to shared data across these diverse platforms would substantially improve usability. Another heterogerieity issue in file systems is the ability to accommodate several different storage media. Therefore a DFS should be designed to allow the integration of a new type of workstation or storage media in a relatively simple manner.

Q. 5. What is objective behind Distributed Shared Memory Model? Describe page based and object based DSM.

Ans. Distributed shared memory (DSM) is an abstraction used for sharing data between computers that do not share physical memory. Processes access DSM by reads, and updates to what appears to be ordinary memory within their address space. However, an underlying run-time system ensures transparently that processes executing at different computers observe the updates mode by one another. It is as though the processes access a single shared memory, but in fact the physical memory is distributed.

The main point of DSM is that it spares the programmer the concerns of message passing when writing applications that might otherwise have to use it.

Page Based DSM :

Basic Design :

The idea behind DSM is simple :

Try to emulate the cache of a multiprocessor using the MMU and operating system software.

In a DSM system, the address space is divided up into chunks, with the chunks being spread over all the processors in the system. When a processor references an address that is not local, a trap occurs, and the DSM software fetches the chunk containing the address and restarts the faulting instruction, which now completes successfully.

Replication :

One improvement to the basic system that can improve performance considerably is to replicate chunks that are read only, read-only constants, or other read-only data structures. Another possibility is to replicate not only read-only chunks, but all chunks. As long as reads are being done, there is effectively no difference between replicating a read-only chunk and replicating a read-write chunk. However, if a replicated chunk is suddenly modified, inconsistent copies are in existence. The inconsistency is prevented by using some consistency protocols.

Finding the Owner :

The simplest solution for finding the owner is by doing a broadcast, asking for the owner of the specified page to respond. An optimization is not just to ask who the owner is, but also to tell whether the sender wants to read or write and say whether it needs a copy of the page. The owner can then send a single message transferring ownership and the page is well, if needed. Broadcasting has the disadvantage of interrupting each

processor, forcing it to inspect the request packet. For all the processors except the owner's, handling the interrupt is essentially wasted time. Broadcasting can use up considerable bandwidth, depending on the hardware.

There could be several other possibilities as well. In one of these, a process is designated as the page manager. It is the job of the manager to keep track of who owns each page. When a process, P, wants to read a page it does not have or wants to write a page it does not own, it sends a message to the page manager telling which operation it wants to perform and on which page. The manager then sends back a message telling the ownership, as required. A problem with this protocol is the potentially heavy load on the page manger, handling all the incoming requests. This problem can be reduced by having multiple page mangers instead of just one.

Another possible algorithm is having each process keep track of the probable owner of each page. Requests for ownership are sent to the probable owner, which forwards them if ownership has changed. If ownership has changed several times, the request message will also have to be forwarded several times. At the start of execution and every n times ownership changes, the location of the new owner should be broadcast, to allow all processors to update their tables of probable owners.

Finding the Copies :

Another important detail is how all the copies are found when they must be invalidated. Again, two possibilities present themselves. The first is to broadcast a message giving the page number and ask all processors holding the page to invalidate it. This approach works only if broadcast messages are totally reliable and can never be lost.

The second possibility is to have the owner or page manager maintain a list or copyset telling which processors hold which pages. When a page must be invalidated, the old owner, new owner, or page manager sends a message to each processor holding the page and waits for an acknowledgment. When each message has been acknowledged, the invalidation is complete.

Page Replacement :

In a DSM system, as in any system using virtual memory, it can happen that a page is needed but that there is no free page frame in memory to hold it. When this situation occurs, a page must be evicted from memory to make room for the needed page. Two subproblems immediately arise : which page to evict and where to put it.

To a large extent, the choice of which page to evict can be made using traditional virtual memory algorithms, such as some approximation to the least recently used algorithm. As with conventional algorithms, it is worth keeping track of which pages are 'clean' and which are 'dirty'. In the context of DSM, a replicated page that another process owns is always a prime candidate to evict because it is known that another copy exists.

Consequently, the page does not have to be saved anywhere. If a directory scheme is being used to keep track of copies, the owner or page manager must be informed of this decision, however. If pages are located by broadcasting, the page can just be discarded.

The second best choice is a replicated page that the evicting process owns. It is sufficient to pass ownership to one of the other copies but information that process, the page manager, or both, depending on the implementation. The page itself need not be transferred, which results in a smaller message.

Object-Based Distributed Shared Memory

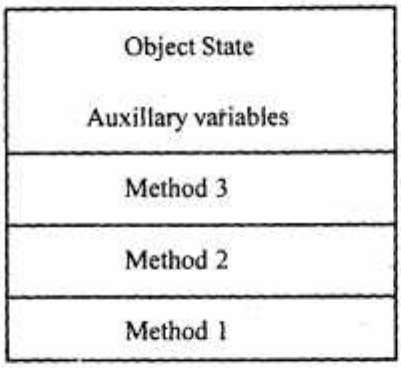
An object is a programmer-defined encapsulated data structure. It consists of :

* internal data

*object state

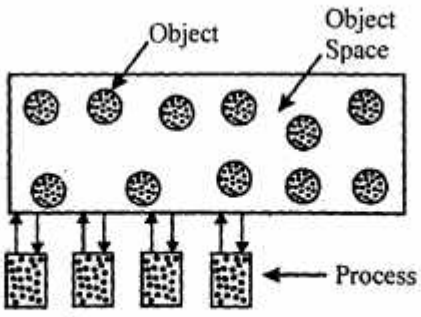
* procedures

The procedures are called methods or operations, that operate on the object state. To access or operate on the internal state, the program must evoke one of the methods. The method can change the internal state, return the state, or something else. Direct access to the internal state is not allowed. This property, called information hiding. Forcing all references to an object's data to go through the methods helps structure the program in a modular way.



In an object-based distributed shared memory, processes on multiple machines share an abstract space filled with shared objects. The location and management of the objects is handled automatically by the runtime system. This model is in contrast to page-based DSM systems, which just provide a raw linear memory of bytes from 0 to some maximum. Any process can invoke any object's methods, regardless of where the process and object are located. It is the job f the operating system and runtime system to make the act of invoking work no matter where the process and the object are located.

Because processes cannot directly access the internal state of any of the shared objects, various optimizations are possible here that are not possible with page-based DSM. For example, since access to the internal state is strictly controlled, it may be possible to relax the memory consistency protocol without the programmer even knowing it.



In object-based DSM, processes communicte by invoking methods on shared objects

Once a decision has been made to structure a shared memory as a collection of separate objects instead of as a linear address space, there are many other choices to be made. Probably the most important issue is whether objects should be replicated or not. It replication is not used, all accesses to an object go through the one and only copy, which is simple, but may lead to poor performance. By allowing objects to migrate from machine to machine, as needed, it may be possible to reduce the performance loss by moving objects to where they are needed.

On the other hand, if objects are replicated, what should be done when one copy is updated? One approach is to invalidate all the other copies so that only the up to date copy remains. Additional copies can be created later, on demand, as needed. An alternative choice is not to invalidate the copies, but to update them. Shared-variable DSM also has this choice, but for page-based DSM, invalidation is the only feasible choice. Similarly, object-based DSM, like shared-variable DSM, eliminates most false sharing.

Object-based DSM has three advantages over the other methods :

1. It is more modular than the other techniques.
2. The implementation is more flexible because accesses are controlled.
3. Synchronization and access can be integrated together cleanly.

Object-based DSM also has disadvantages. For one thing, it cannot be used to run old "dusty deck" multiprocessor program that assume the existence of a shred linear address space that every process can read and write at random. However, since multiprocessors are relatively new, the existing stock of multiprocessor programs that anyone cares about is small.

A second potential disadvantage is that since all accesses to shared objects must be done by invoking the objects's methods, extra overhead is incurred that is not present with shared pages that can be accessed directly.

Q. 6. Explain the various features of MACH.

Ans. Mach is a microkernel-based operating system.

Design Goals and main features : Mach's design was influenced by the research and design goals described below :

1. Open-System Architecture :

One of the main goals of Mach was to design an open system that could provide a base for building new operating systems and emulating existing ones. To achieve this goal, the design philosophy used in Mach was to have a minimal microkernel that would provide a small set of basic abstractions sufficient for deriving other functionality and to implement many traditional kernel-based functions as user-level servers. With this approach, it is both possible & rational to think of traditional operating systems such as UNIX and MS-DOS not as operating system Kernels but as application programs servers or a set of servers that can provide client programs with specific programming abstractions. Note that the microkernel-based design of Mach allows multiple emulators to be run simultaneously. This makes it possible to run programs written for different operating systems such as UNIX and MS-DOS on the same machine at the same time.

2. Compatibility with BSD UNIX :

From the very beginning, an important goal of Mach was to provide full compatibility with BSD UNIX systems so that it could receive wide acceptance in the research and academic communities. This goal was initially achieved by combing Mach and the 4.2 BSD UNIX into a single kernel.

3. Network Transparency :

Similar to any other distributed operating system, network transparency was also a goal in Mach. In order to allow for distributed programs that extend transparently between uniprocessors and multiprocessors across a network, Mach has adopted a location-independent communication model involving ports as destinations. The Mach Kernel, however, is designed to be 100% unaware of networks. The Mach design relies totally on user-level network server processes to Ferry messages transparently across the network (fig.1)

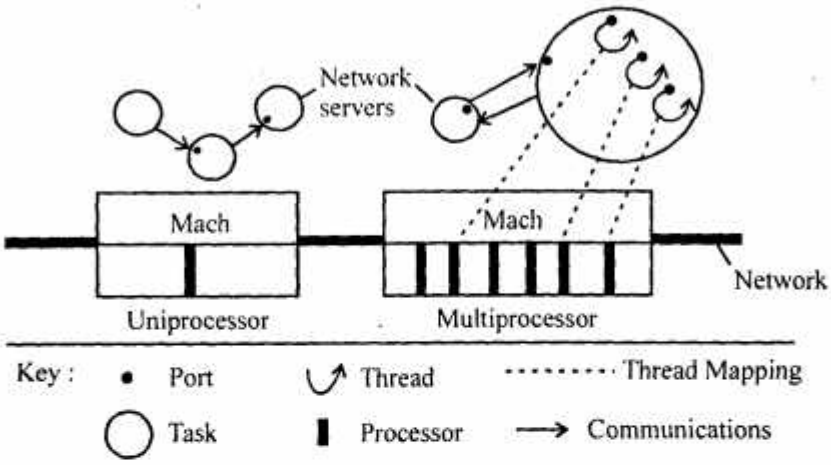


Fig. Mach tasks, threads & communication

4. Flexible Memory Management :

Another important goal of Mach was to support a powerful & flexible memory management system. To achieve this goal, Mach provides an elaborate virtual-memory system that is implemented in terms of fixed-size pages. Some of the attractive features of Mach's memory management system are as follows :

(i) It clearly separates the machine-independent parts of the memory management system from the machine-dependent parts, making the memory management system for more portable than in other systems.

(ii) It is integrated with the communication system, allowing the realization of fast local IPC.

(iii) It has a copy-on-write mechanism for efficient sharing of data between two or more processes. In this mechanism, data are physically copied only when they are changed. Hence, it provides the potential to eliminate much data movement between the kernel, block I/O devices, clients, and servers.

(iv) It has an inheritance mechanism that allows a parent process to declare which regions of memory are to be inherited by its children and which are to be read-writable. This mechanism provides for various sharing policies to enforce protection between the parent and its children processes.

(v) It has an external memory manager concept, which allows the implementation and use of multiple user-level memory managers for handling different memory managers. Each user-level memory manager can implement its own semantics and paging algorithm. Suitable to the object it is backing.

5. Flexible IPC : It provides a flexible IPC system that can allow processes to communicate in a reliable and efficient manner. To achieve this, Mach uses a message-based IPC that is based on ports, which are kernel objects that hold messages. This IPC system has the following features :

(i) It supports both synchronous & asynchronous message passing.

(ii) It guarantees reliable and sequenced delivery of messages.

(iii) It ensures secure message communication by using a capability-based access control mechanism for controlling access to ports. All messages are sent to and received from ports.

(iv) It supports network transparency.

(v) It supports network transparency.

(v) It supports heterogeneity by translating data types from one machine's representation to another's when the data is transferred between two machines of different types.

High Performance :

The use of micro-kernal-based design approach in Mach is subject to a performance penalty because message passing between serving processes & the microkernel requires context switches, slowing down the system. Some of the important techniques used in design of Mach to obtain high performance are :

(i) Use of multithreaded processes to take advantage of fine-grained parallelism for multiprocessing.

(ii) Use of hand-off thread scheduling policy for fast local IPC.

(iii) Use of copy-on-write mechanism to minimize the copying of data. Notice that the largest CPU cost of many operations in a traditional Kernel is the copying of data.

(iv) Use of transparent shared library in the user space to perform server work in the client address space.

7. Simple Programmer Interface : To achieve this Mach provides an interface generator called Mach interface Generator (MIG). MIG is basically a compiler that generates stub procedure from a service definition. The stub procedures of all services are placed in a transparent shared library.

Q. 7. What is meant by DCE? Describe the top DCE threads and security service in DCE.

Ans. OCE means Distributed Computing Environment :

The primary goal of DCE is to provide a coherent, seamless environment that can serve as a platform for running distributed applications.

Threads : The threads package, along with the RPC package is one of the fundamental building blocks of DCE. The DCE threads is a collection of user-level library procedures that allow processes to create, delete and manipulate threads. A thread can be in one of four states as shown in figure :

A thread can be in one of four states :

State	Description
Running	The thread is actively using the CUP
Ready	The thread wants to run
Waiting	The thread is blocked waiting for some event
Terminated	The thread has exited but not yet destroyed

A running thread is one that is actively using the CPU to do computation and so on. On a machine with one CPU, only one thread can be actually running at a given instant. On a multiprocessor, several threads within a single process can be running at one, on different CPUs.

The threads package has been designed to minimize the impact on existing software, so programs designed for a single-threaded environment can be converted to multithreaded processes with a minimum of work. Ideally, a single-threaded program can be converted into a multithreaded one just by setting a parameter indicating that more threads should be used.

Scheduling : Thread scheduling is similar to process scheduling except that it is visible to application. The scheduling algorithm determines how long a thread may run and which thread runs next.

Synchronization : DCE provides two ways of synchronization : (a) Mutexs (b) Condition variables.

Mutexes are used when it is essential to prevent multiple threads from accessing the same resource of the same time. Condition variables are used in conduction with mutexes. Typically, when a thread needs some resource, it uses a mutex to gain exclusive access to a data structure that keeps track of the status of the resource. If the resource is not available, the thread waits on condition variable, which atomically suspends the thread and releases the mutex.

Security Services : In DCE, a user or a process (client / server) that needs to communicate securely is called a principal. For convenience of access control, principals are assigned membership in one or more groups and organizations. All principals of the same group or organizations have the same access rights. Typically, a principal is a member of one organization but may simultaneously be a member of multiple groups. Each principal has a unique identifier associated with it. Together, a principal's identifier, group and organization membership are known as the principal's privilege attributes.

The main components of the DCE Security Service for a single cell are shown in the fig. These components collectively provide authentication, authorization, message integrity, and security administration services :

1. Authentication in DCE :

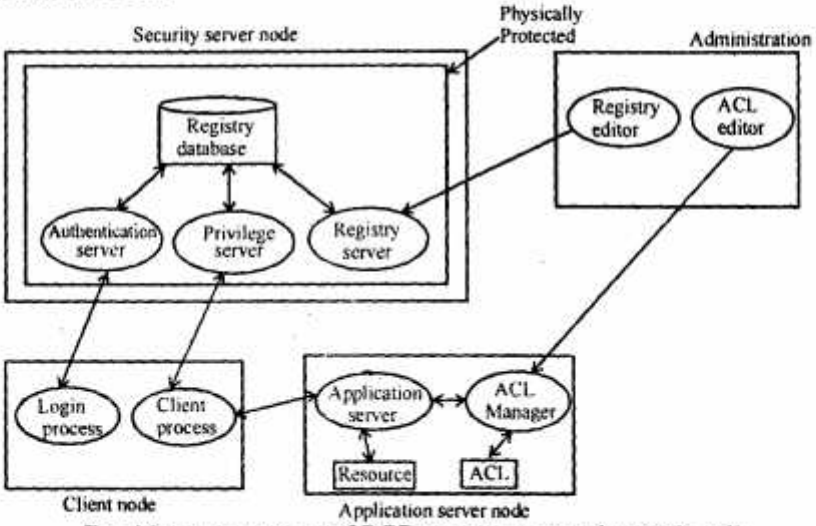


Fig. Main components of DCE security service for single cell

The information registered in the registry database includes each principal's secret key and priviledge attributes. The protocols are used for authenticating a user at the time of login and for mutual authentication of

a client and a server. The establishment of a secure logical communication channel between a client and a server by using the authentication protocol is known as authenticated RPC in DCE. Once authenticated RPC has been established, it is up to the client & the server to determine how much security is desired.

2. Authorization in DCE : It is based on ACLs. Associated with each application server is an ACL and an ACL manager. The ACL contains complete information about which principals have what rights for the resources managed by the server. When a client's request comes to a server, it extracts the client's ID and its group and organization membership information from the received encrypted ticked. It then passes the clients ID, membership and the operation desired to the ACL manager. Using this information, the ACL manager checks the ACL to make a decision if the client is authorized to perform the requested operation. It returns an access granted or denied reply to the server, after which server acts accordingly.

3. Message Integrity in DCE :

Once authenticated RPC has been established, it is up to the client & server to determine how much security is desired. Therefore if message integrity is desired, it can be ensured by the use of a digital signature technique. That is, application can ensure data integrity by including an encrypted digest of the message data passed between clients and servers. The digest must be encrypted and decrypted by using the session key that a client & a server share for secure communication between them.

4. Security Administration in DCE :

The administrator, registry server and ACL manager jointly perform security administration tasks. Tow programs are used by the administrator for performing administration tasks. One is the registry editor program & the other is the ACL editor program. The registry editor program may be used by the system administrator to view, add, delete & modify information in the registry database. Even system administrators do not have direct access to the registry database, & they access the the registry database only by making requests to the registry server. This is much safer, for although an administrator can change any password.

On the other hand, the ACL editor program may be used by an application administrator to view, add, delete & modify entries in ACL's for applications of ACLS for objects controlled by them.

Q. 8. Explain the following :

(a) Deadlocks in Distributed Systems.

Ans. Deadlocks in Distributed Systems : If the total request made by multiple concurrent processes for resources of a certain type exceeds the amount available, some strategy is needed to order the assignment of resources in time. A situation in which competing processes prevent their mutual progress even though no single one requests more resource than are available is a deadlock situation. It may happen that some of the processes that entered the waiting state (because the requested resources were not available at the time of request) will never change state, because the resources they have requested are held by other waiting processes. This situation is called a deadlock and processes involved are said to be deadlocked. Hence, deadlock is the state of permanent blocking of a set of processes each of which is waiting for an event that only another process in the set can cause.

Necessary Conditions for Deadlock : It is necessary for a deadlock situation to occur in a system if these conditions hold true :

(i) Mutual-exclusion condition : If a resource is held by a process, any other process requesting for the resource must wait until the resource has been released.

2. Hold-and-wait condition : Processes are allowed to request for new resources without releasing the resources that they are currently holding.

3. No-preemption condition : A resource the has been allocated to a process becomes available for allocation to another process only after it has been voluntarily released by the process holding it.

4. Circular-wait condition : Two or more processes must form a circular chain in which each process is waiting for a resource that is held by the next member of the chain.

Various strategies are used to handle deadlocks. Four of the best-known ones are :

1. The ostrich algorithm (ignore the problem)
2. Detection (let deadlock occur, detect them, and try to recover)
3. Prevention (statically make deadlocks structurally impossible)
4. Avoidance (avoid deadlocks by allocating resources carefully).

Q. 8. (b) Remote Procedure call and Group Communication.

Ans. Remote Procedure Call : Although the client-server model provides a convenient way to structure a distributed operating system, it suffers from one incurable flaw : the basic paradigm around which all communication is built is input / output. The procedures send & receive are fundamentally engaged in doing I/O is not one of the key concepts of centralized systems, making it the basis for distributed computing has struck many workers. The goal is to make distributed computing look like centralized computing. When a process on machine A calls a procedure on machine B, the calling process on A is suspended & execution of the called procedure takes place on B. Information can be transported from the caller to the callee in the parameters and can come back in the procedure result. No message passing or I/O at all is visible to the programmer. This method is known as remote procedure call, or often just RPC.

Basic RPC operation : The idea behind RPC is to make a remote procedure call. In other words, we want RPC to be transparent-the calling procedure should not be aware that the called procedure is executing on a different machine or vice-versa. Suppose that a program needs to read some data from a file. The programmer puts a call to read in the code to get the data. RPC achieves its transparency in an analogous way. When read is actually a remote procedure (eg., one that will run on the file server's machine), a different version of read, called a client stub is put into the library. The parameters are packed into a message & asks the kernel to send the message to the server as shown :

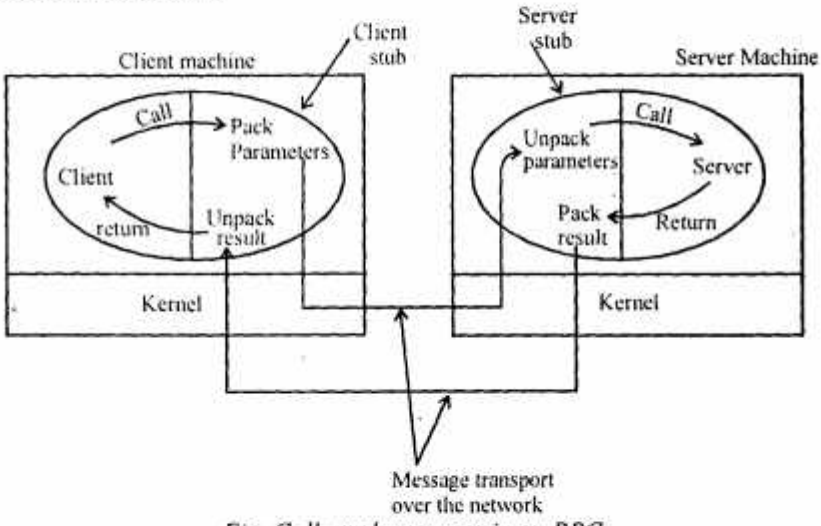


Fig. Calls and messages in an RPC

Steps in RPC :

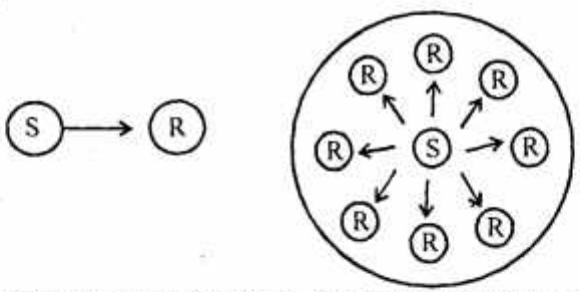
1. The client procedure calls the client stub in norway way.
2. The client stub builds a message & traps to the kernel.

3. The kernel sends the message to the remote kernel.
4. The remote kernel gives the message to the server stub.
5. The server stub unpacks the parameters and calls the server.
6. The server does the work & returns the result to the stub.
7. The server stub packs it in a message & traps to kernel.
8. The remote kernel sends the message to the client's kernel.
9. The client's kernel gives the message to the client stub.
10. The stub unpacks the result & returns to the client.

Group Communication :

An underlying assumption intrinsic to RPC is that communication involves only two parties, the client and the server. Sometimes there are circumstances in which communication involves multiple processes, not just two. In such a system, it might be desirable for a client to send a message to all the servers, to make sure that the request could be carried out even if one of them crashed. RPC connot handle communication from one sender to many receivers, other than by performing separate RPCs with each one.

A group is a collection of processes that act together is some system or user-specified way. The key property that all groups have is that when a massage is sent to the group itself, all members of the group receive it. It is a form of one-to-many communication (one sender, many receivers), and is contrasted with point-to-point communication in fig.



(a) Point-to-point communication is from one sender to one receiver *(b) One-to-many communication is from one sender to multiple receivers.*

Groups are dynamic. New groups can be created and old groups can be destroyed. A process can join a group or leave one. A process can be a member of several groups at the same time. Consequently, mechanisms are needed for managing groups and group membership.

The purpose of groups is to allow processes to deal with collections of processes as a single abstraction. Thus a process can send a message to a group of servers without having to know how many there are or where they are, which may change from one call to the next.

The implementation of group communication depends to a large extent on the hardware. On some networks, it is possible to create a special network address (for eg. indicated by setting one of the high-order bits to 1), to which multiple machines can listen. When a packet is sent to one of these addresses, it is automatically delivered to all machines listening to the address. This technique is called multicasting. Implementing groups using multicast is straight forward : just assign each group a different multicast address.