

B.E.

Eighth Semester Examination, Dec-2005

DISTRIBUTED SYSTEM

Note : Attempt any five questions.

Q. 1. (a) Write and discuss any three advantages and disadvantages of distributed systems over independent P.C.S.

Ans. Given that microprocessors are a cost-effective way to do business, one can have his own PC and work independently but for some things, many users need to share data. For eg. airline reservation clerks need access to the master data base of flights & existing reservations. Giving each clerk his own private copy of the entire data based would not work, since nobody would know which seats the other clerks had already sold. Shared data are absolutely essential to this and many other applications, so the machine must be interconnected. Interconnecting the machines leads to a distributed system.

Secondary, sharing often involves more than just data. Expensive peripherals, such as color laser printers, phototypesetters, and massive archival storage device (eg. optical jukeboxes), are also candidates.

A third reason to connect a group of isolated computers into a distributed system is to achieve enhanced person-to-person communication. For many people, electronic mail has numerous attractions over paper mail, telephone and FAX. It is much faster than paper mail, does not require both parties to be available at the same time as does the telephone and unlike FAX, produces documents that can be edited, rearranged, stored in the computer to manipulated with text processing programs.

Disadvantages :

Although distributed systems have their strengths, they also have their weaknesses. First problem is the software. With the current state-of-the art, we do not have much experience in designing, implementing, and using distributed software. What kinds of operating systems, programming languages, and applications are appropriate for these systems? How much should the users know about the distribution? How much should the system do and how much should the users do?

A second potential problem is due to the communication network. It can lose messages, which requires special software to be able to recover, and it can become overloaded. When the network saturates, it must either be replaced or a second one must be added. In both cases, some portion of one or more buildings may have to be rewired at great expense, or network interface boards may have to be replaced (eg. by fiber optics). Once the system comes to depend on the network, its loss or saturation can negate most of the advantages the distributed system was built to achieve.

Finally, the easy sharing of data, which we described as an advantage, may turn out to be a two-edged sword. If people can conveniently access data all over the system, they may equally be able to conveniently access data that they have no business looking at. In other words, security is often a problem. For data that must be kept secret at all costs, it is often preferable to have a dedicated, isolated personal computer that has no network connections to any other machines, and is kept in a locked room with a secure safe in which all the floppy disks are stored.

Q. 1. (b) What are the design issues for distributed systems? Discuss briefly.

Ans. The design issues are :

1. Transparency : The single most important issue is how to achieve the single-system image. A system that realizes this goal is often said to be transparent. Transparency can be achieved at two different levels. Easiest to do is to hide the distribution from the users. At a lower level it is possible, but harder, to make the system look transparent to programs.

The concept of transparency can be applied to several aspects of a distribution system :

Kind	Meaning
Location transparency	The users cannot tell where resources are located.
Migration transparency	Resources can move at will without changing their names.
Replication transparency	The users cannot tell how many copies exist.
Concurrency replication	Multiple users can share resources automatically.
Parallelism replication	Activities can happen in parallel without users knowing.

Different kinds of transparency in a distributed system.

2. Flexibility : It is important that the system be flexible. Flexibility, along with transparency, is preferred. But the things are not as simple as they seem. There are two schools of thought concerning the structure of distributed system. One school maintains that each machine should run a traditional kernel that provides most services itself. The other maintains that the kernel should provide as little as possible, with the bulk of the operating system services available from user-level servers. The two models are known as the monolithic kernel and micro kernel respectively.

The micro kernel is more flexible because it does almost nothing. It basically provides just four minimal services :

- (i) An interprocess communication mechanism.
- (ii) Some memory management.
- (iii) A small amount of low-level process management and scheduling.
- (iv) Low-level input/output.

3. Reliability : One of the original goals of building distributed systems was to make them more reliable than single-processor systems. The goal is that to function at all, current distributed systems count on a number of specific servers being up. It is important to distinguish various aspects of reliability. Availability refers to the fraction of time that the system is usable. Availability can be enhanced by a design that does not require the simultaneous functioning of a substantial number of critical components. Redundancy also improves availability. Key pieces of hardware & software should be replicated so that if one of them fails the others will be able to take up the slack.

A highly reliable system must be highly available. Another aspect of overall reliability is security. Files & other resources must be protected from unauthorized usage. Another issue relating to reliability is fault tolerance. i.e. the distributed systems can be designed to mask failures, that is, to hide them from the users.

4. Performance : Any distributed system should run faster as compared to a single processor. The performance problem is compounded by the fact that communication, which is essential in a distributed system is typically quite slow. Sending a message and getting a reply over a LAN takes about 1 msec most of this time is due to unavoidable protocol handing on both ends, rather than the time the bits spend on the wire. Thus to optimize performance, one often has to minimize the number of messages. The difficulty with this strategy is that the best way to gain performance is to have many activities running in parallel on different processors, but doing so requires sending many messages.

One possible way out is to pay considerable attention to the grain size of all computations. In general, jobs that involve a large number of small computations, especially ones that interact highly with one another, may cause trouble on a distributed system with relatively slow communication. Such jobs are said to exhibit fine-grained parallelism. Whereas jobs are said to exhibit cross-grained parallelism that involve large computation & may be a better fit.

5. Scalability : One principle used is to avoid centralized components, tables and algorithms. Having a single mail server for 50 million users would not be a good idea. Even if it had enough CPU & storage capacity, the network capacity into and out of it would surely be a problem. Furthermore, the system would not tolerate faults well. A single power outage could bring the entire system down.

Concept	Example
Centralized components.	A single mail server for all users.
Centralized tables.	A single one-line telephone book
Centralized Algorithms.	Doing routing based on complete information.

Fig. 1 : Potential bottlenecks that designers should try to avoid in very large distributed systems

Q. 2. (a) What do you mean by logical clocks? Discuss briefly?

Ans. With a single computer and a single clock, it does not matter if the clock is off by a small amount. Since all processes on the machine use the same clock, they will still be internally consistent. As soon as multiple CPUs are introduced, each with its own clock, the situation changes. Although the frequency at which a crystal oscillator runs is unusually fairly stable, it is impossible to guarantee that the crystals in different computers all run at exactly the same frequency. In practice, when a system has n computers, all n crystals will run at slightly different rates, causing the clocks gradually to get out of sync and give different values when read out. This difference in the values is called clock skew. As a consequence of this clock skew, programs that expect the time associated with a file, object, process or message to be correct & independent of the machine on which it was generated can fail.

This brings us to a question, whether it is possible to synchronize all the clocks to produce a single, unambiguous time standard. In a classic paper, lamport showed that clock synchronization is possible & presented an algorithm for achieving it.

Lamport pointed out that clock synchronization need not be absolute. If two processes do not interact, it is not necessary that their clocks be synchronised because the lack of synchronisation would not be observable and thus could not cause problems. Furthermore, he pointed out that what usually matters is not that all processes agree on exactly what time it is, but rather, that they agree on the order in which events occur. In the make example above, what counts is whether input. C is older or newer than input. O, not their absolute certain

times.

For many purpose, it is sufficient that all machines agree on the same time. It is not essential that this time also agree with the real time as announced on radio every hours. For running make, for example, it is adequate that all machines agree that it is 10 :00, even if it is really 10 : 02. Thus for a certain class of algorithms it is the internal consistency of the clocks that matters, not whether they are particularly close to the real time. For there algorithms, it is conventional to speak of the clocks as Logical Clocks.

To synchronize logical clocks, Lamport defined a relation called happens before. The expression $a \rightarrow b$ is read "a happens before b" and means that all processes agree that first event a occurs, then afterward, event b occurs. The happens-before relation can be observed directly in two situations :

1. If a and b are events in the same process, and a occurs before b, then $a \rightarrow b$ is true.
2. If a is the event of a message being sent by one process, and b is the event of message being received by the another process, then $a \rightarrow b$ is also true. A message cannot received before it is sent. Or even at the same time it is sent, since it takes a finite amount of time to arrive.

Q. 2. (b) Does using time stamping for concurrency control ensure serializability? Discuss.

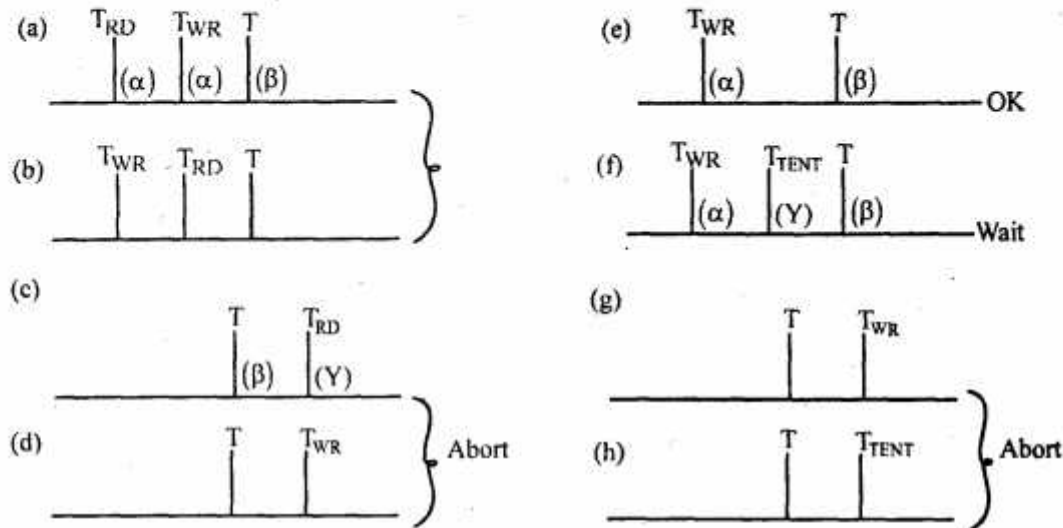
Ans. An approach to concurrency control is to assign each transaction a time stamp at the moment it does BEGIN-TRANSACTION. Using Lamport's algorithm, we can ensure that the timestamps are unique, which is important. Every file in the system has a read timestamp and a write timestamp associated with it, telling which committed transaction last read & wrote it, respectively. If transactions are short and widely spaced in time, if will normally occur that when a process tries to access a file, the file's read & write timestamps will be lower (older) than the current transaction's time-stamp. This ordering means that the transactions are being processed in the proper order, so everything is all right. When the ordering is incorrect, it means that a transaction that started later than the current one has managed to get in there, access the file, and commit. This situation means that the current transaction is too late, so it is observed. Consider an example-imagine that there are three transactions, alpha, beta and gamma. Alpha run a long time ago, and used every file needed by beta and gamma. Alpha ran a long time ago, and used every file needed by beta and gamma, so all their files have read & write timestamps set to alpha's timestamp. Beta and gamma start concurrently, with beta having a lower timestamp than gama. Let us first consider beta writing a file. Call its timestamp, T, and the read & writing a file. Call its timestamp, T, and the read & write Timestamps of the file to be written T_{RD} and T_{WR} , respectively. Unless gamma has snuck in already and committed, both T_{RD} & T_{WR} will be alpha's timestamp, and thus less than T. In figure (a) & (b) we see that T is larger than both T_{RD} & T_{WR} (gamma has not already committed), so the write is accepted and done tentatively. It will become permanent when beta commits. Beta's timestamp is now recorded in the file as a tentative rule.

In fig (c) & (d) beta is out of luck. Gamma has either read (c) or written (d) the file and committed. Beta's transaction is aborted. However, it can apply for a new timestamp and start all over again.

Now look at reads. In fig (e), there is no conflict, so the read can happen immediately. In fig (f), some interloper has gotten in there and is trying to write the file. The interloper's timestamps is lower than beta's, so beta simply waits until the interloper commits, at which time it can read the new file and continue.

In fig (g), gamma has changed the file and already committed. Again beta must abort. In fig (h), gamma is in the process of changing the file, although it has not committed yet. Still, beta is too late and must abort.

Timestamping has different properties than locking. When a transaction encounters a larger timestamp, it aborts, whereas under the same circumstances, with locking it would either late or be able to proceed immediately. On the other hand, it is deadlock free, which is a big plus. All in all, transactions offer many advantages and thus are a promising technique for building reliable distributed systems. Their chief problem is their great implementation complexity, which yields low performance. These problems are being worked on, and perhaps in due course they will be solved.



Concurrency Control Using Timestamps

Q. 3. What are deadlocks? Discuss Chandy-Misra-Hass distributed deadlock detection algorithm.

Ans. Deadlocks in distributed system are similar to deadlocks in single processor system. They are harder to avoid, prevent or even detect and harder to cure when tracked down because all the relevant information is scattered over many machines. Two kinds of distributed deadlocks :

1. Communication deadlocks.
2. Resource deadlocks.

A communication deadlock occurs for e.g. when process A is trying to send a message to process B, which in turn is trying to send one to process C, which is trying to send one to A.

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

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

Distributed Deadlock Detection : Finding general methods for preventing or avoiding distributed deadlocks appears to be quite difficult, so many researchers have tried to deal with the simpler problem of just detecting deadlocks, rather than trying to inhibit their occurrence.

Chandy-Misra-Hass algorithm : In this algorithm, processes are allowed to request multiple resource (eg locks) at once, instead of one at a time. By allowing multiple requests simultaneously, the growing phase of a transaction can be speeded up considerably. The consequence of this change to the model is that a process may now wait on two or more resource simultaneously. Fig shows a modified resource graph, where only the processes are shown. Each arc passes through a resource, as usual, but for simplicity the resources have been omitted. Process 3 on machine 1 is waiting for two resources, one held by process 4 and one held by process 5.

Some of the processes are waiting for local resources, such as process 1, but others, such as process 2, are waiting for resources that are located on a different machine.

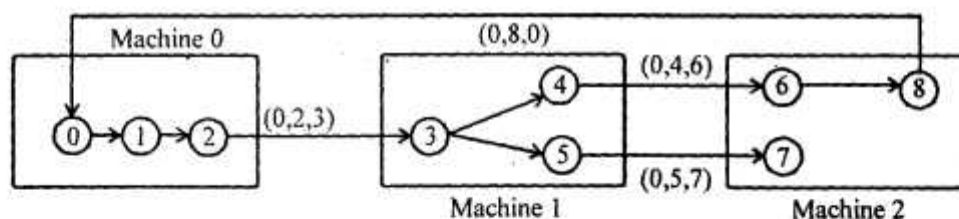


Fig : The chandy-Misra-Hass distributed deadlock detection algorithm

It is precisely these cross-machine arcs that make looking for cycle difficult. The Chandy-Misra-Hass algorithm is invoked when a process has to wait for some resources. For e.g, process a blocking on process 1. At that point a special probe message is generated and sent to the process (or processes) holding the needed resources. The message consist of three numbers; the process that just blocked, the process sending the message & the process to whom it is being sent. The initial message from 0 to 1 contains the triple (0, 0, 1).

When the message arrives, the recipient checks to see if it itself by its own process number & the third one by the number of the process it is waiting for. The message is then sent to the process on which it is blocked. If it is blocked on multiple processes, all of them are sent (different) messages. This algorithm is followed whether the resource is local or remote. In fig we see the remote messages labelled (0, 2, 3), (0, 4, 6), (0, 5, 7) and (0, 8, 0). If a message goes all the way around & comes back to the original sender i.e. the process listed in the first field, a cycle exists and the system is dead-locked. There are various ways in which the deadlock can be broken. One way is to have the process that initiated the probe commit suicide. However this method has problems if several processes invoke the algorithm simultaneously. In fig. for e.g. imagine that both 0 and 6 block at the same moment & both initiate probes. Each would eventually discover the deadlock & each would kill itself. This is overkill. Getting rid of one of them is enough.

Q. 4. (a) Write a few design issues for threads-packages.

Ans. A set of primitives (e.g. library calls) available to the user relating to threads is called a threads package. The first issue concerned with the architecture & functionality of thread packages is thread management. Two alternatives are possible-static threads & dynamic threads. In case of static design, the choice of how many threads there will be is made when the program is written or when it is compiled. Each thread is allocated a fixed stack. This approach is simple, but inflexible.

A more general approach is to allow threads to be created & destroyed on-the-fly during execution. The thread creation call usually procedure) and a stack size, and may specify other parameter as well, for eg, a scheduling priority. This all usually returns a thread identifier to be used in subsequent calls involving the thread. In this model, a process starts out with one (implicit) thread, but can create one or more threads as needed & these can exit when finished. Threads can be terminated in one of two ways. A thread can exit voluntarily when it finishes its job, or it can be killed from outside.

Since threads share a common memory, they can, use it for holding data that are shared among multiple threads such as buffers in a producer-consumer system. Access to shared data is usually programmed using critical regions, to prevent multiple threads, from trying to access the same data at the same time. Critical regions are most easily implemented using semaphores, monitors etc. One technique that is commonly used in threads packages is the mutex which is a kind of watered-down semaphore. Two operations are possible on mutex—Lock and Unlock, the Lock succeeds & the mutex becomes locked in a single atomic action.

Another synchronization feature that is sometimes available in threads packages is the condition variable, which is similar to the condition variable used for synchronization in monitors. Each condition variable is normally associated with a mutex at the time it is created. The difference between mutexes and condition variables is that mutexes are used for short-term locking, mostly for guarding the entry to critical regions. Condition variables are used for long-term waiting until a resource becomes available. A thread locks a mutex to gain entry to a critical region. Once inside the critical region, it examines system tables and discovers that some resource it needs is busy. If it simply locks a second mutex (associated with the resource) the outer mutex will remain locked and the thread holding the resource will not be able to enter the critical region to free it. Deadlock occurs.

One solution is to use condition variables to acquire the resource as shown below :

```
Lock mutex;  
    check data structures;  
    while (resource busy)  
        wait (condition variable);  
    mark resource as busy;  
unlock mutex;
```

fig. 1. (a)

Here, waiting on the condition variable is defined to perform the wait & unlock the mutex atomically. Later, when the thread holding the resources free it as shown below, it calls wakeup, which is defined to wakeup either exactly one thread or all the threads waiting on the specified condition variable.

```
lock mutex;  
    mark resource as free;  
unlock mutex;  
wakeup (condition variable);
```

Fig. 1 (b)

The use of while loop instead of If in fig 1 (a) guards against the case that the thread is awakened but that someone else seizes the resource before the thread runs.

Q. 4. (b) A real time system has periodic processes with following computational requirements and periods :

P 1 : 20 msec every 40 msec.

P 2 : 60 msec every 500 msec.

P 3 : 5 msec every 20 msec.

P 4 : 15 msec every 100 msec.

Is this system schedulable on one C.P.U.? Justify Your answer.

Ans.

P1 : 20 m sec every 40 m sec

P3 : 5 m sec every 20 msec

Now after every 40 m sec process P1 runs for 20 m sec. On the other hand after every 20 m sec, process P3 runs for 5 m sec.

Now after 40 m sec P3 will run for 10 m sec.

But at that time processor is busy with process P1. It is not possible to run P3. Thus we need more than one processor.

	0	5	10	15	20	25	30	35	40	45	50	55
P ₁									✓	✓	✓	✓
P ₂												
P ₃				✓					✓			✓
P ₄												

Consider that P1 & P3 need processor at same time.

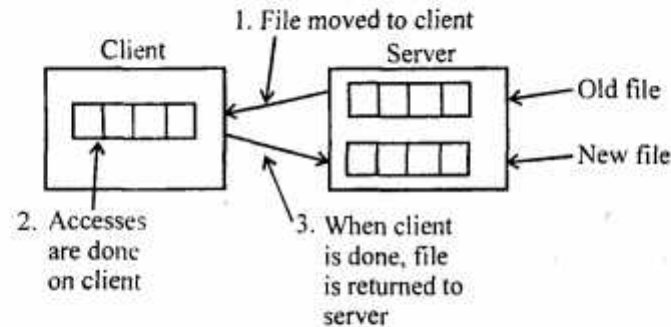
Q. 5. (a) Define file service interface. Discuss upload/download model briefly.

Ans. For any file service, whether for a single processor or for a distributed system, the most fundamental issue is : what is a file? The meaning & structure of the information in the files is entirely up to the application programs. A file can have attributes which are pieces of information about the file but which are not part of the file itself. Typical attributes are the owner, size, creation date, and access permissions. The file service usually provides primitives to read and write some of the attributes. For e.g., it may be possible to change the access permissions but not the size (other than by appending data to the file).

Another important aspect of the file model is whether files can be modified after they have been created. Normally, they can be, but in some distributed systems, the only file operations are CREATE and READ. Once a file has been created, it cannot be changed. Such a file is said to be immutable. Having files be immutable makes it much easier to support file caching and replication because it eliminates all the problems associated with having to update all copies of a file whenever it changes.

Protection in distributed systems uses essentially the same technique as in single-processor systems : capabilities and access control lists. With capabilities, each user has a kind of ticket, called a capability, for each object to which it has access. The capability specifies which kinds of accesses are permitted (eg, reading is allowed but writing is not).

All access control list schemes associate with each file a list of users who may access the file and how. The UNIX scheme, with bits for controlling reading, writing and executing each file separately for the owner, the owner's group, and everyone else is a simplified access control list.



The upload / download model

File services can be split into two types, depending on whether they support an upload/download model or a remote access model. In the upload/download model, shown in figure 2 (a), the file service provides only two major operations : read file and write file. The former operation transfer an entire file from one of the file servers to the requesting client. The latter operation transfers an entire file the other way, from client to server. Thus the conceptual model is moving whole files in either direction. The files can be stored in memory or on a local disk, as needed.

The advantage of the upload/download model is its conceptual simplicity. Application programs fetch the files they need, then use them locally. Any modified files or newly created files are written back when the program finishes. No complicated file service interface has to be mastered to use this model. Furthermore, whole file transfer is highly efficient. However, enough storage must be available on the client to store all the files required. Furthermore, if only a fraction of a file is needed, moving the entire file is wasteful.

Q. 5. (b) Why replication is needed? What are the reasons for offering such a service?

Ans. Distributed file systems often provide file replication as a service to their clients. In other words, multiple copies of selected files are maintained, with each copy on a separate file server. The reasons for offering such a service vary, but among the major reasons are :

1. To increase reliability by having independent backups of each file. If one server goes down, or is even lost permanently, no data are lost. For many applications, this property is extremely desirable.
2. To allow file access to occur even if one file server is down. The motto is : The show must go on. A server crash should not bring the entire system down until the server can be rebooted.
3. To split the workload over multiple servers. As the system grows in size, having all the files on one

server can become a performance bottleneck. By having files replicated on two or more servers, the least heavily loaded one can be used.

The first two relate to improving reliability and availability; the third concerns performance. All are important.

A key issue relating to replication is transparency (as usual). To what extent are the users aware that some files are replicated? Do they play any role in the replication process, or it is handled entirely automatically. At one extreme, the users are fully aware of the replication process and can even control it. At the other, the system does everything behind their backs. In the latter case, we say that the system is replication transparent.

Fig. (3) shows three ways replication can be done. The first way, shown in fig 3 (a), is for the programmer to control the entire process. When a process makes a file, it does so on one specific server. Then it can make additional copies on other servers, if desired. If the directory server permits multiple copies of a file, the network addresses of all copies can then be associated with the file name as shown at the bottom of fig 3 (a), so that when the name is looked up, all copies will be found. When the file is subsequently opened, the copies can be tried sequentially in some order, until an available one is found.

To make the concept of explicit replication more familiar, consider how it can be done in a system based on remote mounting in UNIX.

In Fig 3 (b), there is an alternative approach, lazy replication. Only one copy of each file is created, on some server. Later, the server itself makes replicas on other servers automatically, without the programmer's knowledge. The system must be smart enough to be able to retrieve any of these copies if need be. When making copies in the background like this, it is important to pay attention to the possibility that the file might change before the copies can be made.

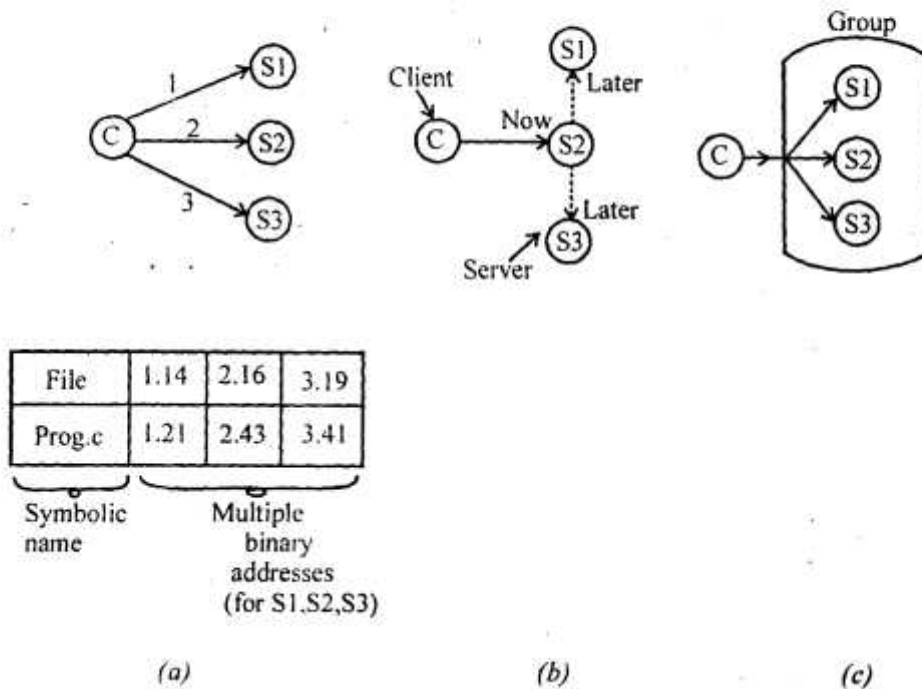


Fig. 3 (a) Explicit file replication (b) Lazy file replication. (c) File replication using a group.

The last method is to use group communication, as shown in fig 3 (c). In this, all WRITE system calls are simultaneously transmitted to all the servers, so extra copies are made at the same time the original is made. There are two principal differences between lazy replication and using a group. First with lazy replication, one server is addressed rather than a group. Second, lazy replication happens in the background, when a server has some free time, whereas when group communication is used, all copies are made at the same time.

Q. 6. What is the importance of consistency in DSM? Discuss :

(a) Strict consistency.

(b) Casual consistency.

Ans. In DSM systems, they have one or more copies of each read-only page and one copy of each writable page. In the simplest implementation, when a writable page is referenced by a remote machine, a trap occurs and the page is fetched. However, if some writable pages are heavily shared, having only a single copy of each one can be a serious performance bottleneck.

Allowing multiple copies eases the performance problem, since it is then sufficient to update any copy, but doing so introduces a new problem : how to keep all the copies consistent. Maintaining perfect consistency is especially painful when the various copies are on different machines that can only communicate by sending messages over a slow (compared to memory speeds) network. In some DSM systems, the solution is to accept less than perfect consistency as the price for better performance. Precisely what consistency means & how it can be relaxed without making programming unbearable is a major issue among DSM researchers.

(a) Strict Consistency :

The most stringent consistency model is called strict consistency. It is defined by the following condition :

Any read to a memory location X returns the value stored by the most recent write operation to X.

This definition is natural and obvious, although it implicitly assumes the existence of absolute global time so that the determination of most recent is unambiguous. Uniprocessors have traditionally observed strict consistency and uniprocessor programmers have come to expect such behaviour as a matter of course. A system on which the program.

```
a = 1; a = 2; print (a);
```

Printed 1 or any value other than 2 would quickly lead to a lot of very agitated programmers.

In a DSM system, the matter is more complicated. Suppose x is a variable stored only on machine B. Imagine that a process on machine A reads x at time T_1 , which means that a message is then sent to B to get x. Slightly later at T_2 a process on B does a write to x. If strict consistency holds, the read should always return the old value regardless of where the machines are and how close T_2 is to T_1 . However, if $T_2 - T_1$ is say, 1 nanosecond and the machines are 3 meters apart, in order to propagate the read request from A to B to get there before the write the signal would have to travel at 10 times the speed of light. To study consistency in detail, we can give examples. To make these examples precise, we need a special notation. In this notation, several processes, P_1 , P_2 and so on can be shown at different heights in the figure. The operations done by each process are shown horizontally, with time increasing to the right. Straight lines separate the processes. The symbols :

$W(x) a$ and $R(y) b$

Mean that a write to x with the value a and a read from of returning b have been done, respectively. The initial value of all variables in these diagrams is assumed to be 0. As an example, in fig 4 (a) P_1 does a write to location x , storing the value 1. Latter, P_2 , reads x and sees the 1. This behaviour is correct for a strictly consistent memory.

In contrast in fig 4 (b), P_2 does a read after the write (possibly only a nanosecond after it, but still after

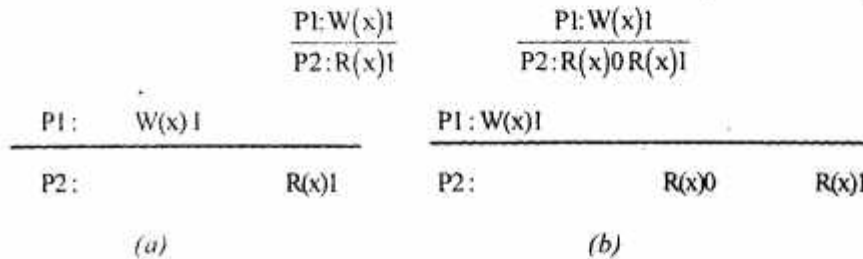


Fig. 4. Behaviour of two processes. The horizontal axis is time.

(a) Strictly consistent memory. (b) Memory that is not strictly consistent.

$i + 1$, and gets 0. A subsequent read gives 1. Such behaviour is incorrect for a strictly memory.

When memory is strictly consistent, all writes are instantaneously visible to all processes and an absolute global time order is maintained. If a memory location is changed, all subsequent reads from that location see the new value, no matter how soon after the change the reads are done and no matter which processes are doing the reading and where they are located. Similarly, if a read is done, it gets the then-current value, no matter how quickly the next write is done.

(b) Casual consistency : The casual consistency model represents a weakening of sequential consistency in that it makes a distinction between events that are potentially casually related and those that are not. To see what causality is all about, consider an example of a memory. Suppose that process P_1 writes a variable x . Then P_2 reads x and writes y . Here the reading of x and the writing of y are potentially casually related because the computation of y may have depended on the value of x read by P_2 (i.e. the value written by P_1). On the other hand, if two processes spontaneously and simultaneously write two variables, these are not casually related. When there is a read followed later by a write, the two events are potentially casually related. Similarly, a read is casually related to the write that provided the data the read got. Operations that are not casually related are said to be concurrent.

For a memory to be considered casually consistent, it is necessary that the memory obey the following condition :

Writes that are potentially casually related must be seen by all processes in the same order. Concurrent writes may be seen in a different order on different machines. As an example of causal consistency, consider Fig 5. Here we have an event sequence that is allowed with casually consistent memory, but which is forbidden with a sequentially consistent memory or a strictly consistent memory. The thing to note is that the writes $W(x)$

2 and $W(x)2$ are concurrent, so it is not required that all processes see them in the same order. If the software fails when different processes see concurrent events in a different order, it has violated the memory contract offered by causal memory.

P_1 :	$W(x)1$		$W(x)3$
P_2 :		$R(x)1$	$W(x)2$
P_3 :		$R(x)1$	$R(x)3$ $R(x)2$
P_4 :		$R(x)1$	$R(x)2$ $R(x)3$

Fig (5) : This sequence is allowed with causally consistent memory, but not with sequentially consistent memory or strictly consistent memory

Now consider a second example. In Fig. 6 (a) we have $W(x)2$ potentially depending on $W(x)1$ because the 2 may be a result of a computation involving the value read by $R(x)1$. The two writes are causally related, so all processes must see them in the same order. Therefore fig 6 (a) is incorrect. On the other hand, in fig 6 (b) the read has been removed, so $W(x)1$ and $W(x)2$ are now concurrent writes. Causal memory does not require concurrent writes to be globally ordered, so fig 6 (b) is correct.

Implementing causal consistency requires keeping track of which processes have seen which writes. It effectively means that a dependency graph of which operation is dependent on which other operations must be constructed and maintained. Doing so involves some overhead.

P_1 :	$W(x)1$	P_1 :	$W(x)1$
P_2 :	$R(x)1$ $W(x)2$	P_2 :	$W(x)1$
P_3 :	$R(x)2$ $R(x)1$	P_3 :	$R(x)2$ $R(x)1$
P_4 :	$R(x)1$ $R(x)2$	P_4 :	$R(x)1$ $R(x)2$
(a)		(b)	

Fig 6. (a) A violation of causal memory. (b) Correct sequence of events in causal memory

Q. 7. (a) MACH supports the concept of a processor set. On what class of machines does this concept make most sense? What is it used for?

Ans. Mach scheduling has been heavily influenced by its goal of running on multiprocessors. Since a single processor system is effectively a special case of a multiprocessor (with only one CPU), focus is on scheduling in multiprocessor system. The CPUS in a multiprocessor can be assigned to processor sets by software. Each CPU belongs to exactly one processor set. Threads can also be assigned to processor sets by software. Thus each processor set has a collection of CPUS at its disposal & a collection of threads that need computing power. The job of the scheduling algorithm is to assign threads to CPUS in a fair and efficient way. For purpose of scheduling, each processor set is a closed world, with its own resources and its own customers,

independent of all the other processor sets.

This mechanism gives processes a large amount of control over their threads. A process can assign an important threads to a processor set with one CPU and no other threads, thus ensuring that the thread runs all the time. It can also dynamically reassign threads to processor sets as the work proceeds, keeping the load balanced. While the average compiler is not likely to use this facility, a database management system or a real-time system might well use it.

Thread scheduling in mach is based on priorities. Priorities are integers from 0 to some maximum (usually 31 or 127), with 0 being the highest priority and 31 or 127 being the lowest priority. This priority reversal comes from UNIX. Each thread has three priorities assigned to it. The first priority is a base priority, which the thread can set itself, within certain limits. The second priority is the lowest numerical value that the thread may set its base priority to. Since using a value gives worse service, a thread will normally. Set its value to the lowest value it is permitted, unless it is trying intentionally to defer to other threads. The third priority is the current priority, used for scheduling purposes. It is computed by the kernel by adding to the base priority a function based on the thread's recent CPU usage.

Mach threads are visible to the kernel, atleast when the model of fig (b) is used. Each thread competes.

Q. 7. (b) Discuss how memory is managed in MACH.

Ans. Mach has a powerful, elaborate & highly flexible memory management system based on paging, including features found in few other operating systems. In particular it separates the machine-dependent parts in an extremely clear and unusual way. This separation makes the memory management for more portable than in other systems. In addition, the memory management system interacts closely with the communication system. The aspect of Mach's memory management that sets it apart from all others is that the code is split into three parts. The first part is the Pnapi module, which runs in the kernel and is concerned with managing the MMU. It sets up the MMU register & hardware page tables, and catches all page faults. This code depends on the MMU architecture and must be rewritten for each new MMU. Mach has to support. The second part, the machine-in dependent kernel code, is concerned with processing page a faults, managing address maps & replacing pages.

The third part of the memory management code runs as a user process called a memory manager or sometimes an external pager.

The third part of the memory management code runs as a user process called a memory manager or sometimes an external pager. It handles the logical part of the memory management system, primarily management of the backing store (disk). The kernel and the memory manager communicate through a well-defined protocol, making it possible for users to write their own memory managers. This division of labor gives users the ability to implement special-purpose paging systems in order to write systems with special requirements. It also has the potential for making the kernel smaller & simpler by moving a large section of the code out into user space. On the other hand, it has the potential for making a it more complicated, since the kernel must protect itself from buggy or malicious memory managers, and with two active entities involved in handling memory, there is now the danger of race conditions for CPU cycles with all other threads, without regard to which threads is in which process. When making scheduling decisions, the kernel does not take into account which thread belongs to which process.

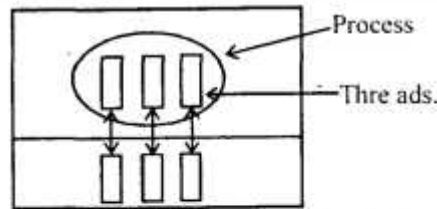


Fig. Each thread has its own kernel thread

Associated with each processor set is an array of run queues as shown in fig. (7). The array has 32 queues, corresponding to threads currently at priorities 0 through 31. When a thread at priority n becomes runnable, it is put at the end of queue x . A thread that is not runnable is not present on any run queue.

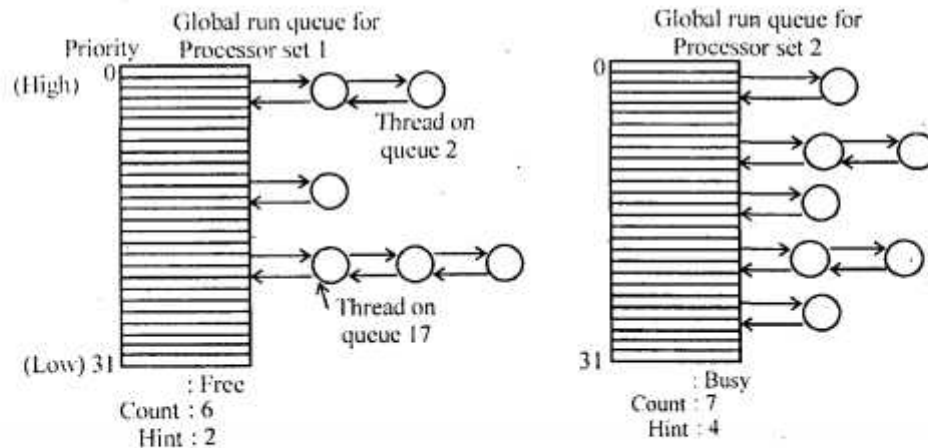


Fig. (7) : The global run queues for a system with two processor sets.

Each run queue has three variables attached to it. The first is a mutex that is used to lock the data structure. It is used to make sure that only one CPU at a time is manipulating the queues. The second variable is the count of the number of threads on all the queues combined. If this count becomes 0, there is no work to do. The third variable is a hint as to where to find the highest-priority thread. This hint allows the search for highest priority thread to avoid the empty queues at the top. Each CPU has its own local run queue. Each local run queue holds those threads that are permanently bound to that CPU. These threads can run on only one CPU.

Q. 8. Write short notes on :

- (i) Remote Procedure calls in DCE.
- (ii) Condition variable in DCE.

Ans. (i) Remote Procedure calls in DCE :

DCE is based on the client/server model. Clients request services by making remote procedure calls to distant servers. The goals of the DCE RPC system are relatively traditional. First and for most, the RPC system makes it possible for a client to access a remote service by simply calling a local procedure. This interface makes it possible for client (i.e. application) programs to be written in a simple way, familiar to most programmers. It also makes it easy to have large volumes of existing code run in a distributed environment with few, if any, changes.

It is upto the RPC system to hide all the details from the clients, and, to some extent, from the servers as well. To start with, the RPC system can automatically locate the correct server and bind to it, without the client having to be aware that this is occurring. It can also handle the message transport in both directions, fragmenting & reassembling them as needed. Finally, the RPC system can automatically handle data type conversions between the client & the server, even if they run on different architectures and have a different byte ordering.

As a consequence of the RPC system's ability to hide the details, clients and servers are highly independent of one another. A client and server can run on different hardware platforms & use different operating systems. A variety of network protocols and data representations are also supported, all without any intervention from the client or server.

Binding a client to a server :

Before a client can call a server, it has to locate the server and bind to it. The main problem in binding is how the client locates the correct server. Broadcasting a message containing the unique identifier to every process in every cell and asking all servers for it to please raise their hands might work, but this approach is so slow and wasteful that it is not practical. Besides, not all networks support broadcasting. Instead, server location is done in two steps :

1. Locate the server's machine.
2. Locate the correct process on that machine.

Different mechanism are used for each of these steps. The need to locate the server's machine is obvious, but the problem with locating the server once the machine is known is more subtle. Basically what it comes down to is that for a client to communicate reliably & securely with a server, a network connection is generally required. Such a connection needs an endpoint, a numerical address on the server's machine to which network connections can be attached and messages sent. Having the server choose a permanent numerical address is risky, since another server on the same machine might accidentally choose other same one. For this reason, endpoints can be dynamically assigned, and a database of (server, endpoint) entries is maintained on each server machine by a process called the RPC daemon. The steps involved in binding are shown in fig.

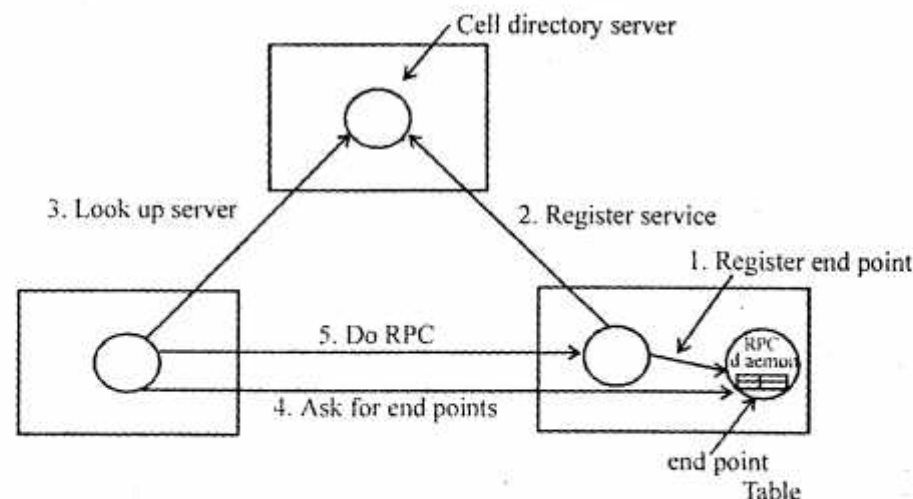


Fig. Client-to-server binding in DCE

Before it becomes available for incoming requests, the server must ask the operating system for an endpoint. It then registers this endpoint with the RRC daemon. The RPC daemon records this information (including which protocols the server speaks) in the endpoint table for future use. The server also registers with some cell directory server, passing it the number of its host.

In the simplest case, at the time of the first RPC, the client stub asks the cell directory server to find it a host running on an instance of the server. The client then goes to the RPC daemon, which has a well-known endpoint, and asks it to look up the endpoint (e.g. TCP port) in its endpoint table. Armed with this information, the RPC can now take place. On subsequent RPCs this lookup is not needed-DCE also gives clients the ability to do more sophisticated searches for a suitable server when that is needed. Authenticated RPC is also an option.

Performing an RPC : The actual RPC is carried out transparently and in the usual way. The client stub marshals the parameters & passes the resulting buffer to the runtime library for transmission using the protocol chosen at binding time. For when a message arrives at the server side, it is routed to the correct server based on the endpoint contained in the incoming message. The runtime library passes the message to the server stub, which unmarshals the parameters & calls the server. The reply goes back by the reverse route.

DCE provides several semantic options. The default is most-once operation in which case no call is ever carried out more than once, even in the face of system crashes. In practice, what this means is that if a server crashes during an RPC & then recovers quickly, the client does not repeat the operation for fear that it might already have been carried out once.

(ii) Condition variable in DCE :

DCE provides two ways for threads to synchronize :

1. Mutexes
2. Condition Variables.

Condition variables provide a synchronization mechanism which are used in conjunction 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 a condition variable which automatically suspends the thread & releases the mutex. Later, when another thread signals the condition variable, the waiting thread is restarted. Mutexes are used when it is essential to prevent multiple threads from accessing the same resource at the time. For eg. when moving items around on a linked list, partway through the move, the list will be in an inconsistent state. To prevent disaster, when one thread is manipulating the list, all other threads must be kept away. By requiring a thread to first successfully lock the mutex associated with the list before touching the list (and unlock it afterward), correct operation can be ensured.