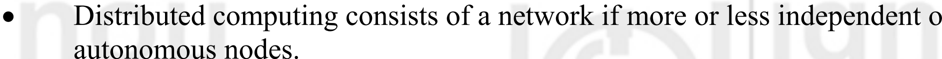
UNIT 2 DISTRIBUTED OPERATING SYSTEMS

Structure **Page Nos.**

2.0 Introduction 17
2.1 Objectives 17
2.2 History of Distributed Computing 17
2.2.1 Workstations – Networks
2.2.2 Wide-Scale Distributed Computing and the Internet
2.3 Distributed Systems 19
2.4 Key Features and Advantages of a Distributed System 20
2.4.1 Advantages of Distributed Systems Over Centralised Systems
2.4.2 Advantages of Distributed Systems over Isolated PC's
2.4.3 Disadvantages of Distributed Systems
2.5 Design Goals of Distributed Systems 21
2.5.1 Concurrency
2.5.2 Scalability
2.5.3 Openness
2.5.4 Fault Tolerance
2.5.5 Privacy and Authentication
2.5.6 Transparency
2.6 Design Issues Involved in Distributed Systems 23
2.6.1 Naming
2.6.2 Communication
2.6.3 Software Structure
2.6.4 Workload Allocation
2.6.5 Consistency Maintenance
2.6.6 Lamport's Scheme of Ordering of Events
2.7 Distributed System Structure 26
2.7.1 Application Model 1: Client Server
2.7.2 Application Model 2: Distributed Objects
2.7.3 Application Model 3: Distributed Shared Memory
2.8 Mutual Exclusion in Distributed Systems 29
2.8.1 Mutual Exclusion Servers
2.8.2 Token Based Mutual Exclusion
2.8.3 Lamport's Bakery Algorithm
2.8.4 Ricart and Agrawala's Mutual Exclusion Algorithm
2.9 Remote Procedure Calls 35
2.9.1 How RPC Works?
2.9.2 Remote Procedure Calling Mechanism
2.9.3 Implementation of RPC
2.9.4 Considerations for Usage
2.9.5 Limitations
2.10 Other Middleware Technologies 37
2.11 Summary 38
2.12 Solutions /Answers 39
2.13 Further Readings 39

2.0 INTRODUCTION

In the earlier unit we have discussed the Multiprocessor systems. In the process coupling we have come across the tightly coupled systems and loosely coupled systems. In this unit we concentrate on the loosely coupled systems called as the distributed systems. Distributed computing is the process of aggregating the power of several computing entities to collaboratively run a computational task in a transparent and coherent way, so that it appears as a single, centralised system.
The easiest way of explaining what distributed computing is all about, is by naming a few of its properties:



- Distributed computing consists of a network if more or less independent or autonomous nodes.
- The nodes do *not* share primary storage (i.e., RAM) or secondary storage (i.e., disk) - in the sense that the nodes cannot directly access another node's disk or RAM. Think about it in contrast to a multiprocessors machine where the different "nodes" (CPUs)) share the same RAM and secondary storage by using a common bus.
- A well designed distributed system does not crash if a node goes down.
- If you are to perform a computing task which is parallel in nature, scaling your system is a lot cheaper by adding extra nodes, compared to getting a faster single machine. Of course, if your processing task is highly non-parallel (every result depends on the previous), using a distributed computing system may not be very beneficial.

Before going into the actual details of the distributed systems let us study how distributed computing evolved.

2.1 OBJECTIVES

After going through this unit, you should be able to:

- define a distributed system, key features and design goals of it;
- explain the design issues involved in the distributed systems;
- describe the models of distributed system structure;
- explain the mutual exclusion in distributed systems, and
- define RPC, its usage and limitations.

2.2 HISTORY OF DISTRIBUTED COMPUTING

Distributed computing began around 1970 with the emergence of two technologies:

- minicomputers, then workstations, and then PCs.
- computer networks (eventually Ethernet and the Internet).

With the minicomputer (e.g., Digital's PDP-11) came the timesharing operating system (e.g., MULTICS, Unix, RSX, RSTS) - many users using the same machine but it looks to the users as if they each have their own machine.

The problem with mini-computers was that they were slower than the mainframes made by IBM, Control Data, Univac, etc. As they became popular they failed to scale to large number of users as the mainframes could. The way to scale mini-computers was to buy more of them. The trend toward cheaper machines made the idea of having many minis as a feasible replacement for a single mainframe and made it possible to contemplate a future computing environment where every user had their own computer on their desk, which is a computer workstation.

Work on the first computer workstation began in 1970 at Xerox Corporation's Palo Alto Research Center (PARC). This computer was called the Alto. Over the next 10 years, the computer system's group at PARC would invent almost everything that's interesting about the computer workstations and personal computers we use today. The Alto introduced ideas like the workstation, bit-mapped displays (before that computer interfaces were strictly character-based) and the mouse.

Other innovations that came from PARC from 1970 to 1980 include Ethernet, the first local-area network, window- and icon-based computing (the inspiration for the Apple Lisa, the progenitor of the Macintosh, which in turn inspired Microsoft Windows and IBM OS/2 came from a visit to PARC), the first distributed fileserver XDFS , the first

print server, one of the first distributed services: Grapevine (a messaging and authentication system), object-oriented programming (Smalltalk) and Hoare's condition variables and monitors were invented as part of the Mesa programming language used in the Cedar system.

2.2.1 Workstations–Networks

The vision of the PARC research (and the commercial systems that followed) was to replace the timesharing mini-computer with single-user workstations. The genesis of this idea is that it made computers (i.e., workstations and PCs) into a commodity item that like a TV or a car could be produced efficiently and cheaply. The main costs of a computer are the engineering costs of designing it and designing the manufacturing process to build it. If you build more units, you can amortised the engineering costs better and thus make the computers cheaper. This is a very important idea and is the main reason that distribution is an excellent way to build a cheap scalable system.

2.2.2 Wide-Scale Distributed Computing and the Internet

Another type of distributed computing that is becoming increasingly important today is the sort of wide-scale distribution possible with the Internet.

At roughly the same time as the workstation and local-area network were being invented, the U.S. Department of Defence was putting tons of U.S. taxpayer money to work to set up a world-wide communication system that could be used to support distributed science and engineering research needed to keep the Defence Dept. supplied with toys. They were greatly concerned that research assets not be centrally located, as doing so would allow one well-placed bomb to put them out of business. This is an example of another benefit of distributed systems: fault tolerance through replication.

2.3 DISTRIBUTED SYSTEMS

Multiprocessor systems have more than one processing unit sharing memory/peripheral devices. They have greater computing power, and higher reliability. Multiprocessor systems are classified into two:

- **Tightly-coupled**: Each processor is assigned a specific duty but processors work in close association, possibly sharing one memory module.

- **Loosely-coupled (distributed)**: Each processor has its own memory and copy of the OS.

A distributed computer system is a loosely coupled collection of autonomous computers connected by a network using system software to produce a single integrated computing environment.

A distributed operating system differs from a network of machines each supporting a network operating system in only one way: The machines supporting a distributed operating system are all running under a single operating system that spans the network. Thus, the print spooler might, at some instant, be running on one machine, while the file system is running on others, while other machines are running other parts of the system, and under some distributed operating systems, these software parts may at times migrate from machine to machine.

With network operating systems, each machine runs an entire operating system. In contrast, with distributed operating systems, the entire system is itself distributed across the network. As a result, distributed operating systems typically make little distinction between remote execution of a command and local execution of that same command. In theory, all commands may be executed anywhere; it is up to the system to execute commands where it is convenient.

2.4 KEY FEATURES AND ADVANTAGES OF A DISTRIBUTED SYSTEM

The following are the key features of a distributed system:

- They are Loosely coupled
  - remote access is many times slower than local access
- Nodes are autonomous
  - workstation resources are managed locally
- Network connections using system software
  - remote access requires explicit message passing between nodes
  - messages are CPU to CPU
  - protocols for reliability, flow control, failure detection, etc., implemented in software
  - the *only* way two nodes can communicate is by sending and receiving network messages−this differs from a hardware approach in which hardware signalling can be used for flow control or failure detection.

2.4.1 Advantages of Distributed Systems over Centralised Systems

- Better price/performance than mainframes
- More computing power
  - sum of the computing power of the processors in the distributed system may be greater than any single processor available (parallel processing)
- Some applications are inherently distributed
- Improved reliability because system can survive crash of one processor
- Incremental growth can be achieved by adding one processor at a time
- Shared ownership facilitated.

2.4.2 Advantages of Distributed Systems over Isolated PCs

- Shared utilisation of resources.
- Communication.
- Better performance and flexibility than isolated personal computers.
- Simpler maintenance if compared with individual PC's.

2.4.3 Disadvantages of Distributed Systems

Although we have seen several advantages of distributed systems, there are certain disadvantages also which are listed below:

- Network performance parameters.
- *Latency*: Delay that occurs after a send operation is executed before data starts to arrive at the destination computer.
- *Data Transfer Rate*: Speed at which data can be transferred between two computers once transmission has begun.
- *Total network bandwidth*: Total volume of traffic that can be transferred across the network in a give time.
- Dependency on reliability of the underlying network.
- Higher security risk due to more possible access points for intruders and possible communication with insecure systems.
- Software complexity.

2.5 DESIGN GOALS OF DISTRIBUTED SYSTEMS

In order to design a good distributed system. There are six key design goals. They are:

- Concurrency
- Scalability
- Openness
- Fault Tolerance
- Privacy and Authentication
- Transparency.

Let us discuss them one by one.

2.5.1 Concurrency

A server must handle many client requests at the same time. Distributed systems are naturally concurrent; that is, there are multiple workstations running programs independently and at the same time. Concurrency is important because any distributed service that isn't concurrent would become a bottleneck that would serialise the actions of its clients and thus reduce the natural concurrency of the system.

2.5.2 Scalability

The goal is to be able to use the same software for different size systems. A distributed software system is scalable if it can handle increased demand on any part of the system (i.e., more clients, bigger networks, faster networks, etc.) without a change to the software. In other words, we would like the engineering impact of increased demand to be proportional to that increase. Distributed systems, however, can be built for a very wide range of scales and it is thus not a good idea to try to build a system that can handle everything. A local-area network file server should be built differently from a Web server that must handle millions of requests a day from throughout the world. The key goal is to understand the target system's expected size and expected growth and to understand how the distributed system will scale as the system grows.

2.5.3 Openness

Two types of openness are important: non-proprietary and extensibility.
Public protocols are important because they make it possible for many software manufacturers to build clients and servers that will be able to talk to each other. Proprietary protocols limit the "players" to those from a single company and thus limit the success of the protocol.

A system is extensible if it permits customisations needed to meet unanticipated requirements. Extensibility is important because it aids scalability and allows a system to survive over time as the demands on it and the ways it is used change.

2.5.4 Fault Tolerance

It is critically important that a distributed system be able to tolerate "partial failures". Why is it so important? Two reasons are as follows:

- *Failures are more harmful:* Many clients are affected by the failure of a distributed service, unlike a non-distributed system in which a failure affects only a single node.
- *Failures are more likely:* A distributed service depends on many components (workstation nodes, network interfaces, networks, switches, routers, etc.) all of which must work. Furthermore, a client will often depend on multiple distributed services (e.g., multiple file systems or databases) in order to function properly. The probability that such a client will experience a failure can be approximated as the sum of the individual failure probabilities of everything that it depends on. Thus, a client that depends on N components (hardware or software) that each have failure probability P will fail with probability roughly $N*P$. (This approximation is valid for small values of P. The exact failure probability is (1−(1−P)^N).)

There are two aspects of failure tolerance to be studied as shown below:

Recovery

- A failure shouldn't cause the loss (or corruption) of critical data, or computation.
- After a failure, the system should recover critical data, even data that was being modified when the failure occurred. Data that survives failures is called "persistent" data.
- Very long-running computations must also be made recoverable in order to restart them where they left off instead of from the beginning.
- For example, if a fileserver crashes, the data in the file system it serves should be intact after the server is restarted.

Availability

- A failure shouldn't interrupt the service provided by a critical server.
- This is a bit harder to achieve than recovery. We often speak of a highly-available service as one that is almost always available even if failures occur.
- The main technique for ensuring availability is service replication.
- For example, a fileserver could be made highly available by running two copies of the server on different nodes. If one of the servers fails, the other should be able to step in without service interruption.

2.5.5 Privacy and Authentication

Privacy is achieved when the sender of a message can control what other programs (or people) can read the message. The goal is to protect against eavesdropping. For example, if you use your credit card to buy something over the Web, you will probably want to prevent anyone but the target Web server from reading the message that contains your credit card account number. Authentication is the process of ensuring that programs can know who they are talking to. This is important for both clients and servers.

For clients authentication is needed to enable a concept called trust. For example, the fact that you are willing to give your credit card number to a merchant when you buy something means that you are implicitly trusting that merchant to use your number according to the rules to which you have both agreed (to debit your account for the amount of the purchase and give the number to no one else). To make a Web purchase, you must trust the merchant's Web server just like you would trust the merchant for an in-person purchase. To establish this trust, however, you must ensure that your Web browser is really talking to the merchant's Web server and not to some other program that's just pretending to be their merchant.

For servers authentication is needed to enforce access control. For a server to control who has access to the resources it manages (your files if it is a fileserver, your money if it is a banking server), it must know who it is talking to. A Unix login is a crude example of an authentication used to provide access control. It is a crude example because a remote login sends your username and password in messages for which privacy is not guaranteed. It is thus possible, though usually difficult, for someone to eavesdrop on those messages and thus figure out your username and password.

For a distributed system, the only way to ensure privacy and authentication is by using cryptography.

2.5.6 Transparency

The final goal is transparency. We often use the term **single system image** to refer to this goal of making the distributed system look to programs like it is a tightly coupled (i.e., single) system.

This is really what a distributed system software is all about. We want the system software (operating system, runtime library, language, compiler) to deal with all of the complexities of distributed computing so that writing distributed applications is as easy as possible.

Achieving complete transparency is difficult. There are eight types, namely:

- ***Access Transparency*** enables local and remote resources to be accessed using identical operations
- ***Location Transparency*** enables resources to be accessed without knowledge of their (physical) location. Access transparency and location transparency are together referred to as network transparency.
- ***Concurrency Transparency*** enables several processes to operate concurrently using shared resources without interference between them.
- ***Replication Transparency*** enables multiple instances of resources to be used to increase reliability and performance without knowledge of the replicas by users or application programmers.
- ***Failure Transparency*** enables the concealment of faults, allowing users and application programs to complete their tasks despite the failure of hardware or software components.
- ***Mobility Transparency*** allows the movement of resources and clients within a system without affecting the operation of users or programs.
- ***Performance Transparency*** allows the system to be reconfigured to improve performance as loads change
- ***Scaling Transparency*** Transparency allows the system and applications to expand in scale without change to the system structure or the application algorithms

2.6 DESIGN ISSUES INVOLVED IN DISTRIBUTED SYSTEMS

There are certain design issues to be considered for distributed systems. They are:

a) Naming
b) Communication
c) Software Structure
d) Workload Allocation
e) Consistency Maintenance

Let us discuss the issues briefly:

2.6.1 Naming

- A name is a string of characters used to identify and locate a distributed resource.
- An identifier is a special kind of name that is used directly by the computer to access the resource.

  For example the identifier for a Unix server would include at least (1) an IP address and (2) a port number. The IP address is used to find the node that runs the server and the port number identifies the server process on that node.
- Resolution is the process of turning a name into an identifier. Resolution is performed by a Name Server. It is also called “binding” (as in binding a name to a distributed service).

For example, an IP domain name (e.g., cs.ubc.ca) is turned into a IP address by the IP Domain Name Server (DNS), a distributed hierarchical server running in the Internet.

2.6.2 Communication

- Getting different processes to talk to each other
- Messages
- Remote method invocation.

2.6.3 Software Structure

- The main issues are to choose a software structure that supports our goals, particularly the goal of openness.
- We thus want structures that promote extensibility and otherwise make it easy to program the system.
- Alternatives are:
  - A monolithic structure is basically a big pile of code; it is not so desirable because it is hard to extend or reason about a system like that.
  - A modular structure divides the system into models with well-defined interfaces that define how the models interact. Modular systems are more extensible and easier to reason about than monolithic systems.
  - A layered structure is a special type of modular structure in which modules are organised into layers (one on top of the other). The interaction between layers is restricted such that a layer can communicate directly with only the layer immediately above and the layer immediately below it. In this way, each layer defines an abstraction that is used by the layer immediately above it. Clients interact only with the top layer and only the bottom layer deals with the hardware (e.g., network, disk, etc.) Network protocols have traditionally been organised as layers (as we will see in the next class) and for this reason we often refer to these protocols as "protocol stacks".
  - Operating systems can be either monolithic (e.g., UNIX and Windows) or modular (e.g., Mach and Windows NT). A modular operating system is called a micro kernel. A micro-kernel OS has a small minimalist kernel that includes as little functionality as possible. OS services such as VM, file systems, and networking are added to the system as separate servers and they reside in their own user-mode address spaces outside the kernel.

Let us study more details regarding the software structure in section 2.7 of this unit.

2.6.4 Workload Allocation

- The key issue is load balancing: The allocation of the network workload such that network resources (e.g., CPUs, memory, and disks) are used efficiently.

For CPUs, there are two key approaches. They are:

Processor Pools

Put all of the workstations in a closet and give the users cheap resource-poor X terminals. User processes are allocated to the workstations by a distributed operating system that controls all of the workstations. Advantage is that process scheduling is simplified and resource utilisation more efficient because workstations aren't "owned" by anyone. The OS is free to makes scheduling decisions based solely on the goal of getting the most work done. Disadvantage is that, these days, powerful workstations aren't much more expensive than cheap X terminals. So if each user has a powerful workstation instead of a cheap X terminal, then we don't need the machines in the closet and we thus have the idle-workstation model described below:

Idle Workstations

Every user has a workstation on their desk for doing their work. These workstations are powerful and are thus valuable resources. But, really people don't use their workstations all of the time. There's napping, lunches, reading, meetings, etc. - lots of times when workstations are **idle**. In addition, when people are using their workstation, they often don't need all of its power (e.g., reading mail doesn't require a 200-Mhz CPU and 64-Mbytes of RAM). The goal then is to move processes from active workstations to idle workstations to balance the load and thus make better use of network resources. But, people "own" (or at least feel like they own) their workstations. So a key issue for using idle workstations is to avoid impacting workstation users (i.e., we can't slow them down). So what happens when I come back from lunch and find your programs running on my machine? I'll want your processes moved elsewhere NOW. The ability to move an active process from one machine to another is called **process migration**. Process migration is necessary to deal with the user-returning-from-lunch issue and is useful for rebalancing network load as the load characterises a network change over time.

2.6.5 Consistency Maintenance

The final issue is consistency. There are four key aspects of consistency: atomicity, coherence, failure consistency, and clock consistency.

Atomicity

- The goal is to provide the "all-or-nothing" property for sequences of operations.
- Atomicity is important and difficult for distributed systems because operations can be messages to one or more servers. To guarantee atomicity thus requires the coordination of several network nodes.

Coherence

Coherence is the problem of maintaining the consistency of replicated data.
We have said that replication is a useful technique for (1) increasing availability and (2) improving performance.

When there are multiple copies of an object and one of them is modified, we must ensure that the other copies are updated (or invalidated) so that no one can erroneously read an out-of-date version of the object.

A key issue for providing coherence is to deal with the event-ordering problem.

Failure Consistency

- We talked last time about the importance of the goal of failure tolerance. To build systems that can handle failures, we need a model that clearly defines what a failure is. There are two key types of failures: *Fail-stop and Byzantine*.
- ***Fail-stop*** is a simplified failure model in which we assume that the only way a component will fail is by stopping. In particular, a will never fail by giving a wrong answer.

- ***Byzantine*** failure is a more encompassing model that permits failures in which a failed node might not stop but might just start giving the wrong answers. Techniques for dealing with Byzantine failure typically involve performing a computation redundantly (and often in different ways with different software). The system then compares all of the answers generated for a given computation and thus detects when one component starts giving the wrong answer. The ability to detect Byzantine failures is important for many safety-critical systems such as aircraft avionics.
- In this class we will assume failures are Fail-Stop; this is a common assumption for distributed systems and greatly simplifies failure tolerance.

Clock Consistency

- Maintaining a consistent view of time needed to order network events is critical and challenging. This becomes further difficult when the nodes of the distributed system are geographically apart. Let us study the scheme provided by Lamport on Ordering of events in a distributed environment in the next section.

2.6.6 Lamport's Scheme of Ordering of Events

Lamport proposed a scheme to provide ordering of events in a distributed environment using logical clocks. Because it is impossible to have perfectly synchronized clocks and global time in a distributed system, it is often necessary to use logical clocks instead.

Definitions:

Happened Before Relation (->): This relation captures causal dependencies between events, that is, whether or not events have a cause and effect relation. This relation (->) is defined as follows:

- a -> b, if *a* and *b* are in the same process and *a* Occurred before *b*.
- a -> b, if *a* is the event of sending a message and *b* is the receipt of that message by another process.

If a -> b and b -> c, then a -> c, that is, the relation has the property of transitivity.

Causally Related Events: If *event a -> event b*, then *a* casually affects *b*.

Concurrent Events: Two distinct events *a* and *b* are concurrent (*a* || *b*) if (not) *a -> b* and (not) b -> a. That is, the events have no causal relationship. This is equivalent to b || a.

For any two events *a and b* in a system, only one of the following is true: a -> b, b -> a, or a || b.

Lamport introduced a system of logical clocks in order to make the -> relation possible. It works like this: Each process P_i in the system has its own clock C_i. C_i can be looked at as a function that assigns *a* number, $C_i(a)$ to an event *a*. This is the timestamp of the event *a* in process P_i. These numbers are not in any way related to physical time -- that is why they are called logical clocks. These are generally implemented using counters, which increase each time an event occurs. Generally, an event's timestamp is the value of the clock at the time it occurs.

Conditions Satisfied by the Logical Clock system:

For any events *a* and *b*, if a -> b, then $C(a) < C(b)$. This is true if two conditions are met:

- If a occurs before b, then $C_i(a) < C_i(b)$.
- If a is a message sent from P_i and b is the receipt of that same message in P_j, then $C_i(a) < C_j(b)$.

Implementation Rules Required:

Clock C_i is incremented for each event: $C_i := C_i + d \; (d > 0)$
if *a* is the event of sending a message from one process to another, then the receiver sets its clock to the max of its current clock and the sender's clock - that is, $C_j := \max(C_j, t_m + d) \; (d > 0)$.

2.7 DISTRIBUTED SYSTEM STRUCTURE

There are three main alternative ways to structure a distributed application. Each of them defines a different **application model** for writing distributed programs.

The three distributed system application models are:

- Client / Server (and RPC)
- Distributed objects
- Distributed shared memory.

Each model requires a different set of "system" software to support applications written in that style. This system software consists of a combination of operating system and runtime-library code and support from languages and compilers. What all of this code does is translate the distributed features of the particular application model into the sending and receiving of network messages. Because, remember that the only way that nodes (workstations/PCs) can communicate in a distributed system is by sending and receiving messages.

2.7.1 Application Model 1: Client Server

- A designated node exports a "service"; this node is called the "server".
- Nodes that import and use the service are called "clients".
- Clients communicate with servers using a "request-response" protocol.

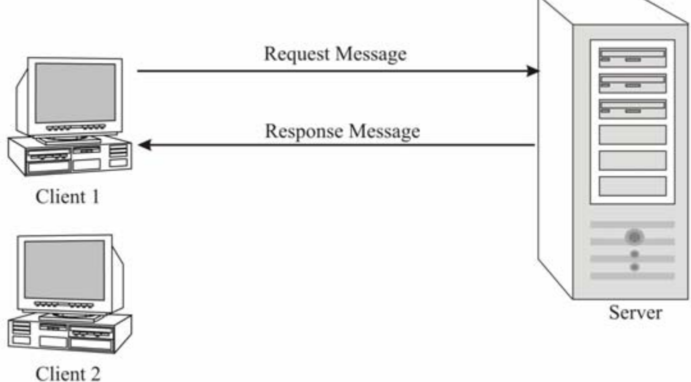


Figure 1: Client Server Model

- Request-response differs from other protocols.
  - Peer-to-peer protocols don't have a designated server.
  - Streaming protocols such as TCP/IP work differently as we will see a couple of classes from now.
- We should really use the terms client and server to refer to the pieces of the software that implement the server and not to the nodes themselves. Why? Because a node can actually be both a client and server. Consider NFS for example. One workstation might export a local filesystem to other nodes (i.e., it is a server for the file system) and import other filesystems exported by remote nodes (i.e., it is a client for these filesystems).
- Application interface can be either send/receive or RPC
  - **Send/receive :** The Unix socket interface is an example. Clients communicate with server by building a message (a sequence of characters) that it then explictly sends to the server. Clients receive messages by issuing a "receive" call.
    We'll talk about this on Friday.
  - **RPC :** A remote procedure call interface allows a client to request a server operation by making what looks like a procedure call. The system software translates this RPC into a message send to the server and a message receives to wait for the reply. The advantages of RPC over send/receive is that the system hides some of the details of message passing from programs: it's easier to call a procedure than format a

message and then explicitly send it and wait for the reply. (Study more on RPC given in section 2.9 of this unit).

2.7.2 Application Model 2: Distributed Objects

- Similar to RPC-based client-server.
- Language-level objects (e.g., C++, Java, Smalltalk) are used to encapsulate data and functions of a distributed service.
- Objects can reside in either the memory of a client or a server.
- Clients communicate with servers through objects that they "share" with the server. These shared objects are located in a client's memory and look to the client just like other "local" objects. We call these objects "remote" objects because they represent a remote service (i.e., something at the server). When a client invokes a method of a remote object, the system might do something locally or it might turn the method invocation into an RPC to the server. (Recall that in the object-oriented world, procedures are called methods and procedure callls are called method invocations).

- There are two main advantages of Distributed Objects over RPC: (1) objects hide even more of the details of distribution than RPC (more next week) and (2) objects allow clients and servers to communicate using in two different ways: function shipping (like RPC) and data shipping.
    - o ***function shipping*** means that a client calls the server and asks it to perform a function; this is basically like RPC.
    - o ***data shipping*** means that the server sends some data to the client (stores it as part of an object) and the client then performs subsequent functions locally instead of having to call the server every time it wants to do something.

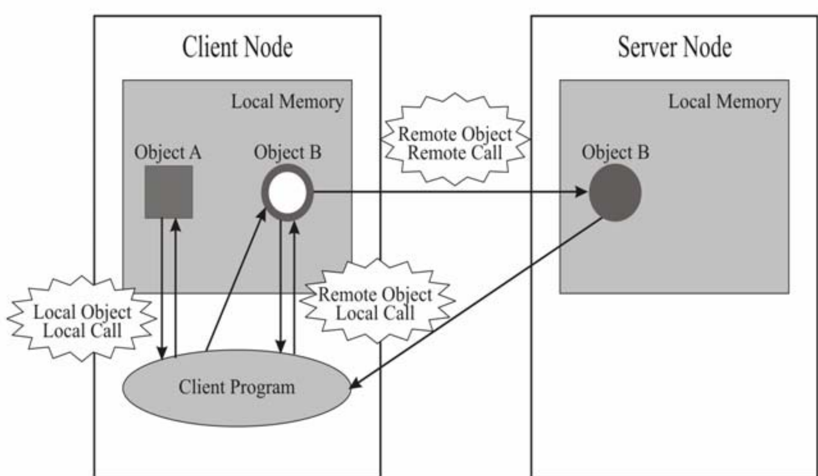


Figure 2: Distributed objects models

- For example, in the picture above, a client program running can access local object A and remote object B (controlled by the server node) in exactly the same way; the arrows show flow of a procedure call into the object and a return from it. The server can design B so that calls are turned into remote calls just like RPC (red) or it can ship some data to the client with the client's copy of the object and thus allow some calls to run locally on the client (orange).
- More on distributed objects later in the course.

2.7.3 Application Model 3: Distributed Shared Memory

- Usually used for peer-to-peer communication instead of client-server.
- Clients communicate with each other through shared variables.

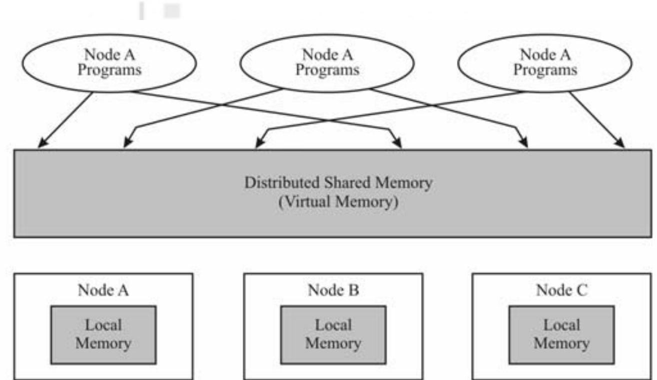


Figure 3: Distributed shared memory model

- The system implements the illusion of shared memory and translates accesses to shared data into the appropriate messages.
- To a program, the distributed system looks just like a shared-memory multiprocessor.
- The advantage is that it is really easy to program: hides all of the details of distribution.
- The disadvantage is that it often hides too much. As far as a program knows, everything in its memory is local. But really, some parts of its memory are stored on or shared with a remote node. Access to this remote data is very SLOW compared with accessing local data. For good performance, a program usually needs to know what data is local and what data is remote.
- Another disadvantage is that it is very complicated to implement a distributed-shared memory system that works correctly and performs well.
- We'll talk more about distributed shared memory towards the end of the term.

2.8 MUTUAL EXCLUSION IN DISTRIBUTED SYSTEMS

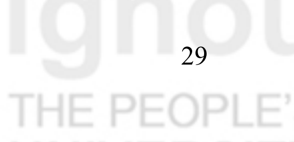
When all processes sharing a resource are on the same machine, mutual exclusion is easily assured by the use of a semaphore, spin-lock or other similar shared abstraction. When the processes involved are on different machines, however, mutual exclusion becomes more difficult.

Consider the following example: A number of machines in a network are competing for access to a printer. The printer is so constructed that every line of text sent to the printer must be sent in a separate message, and thus if a process wants to print an entire file, it must obtain exclusive use of the printer, somehow, send all the lines of the file it wishes to print, and then release the printer for use by others on the network.

A trivial solution to this problem is to install a print spooler process somewhere on the net. That print spooler would gather lines of files provided by various applications processes, maintain a queue of completed files that are ready to print, and print one such file at a time. This works, but it introduces some problems: First, the spooler must have buffer capacity to hold the aggregate of all the files that have not yet been printed. Second, the spooler may become a bottleneck, limiting the performance of the entire system, and third, if the processor supporting the spooler is unreliable, the spooler may limit the reliability of the system.

2.8.1 Mutual Exclusion Servers

In the printer example being used here, the problem of storage space in the spooler typically becomes acute with graphics printers. In such a context, it is desirable to block an applications process until the printer is ready to accept data from that applications process, and then let that process directly deliver data to the printer.



For example, an applications process may be coded as follows:

```
Send request for permission to the spooler
Await reply giving permission to print
Loop
  send data directly to printer
End Loop

Send notice to spooler that printing is done
```

If all users of the spooler use this protocol, the spooler is no longer serving as a spooler, it is merely serving as a mutual exclusion mechanism! In fact, it implements exactly the same semantics as a binary semaphore, but it implements it using a client server model.

The process implementing a semaphore using message passing might have the following basic structure:

```
Loop
  Await message from client
  Case message type of
    P:
      If count > 0
              Send immediate reply
              count = count - 1
      Else
        Enqueue identity of client
      End if
    V:
      If queue is empty,
              count = count + 1
      Else
        Dequeue one blocked client
        Send a delayed reply
      End if
  End case
End Loop
```

This requires a count and a queue of return addresses for each semaphore. Note that, by presenting this, we have proven that blocking message passing can be used to implement semaphores. Since we already know that semaphores plus shared memory are sufficient to implement blocking message passing, we have proven the equivalence, from a computation theory viewpoint, of these two models of interprocess communication.

The disadvantage of implementing semaphores using a server process is that server becomes a potential source of reliability problems. If we can build a mutual exclusion algorithm that avoids use of a dedicated server, for example, by having the processes that are competing for entry to a critical section negotiate directly with each other, we can potentially eliminate the reliability problem.

2.8.2 Token Based Mutual Exclusion

One alternative to the mutual-exclusion server given above is to arrange the competing processes in a ring and let them exchange a token. If a process receives the token and does not need exclusive use of the resource, it must pass the token on to the next process in the ring. If a process needs exclusive use of the resource, it waits for the token and then holds it until it is done with the resource, at which point it puts the token back in circulation.

This is the exact software analog of a token ring network. In a token ring network, only one process at a time may transmit, and the circulating token is used to assure

this, exactly as described. The token ring network protocol was developed for the hardware level or the link-level of the protocol hierarchy. Here, we are proposing building a virtual ring at or above the transport layer and using essentially the same token passing protocol.

This solution is not problem free. What if the token is lost? What if a process in the ring ceases transmission? Nonetheless, it is at the root of a number of interesting and useful distributed mutual exclusion algorithms. The advantage of such distributed algorithms is that they do not rest on a central authority, and thus, they are ideal candidates for use in fault tolerant applications.

An important detail in the token-based mutual exclusion algorithm is that, on receiving a token, a process *must* immediately forward the token if it is not waiting for entry into the critical section. This may be done in a number of ways:

- Each process could periodically check to see if the token has arrived. This requires some kind of non-blocking read service to allow the process to poll the incoming network connection on the token ring. The UNIX FNDELAY flag allows non-blocking read, and the Unix select() kernel call allows testing an I/O descriptor to see if a read from that descriptor would block; either of these is sufficient to support this polling implementation of the token passing protocol. The fact that UNIX offers two such mechanisms is good evidence that these are afterthoughts added to Unix after the original implementation was complete.
- The receipt of an incoming token could cause an interrupt. Under UNIX, for example, the SIGIO signal can be attached to a socket or communications line (see the FASYNC flag set by fcntl). To await the token, the process could disable the SIGIO signal and do a blocking read on the incoming token socket. To exit the critical section, the process could first enable SIGIO and then send the token. The SIGIO handler would read the incoming token and forward it before returning.
- A thread or process could be dedicated to the job of token management. We'll refer to such a thread or process as the mutual exclusion agent of the application process. Typically, the application would communicate with its agent using shared memory, semaphores, and other uni-processor tools, while the agent speaks to other agents over the net. When the user wants entry to the critical section, it sets a variable to "let me in" and then does a wait on the entry semaphore it shares with the agent. When the user is done, the user sets the shared variable to "done" and then signals the go-on semaphore it shares with the agent. The agent always checks the shared variable when it receives the token, and only forwards it when the variable is equal to "done".

2.8.3 Lamport's Bakery Algorithm

One decentralised algorithm in common use, for example, in bakeries, is to issue numbers to each customer. When the customers want to access the scarce resource (the clerk behind the counter), they compare the numbers on their slips and the user with the lowest numbered slip wins.

The problem with this is that there must be some way to distribute numbers, but this has been solved. In bakeries, we use a very small server to distribute numbers, in the form of a roll of tickets where conflicts between two customers are solved by the fact that human hands naturally exclude each other from the critical volume of space that must be occupied to take a ticket. We cannot use this approach for solving the problem on a computer.

Before going on to more interesting implementations for distributing numbers, note that clients of such a protocol may make extensive use of their numbers! For example, if the bakery contains multiple clerks, the clients could use their number to select a clerk (number modulo number of clerks). Similarly, in a FIFO queue implemented with a bounded buffer, the number modulo the queue size could indicate the slot in the

buffer to be used, allowing multiple processes to simultaneously place values in the queue.

Lamport's Bakery Algorithm provides a decentralised implementation of the "take a number" idea. As originally formulated, this requires that each competing process share access to an array, but later distributed algorithms have eliminated this shared data structure. Here is the original formulation:

For each process, *i*, there are two values, C[i] and N[i], giving the status of process I and the number it has picked. In more detail:

```
N[i] = 0 --> Process i is not in the bakery.
N[i] > 0 --> Process i has picked a number and is in the bakery.

C[i] = 0 --> Process i is not trying to pick a number.
C[i] = 1 --> Process i is trying to pick a number.

when

   N[i] = min( for all j, N[j] where N[j] > 0 )
   Process i is allowed into the critical section.
```

Here is the basic algorithm used to pick a number:

```
C[i] := 1;
N[i] := max( for all j, N[j] ) + 1;
C[i] := 0;
```

In effect, the customer walks into the bakery, checks the numbers of all the waiting customers, and then picks a number one larger than the number of any waiting customer.

If two customers each walk in at the same time, they are each likely to pick the same number. Lamport's solution allows this but then makes sure that customers notice that this has happened and break the tie in a sensible way.

To help the customers detect ties, each customer who is currently in the process of picking a number holds his hand up (by setting C[i] to 1. s/he pulls down his hand when s/he is done selecting a number -- note that selecting a number may take time, since it involves inspecting the numbers of everyone else in the waiting room.

A process does the following to wait for the baker:

```
Step 1:
while (for some j, C(j) = 1) do nothing;
```

First, wait until any process which might have tied with you has finished selecting their numbers. Since we require customers to raise their hands while they pick numbers, each customer waits until all hands are down after picking a number in order to guarantee that all ties will be cleanly recognised in the next step.

```
Step 2:
 repeat
   W := (the set of j such that N[j] > 0)
    (where W is the set of indeces of waiting processes)
   M := (the set of j in W
          such that N[j] <= N[k]
          for all k in W)
   (where M is the set of process indices with minimum numbers)
   j := min(M)
    (where  is in M and the tie is broken)
 until i = j;
```

Second, wait until your ticket number is the minimum of all tickets in the room. There may be others with this minimum number, but in inspecting all the tickets in the room,

you found them! If you find a tie, see if your customer ID number is less than the ID numbers of those with whom you've tied, and only then enter the critical section and meet with the baker.

This is inefficient, because you might wait a bit too long while some other process picks a number after the number you picked, but for now, we'll accept this cost.

If you are not the person holding the smallest number, you start checking again. If you hold the smallest number, it is also possible that someone else holds the smallest number. Therefore, what you've got to do is agree with everyone else on how to break ties.

The solution shown above is simple. Instead of computing the value of the smallest number, compute the minimum process ID among the processes that hold the smallest value. In fact, we need not seek the minimum process ID, all we need to do is use any deterministic algorithm that all participants can agree on for breaking the tie. As long as all participants apply the same deterministic algorithms to the same information, they will arrive at the same conclusion.

To return its ticket, and exit the critical section, processes execute the following trivial bit of code:

N[i] := 0;

When you return your ticket, if any other processes are waiting, then on their next scan of the set of processes, one of them will find that it is holding the winning ticket.

Moving to a Distributed Context

In the context of distributed systems, Lamport's bakery algorithm has the useful property that process *i* only modifies its own *N[i]* and *C[i]*, while it must read the entries for all others. In effect, therefore, we can implement this in a context where each process has read-only access to the data of all other processes, and read-write access only to its own data.

A distributed implementation of this algorithm can be produced directly by storing *N[i]* and *C[i]* locally with process *i*, and using message passing when any process wants to examine the values of *N* and *C* for any process other than itself. In this case, each process must be prepared to act as a server for messages from the others requesting the values of its variables; we have the same options for implementing this service as we had for the token passing approach to mutual exclusion. The service could be offered by an agent process, by an interrupt service routine, or by periodic polling of the appropriate incoming message queues.

Note that we can easily make this into a fault tolerant model by using a fault-tolerant client-server protocol for the requests. If there is no reply to a request for the values of process i after some interval and a few retries, we can simply assume that process *i* has failed.

This demonstrates that fault tolerant mutual exclusion can be done without any central authority! This direct port of Lamport's bakery algorithm is not particularly efficient, though. Each process must read the variables of all other processes a minimum of 3 times−once to select a ticket number, once to see if anyone else is in the process of selecting a number, and once to see if it holds the minimum ticket.

For each process contending for entry to the critical section, there are about *6N* messages exchanged, which is clearly not very good. Much better algorithms have been devised, but even this algorithm can be improved by taking advantage of knowledge of the network structure. On an Ethernet or on a tree-structured network, a broadcast can be done in parallel, sending one message to *N* recipients in only a few time units. On a tree-structured network, the reply messages can be merged on the way to the root (the originator of the request) so that sorting and searching for the maximum N or the minimum nonzero N can be distributed efficiently.

2.8.4 Ricart and Agrawala's Mutual Exclusion Algorithm

Another alternative is for anyone wishing to enter a critical section to broadcast their request; as each process agrees that it is OK to enter the section, they reply to the broadcaster saying that it is OK to continue; the broadcaster only continues when all replies are in.

If a process is in a critical section when it receives a request for entry, it defers its reply until it has exited the critical section, and only then does it reply. If a process is not in the critical section, it replies immediately.

This sounds like a remarkably naive algorithm, but with point-to-point communications between N processes, it takes only 2(N−1) messages for a process to enter the critical section, N−1 messages to broadcast the request and N−1 replies.

There are some subtle issues that make the result far from naive. For example, what happens if two processes each ask at the same time? What should be done with requests received while a process is waiting to enter the critical section?

Ricart and Agrawala's mutual exclusion algorithm solves these problems. In this solution, each process has 3 significant states, and its behaviour in response to messages from others depends on its state:

Outside the critical section

> *The process replies immediately to every entry request.*

After requesting entry, awaiting permission to enter.

> *The process replies immediately to higher priority requests and defers all other replies until exit from the critical section.*

Inside critical section.

> *The process defers all replies until exit from the critical section.*

As with Lamport's bakery algorithm, this algorithm has no central authority. Nonetheless, the interactions between a process requesting entry to a critical section and each other process have a character similar to client-server interactions. That is, the interactions take the form of a request followed (possibly some time later) by a reply.

As such, this algorithm can be made fault tolerant by applying the same kinds of tricks as are applied in other client server applications. On receiving a request, a processor can be required to immediately send out either a reply or a negative acknowledgement. The latter says "I got your request and I can't reply yet!"

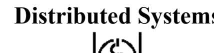
With such a requirement, the requesting process can wait for either a reply or a negative acknowledgement from every other process. If it gets neither, it can retry the request to that process. If it retries some limited number of times and still gets no answer, it can assume that the distant process has failed and give up on it.

If a process receives two consecutive requests from the same process because acknowledgements have been lost, it must resend the acknowledgement. If a process waits a long time and doesn't get an acknowledgement, it can send out a message saying "are you still there", to which the distant process would reply "I got your request but I can't reply yet". If it gets no reply, it can retry some number of times and then give up on the server as being gone.

If a process dies in its critical section, the above code solves the problem and lets one of the surviving processes in. If a process dies outside its critical section, this code also works.

Breaking Ties in Ricart and Agrawala's algorithm

There are many ways to break ties between processes that make simultaneous requests; all of these are based on including the priority of each requesting process in



the request message. It is worth noting that the same alternatives apply to Lamport's bakery algorithm!

A unique process ID can be used as the priority, as was done in Lamport's bakery algorithm. This is a static priority assignment and is almost always needed to break ties in any of the more complex cases. Typically, a process will append its statically assigned process ID to any more interesting information it uses for tiebreaking, thus guaranteeing that if two processes happen to generate the same interesting information, the tie will still be broken.

The number of times the process has previously entered the same critical section can be used; if processes that have entered the critical section more frequently are given lower priority, then the system will be fair, giving the highest priority to the least frequent user of the resource.

The time since last access to the critical section offers a similar opportunity to enforce fairness if the process that used the critical section least recently is given the highest priority.

If dynamic priority assignments are used, what matters is that the priority used on any entry to the critical section is frozen prior to broadcasting the request for entry, and that it remains the same until after the process is done with that round of mutual exclusion. It is also important that each process has a unique priority, but this can be assured by appending the process ID as the least significant bits of the dynamically chosen priority.

2.9 REMOTE PROCEDURE CALLS

Remote Procedure Call (RPC) is a powerful technique for constructing distributed, client-server based applications. It is based on extending the notion of conventional or local procedure calling, so that the called procedure need not exist in the same address space as the calling procedure. The two processes may be on the same system, or they may be on different systems with a network connecting them. By using RPC, programmers of distributed applications avoid the details of the interface with the network. The transport independence of RPC isolates the application from the physical and logical elements of the data communications mechanism and allows the application to use a variety of transports.

RPC is a client/server infrastructure that increases the interoperability, portability and flexibility of an application by allowing the application to be distributed over multiple heterogeneous platforms. It reduces the complexity of developing applications that span multiple operating systems and network protocols by insulating the application developer from the details of the various operating system and network interfaces‒function calls are the programmer's interface when using RPC. RPC makes the client/server model of computing more powerful and easier to program.

2.9.1 How RPC Works?

An RPC is analogous to a function call. Like a function call, when an RPC is made, the calling arguments are passed to the remote procedure and the caller waits for a response to be returned from the remote procedure. The *Figure 4* shows the flow of activity that takes place during an RPC call between two networked systems. The client makes a procedure call that sends a request to the server and waits. The thread is blocked from processing until either a reply is received, or it times out. When the request arrives, the server calls a dispatch routine that performs the requested service, and sends the reply to the client. After the RPC call is completed, the client program continues. RPC specifically supports network applications.

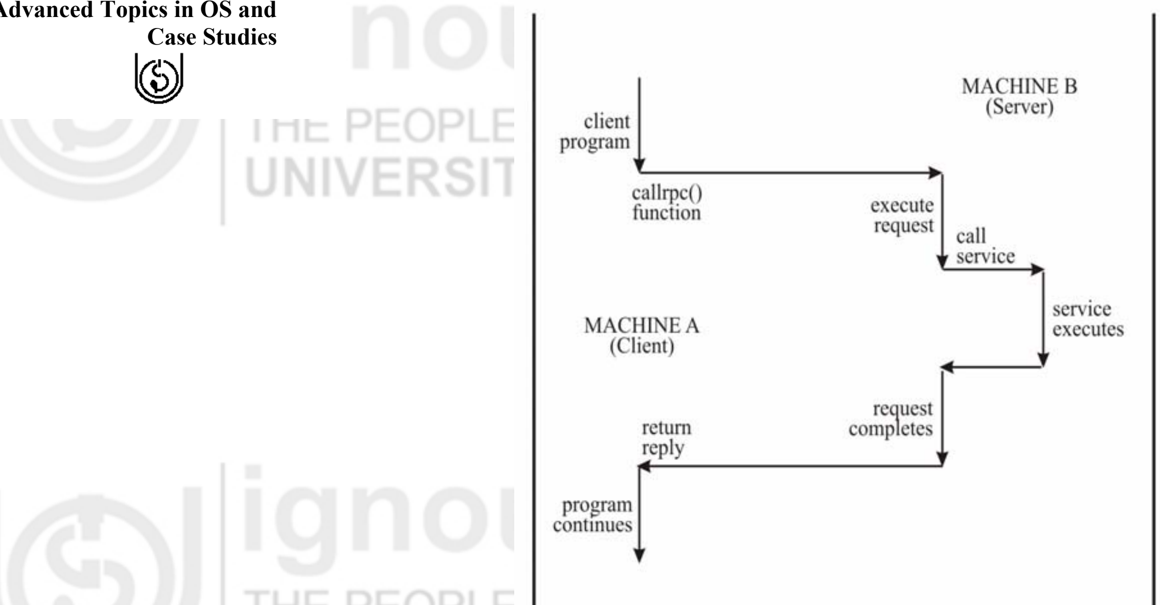


Figure 4: Flow of activity that Tables place during a RPC call between two networked system

2.9.2 Remote Procedure Calling Mechanism

A remote procedure is uniquely identified by the triple: (program number, version number, procedure number). The program number identifies a group of related remote procedures, each of which has a unique procedure number. A program may consist of one or more versions. Each version consists of a collection of procedures which are available to be called remotely. Version numbers enable multiple versions of a RPC protocol to be available simultaneously. Each version contains a number of procedures that can be called remotely. Each procedure has a procedure number.

2.9.3 Implementation of RPC

RPC is typically implemented in one of two ways:

1) Within a broader, more encompassing propriety product.

2) By a programmer using a proprietary tool to create client/server RPC stubs.

2.9.4 Considerations for Usage

RPC is appropriate for client/server applications in which the client can issue a request and wait for the server's response before continuing its own processing. Because most RPC implementations do not support peer-to-peer, or asynchronous, client/server interaction, RPC is not well-suited for applications involving distributed objects or object-oriented programming.

Asynchronous and synchronous mechanisms each have strengths and weaknesses that should be considered when designing any specific application. In contrast to asynchronous mechanisms employed by Message-Oriented Middleware, the use of a synchronous request-reply mechanism in RPC requires that the client and server are always available and functioning (i.e., the client or server is not blocked). In order to allow a client/server application to recover from a blocked condition, an implementation of a RPC is required to provide mechanisms such as error messages, request timers, retransmissions, or redirection to an alternate server. The complexity of the application using a RPC is dependent on the sophistication of the specific RPC implementation (i.e., the more sophisticated the recovery mechanisms supported by RPC, the less complex the application utilising the RPC is required to be). RPC's that implement asynchronous mechanisms are very few and are difficult (complex) to implement.

When utilising RPC over a distributed network, the performance (or load) of the network should be considered. One of the strengths of RPC is that the synchronous, blocking mechanism of RPC guards against overloading a network, unlike the asynchronous mechanism of Message Oriented Middleware (MOM). However, when recovery mechanisms, such as retransmissions, are employed by an RPC application, the resulting load on a network may increase, making the application inappropriate for a congested network. Also, because RPC uses static routing tables established at compile-time, the ability to perform load balancing across a network is difficult and should be considered when designing a RPC-based application.

Tools are available for a programmer to use in developing RPC applications over a wide variety of platforms, including Windows (3.1, NT, 95), Macintosh, 26 variants of UNIX, OS/2, NetWare, and VMS. RPC infrastructures are implemented within the Distributed Computing Environment (DCE), and within Open Network Computing (ONC), developed by Sunsoft, Inc.

2.9.5 Limitations

RPC implementations are nominally incompatible with other RPC implementations, although some are compatible. Using a single implementation of a RPC in a system will most likely result in a dependence on the RPC vendor for maintenance support and future enhancements. This could have a highly negative impact on a system's flexibility, maintainability, portability, and interoperability.

Because there is no single standard for implementing a RPC, different features may be offered by individual RPC implementations. Features that may affect the design and cost of a RPC-based application include the following:

- support of synchronous and/or asynchronous processing
- support of different networking protocols
- support for different file systems
- whether the RPC mechanism can be obtained individually, or only bundled with a server operating system.

Because of the complexity of the synchronous mechanism of RPC and the proprietary and unique nature of RPC implementations, training is essential even for the experienced programmer.

2.10 OTHER MIDDLEWARE TECHNOLOGIES

Other middleware technologies that allow the distribution of processing across multiple processors and platforms are:

- Object Request Broker(ORB)
- Distributed Computing Environment (DCE)
- Message-Oriented Middleware (MOM)
- COM/DCOM
- Transaction Processing Monitor Technology
- 3-Tier S/W Architecture.

☞ Check Your Progress 1

1) How is a distributed OS different from Network OS? Explain.

……………………………………………………………………………………

……………………………………………………………………………………

……………………………………………………………………………………

2) What is a Distributed File System?

…………………………………………………………………………………………

…………………………………………………………………………………………

…………………………………………………………………………………………

3) Define Load Balancing.

…………………………………………………………………………………………

…………………………………………………………………………………………

…………………………………………………………………………………………

4) What is the significance of Time Consistency?

…………………………………………………………………………………………

…………………………………………………………………………………………

…………………………………………………………………………………………

5) What is a Remote Procedure Call and mention its use.

…………………………………………………………………………………………

…………………………………………………………………………………………

2.11 SUMMARY

A distributed operating system takes the abstraction to a higher level, and allows hides from the application where things are. The application can use things on any of many computers just as if it were one big computer. A distributed operating system will also provide for some sort of security across these multiple computers, as well as control the network communication paths between them. A distributed operating system can be created by merging these functions into the traditional operating system, or as another abstraction layer on top of the traditional operating system and network operating system.

Any operating system, including distributed operating systems, provides a number of services. First, they control what application gets to use the CPU and handle switching control between multiple applications. They also manage use of RAM and disk storage. Controlling who has access to which resources of the computer (or computers) is another issue that the operating system handles. In the case of distributed systems, all of these items need to be coordinated for multiple machines. As systems grow larger handling them can be complicated by the fact that not one person controls all of the machines so the security policies on one machine may not be the same as on another.

Some problems can be broken down into very tiny pieces of work that can be done in parallel. Other problems are such that you need the results of step one to do step two and the results of step two to do step three and so on. These problems cannot be broken down into as small of work units. Those things that can be broken down into very small chunks of work are called fine-grained and those that require larger chunks are called coarse-grain. When distributing the work to be done on many CPUs there is a balancing act to be followed. You don't want the chunk of work to be done to be so small that it takes too long to send the work to another CPU because then it is quicker to just have a single CPU do the work, You also don't want the chunk of work to be done to be too big of a chunk because then you can't spread it out over enough machines to make the thing run quickly.

In this unit we have studied the features of the distributed operating system, architecture, algorithms relating to the distributed processing, shared memory concept and remote procedure calls.

2.12 SOLUTIONS / ANSWERS

Check Your Progress 1

1) A distributed operating system differs from a network of machines each supporting a network operating system in only one way: The machines supporting a distributed operating system are all running under a single operating system that spans the network. Thus, the print spooler might, at some instant, be running on one machine, while the file system is running on others, while other machines are running other parts of the system, and under some distributed operating systems, these parts may at times migrate from machine to machine.

   With network operating systems, each machine runs an entire operating system. In contrast, with distributed operating systems, the entire system is itself distributed across the network. As a result, distributed operating systems typically make little distinction between remote execution of a command and local execution of that same command. In theory, all commands may be executed anywhere; it is up to the system to execute commands where it is convenient.

2) Distributed File System (DFS) allows administrators to group shared folders located on different servers and present them to users as a virtual tree of folders known as a namespace. A namespace provides numerous benefits, including increased availability of data, load sharing, and simplified data migration.

3) Given a group of identical machines, it is wasteful to have some machines overloaded while others are almost idle. If each machine broadcasts its load average every few minutes, one can arrange for new processes to use whichever machine was least loaded at the time. However, one machine must not be given too many processes at once. If the system supports migration, and the load difference between two machines is great enough, a process should be migrated from one to the other.

4) Machines on a local area network have their own clocks. If these are not synchronized, strange things can happen: e.g., the modification date of a file can be in the future. All machines on a LAN should synchronize their clocks periodically, setting their own time to the "network time". However, adjusting time by sudden jumps also causes problems: it may lead to time going backward on some machines. A better way is to adjust the speed of the clock temporarily. This is done by protocols such as NTP.

5) Many distributed systems use Remote Procedure Calls (RPCs) as their main communication mechanism. It is a powerful technique for constructing distributed, client server based applications. It is based on extending the notion of conventional or local procedure calling, so that the called procedure need not exist in the same address space as the calling procedure. The two processes may be on the same system, or they may be on different systems with a network connecting them. By using RPC, programmers of distributed applications avoid the details of the interface with the network. The transport independence of RPC isolates the application from the physical and the logical elements of the data communications mechanism and allows the application to use a variety of transports.

2.13 FURTHER READINGS

1) Abraham Silberschatz, Peter Baer Galvin and Greg Gagne, *Applied Operating System Concepts*, 1/e, John Wiley & Sons, Inc, New Delhi.

2) Maarten van Steen and Henk Sips, *Computer and Network Organization: An Introduction*, Prentice Hall, New Delhi.

3) Andrew S. Tanenbaum and Albert S.Woodhull, *Operating Systems Design and Implementation*, 2/e, Prentice Hall, New Delhi.

4) Andrew S. Tanenbaum, *Modern Operating Systems*, 2/e, Prentice Hall, New Delhi.

5) Maarten van Steen and Andrew S. Tanenbaum, *Distributed Systems* 1/e, Prentice Hall, New Delhi.

6) Andrew S. Tanenbaum, *Distributed Operating Systems*, 1/e, Prentice Hall, 1995), New Delhi.

7) Jean Bacon, *Concurrent Systems: An Integrated Approach to Operating Systems, Database, and Distributed Systems*, 2/e, Addison Wesley, New Delhi.

8) Jean bacon and Tim Harris, *Operating Systems: Concurrent and distributed software design*, Addison Wesley, 2003, New Delhi.

9) Coulouris, Dolimore and Kindberg, *Distributed Systems: Concepts and Design* 3/e, Addison-Wesley, New Delhi.

10) D.M. Dhamdhere, *Operating Systems – A Concept Based Approach*, Second Edition, TMGH, 2006, New Delhi.