Distributed Operating Systems 1991
An Introduction plus

Case Study:

How can they improve the power, flexibility and performance of a score of workstations?

George Patapis
Computing Discipline
Macquarie University
N.S.W. 2109 Australia

1. Introduction

"A distributed operating system is one that looks to its users like an ordinary centralised operating system but runs on multiple, independent central processing units (CPUs). The key concept here is transparency", [Tanenbaum and Renesse1985].

The area of **Distributed Operating Systems** has moved steadily in the past decade from one of pure research, to one where a handful of such systems have been developed and implemented. **Transparency** has always been the key and goal of such an operating system. The user should be unaware of where his* files are located and where on the system his jobs are running. This paper is intended for those with a knowledge of centralised operating systems and computer networks, who wish to find out more about **DOSes** (Distributed Operating Systems). It is intended as both an introduction and as a literature review in the area. Also, a case study will be examined to determine what exactly can be accomplished today with a collection of SUN4 workstations and such an operating system. The aim of this paper is not to replicate the one by [Tanenbaum and Renesse1985] or the book by [Goscinski1991], rather, to provide for the reader a swift entrance with up-to-date information to this area of computing.

1.1 The Forerunners of DOSes

It is essential that we put the development and the plethora of operating systems into some sort of grouping and context. Initially, on the first computers there were no operating systems. Everything had to be done by the operator manually using panel switches, punched cards and the like. With subsequent technological developments operating systems made their appearance. An **operating system** can be defined as a program that acts as an intermediary between a user of a computer and the computer hardware. The purpose of an operating system is to provide an environment in which a user can execute programs in a convenient and efficient manner [Silberschatz et al.1991]. With this basic definition in mind, many such operating systems have appeared in the past few decades. They can be grouped with possible overlap, into multiuser, singleuser, network, and operating systems used in distributed systems.

Multiuser operating systems, are the operating systems running on a single computer with many users at once. Users are most likely to be connected to this central computer via terminals. Resources are allocated based upon some embedded system functions. Examples of such an operating system are UNIX and VMS. These are also commonly referred to as centralised operating systems. By resources we are referring to both the hardware and software resources. Hardware resources are storage devices, processors, memory etc., and software resources are shared programs, files and data.

---

* 'his' is used in the context of 'his/her'

A **Singleuser** operating system, is an operating system which runs on a single computer with one user. Obviously in this case all the resources are allocated to the one and only user. These appeared after computers became affordable enough for an individual to own one. MS-DOS and OS/2 are well known to belong in this category.

A **Network Operating System** is one which resides on a number of interconnected computers (mostly homogeneous), using some network topology. It allows the users to easily communicate with any of the other computers and users on the system. They can perform remote logins and transfer files and messages. The key factor here is that the user is generally aware of what machine he is working on. Resources such as plotters, laser printers and hard disks are shared in such a system. In the category of network operating systems, we could also include PC LANs (Local Area Networks). The operating system design here is a mixture of the multiuser system (we have the file server running an operating system i.e. Novell Netware), and the single user (we have individual computers running their own operating system like DOS or OS/2). An additional shell program would reside on each machine allowing the intercommunication to take place.

The term distributed system is often misconceived. The operating systems used in a **Distributed System** are by no means necessarily tied to DOSes. When we talk about distributed systems, we are talking at a higher level, most often at the user level rather than at the low level or process level. For example, banks have a whole collection of machines scattered across a wide area network running all sorts of programs and database applications. The operating systems in those machines are of less importance than the applications themselves. It is the nature of these banking applications which is distributed (many branches, headquarters, and ATMs all interacting), hence the name. A distributed system can be built upon a distributed operating system as we shall see but the two do not necessarily go hand in hand. In such systems, many of the features that we would presume to find at the operating system level, are found at the program level. For example concurrency control, consistency and fault tolerance can be embedded in the design and applications accessing a database rather than relying on the operating system to provide the necessary support for consistency and data integrity. In many references, the distinction between distributed systems and the rest is quite vague. This can be witnessed in [Coulouris and Dollimore1988], and others. Examples of distributed systems include Andrew [Morris et al.1986] and Athena [Rickard1989].

2. Distributed Operating Systems

2.1 Goals

A distributed operating system runs on a collection of computers that are interconnected via a high speed network. Topologies in such a system can include LANs and WANs. The critical research issues in DOSes as presented by [Tilborg1987 p.552] are: transparency of system resources and distribution, fault tolerance and robustness, life cycle enhancement, comprehensible model of interaction with users and standards. Ultimately we would like a distributed operating system to run and handle a collection of computers scattered across the office, building, country or even world and providing much greater benefits than any other operating system. The user would view this collection of computers as a large uniprocessor, or a large global computer whose full resources are available to him. Transparency was mentioned as a key attribute of DOSes. There are several forms of transparency as presented in [Goscinski1991]. We have access, location, name, control, data, execution and performance transparency. **Access transparency** refers to the process having the same type of access mechanism for resources whether they are local or remote. **Location transparency** means that the location of the resource is not visible as far as the access method is concerned. The naming of an object is independent of the node on the network on which the object was issued in **naming transparency**. In **control transparency**, all information describing a system should have identical appearance to a user or application. We have **data transparency** when the ability to access remote data in a transparent manner is supported. In **execution transparency**, the load of a system can be distributed, processes and data can be moved around the system, interprocess communication between remote processes is permitted. Finally in **performance transparency**, the user should witness little reduction in performance as a result of remote calls in the system. It should be noted that there are arguments for and against full transparency of the various types. Some of the transparencies may not even be completely feasible.

The development of DOSes are occurring as a consequence of user requirements and needs, and technological developments in the area of computing. These technological developments include the advancements and cost reductions in personal computers and workstations with VLSI, plus the developments in high speed networks. Traditional methods for operating system improvement and research were summed up in [Fleisch1983] and included making things faster, more efficient and more secure. The author also mentioned trying to incorporate some sort of **"intelligence"** into an operating system may be a future goal. Connecting multiple workstations together instead of purchasing a single mainframe and achieving the same performance is very attractive. A DOS should increase reliability and fault tolerance. This is accomplished by replicating data on the many hard disks, therefore if data is inaccessible from one source, it can be accessed from another. When one of the computers attached to the system goes down, this will not bring the whole system down. In an ideal situation, if ten per cent of the system is down, performance should only decrease by that same amount. If we require an increase in performance we should be able for example to just add another ten workstations onto our existing system of a hundred and therefore increase the power by ten per cent. This aspect is called **incremental growth**. An important goal is to be able to disperse the workload across the system instead of having all the work on one machine. The sharing of files should result as an outcome

of a good DOS design. We would also like the system to handle most of the administration tasks automatically thus making it less expensive to run and maintain (this was the basic design feature of the Athena Model of Computing at M.I.T.).

2.2 Potential Problems

Interprocess communication in a DOS environment takes place over the network. Network protocols and transmission time can become a bottleneck. There is a lack of global state information. Not knowing all the details of the system makes it hard for process schedulers to operate. In a distributed system, there is often no global time, each computer has its own concept of time. Security is a major issue because in most cases the network cable is prone to tampering.

2.3 System Configurations

There have been three configurations proposed for a DOS. The first is the **workstation model**. Here we have a collection of computer workstations each with a high resolution bit-mapped display, plenty of memory, a powerful central processor and in many cases, a local hard disk. This model seems to be gaining popularity because of the increasing availability of such workstations and the need to increase their utilisation. The second model is the **processor pool model**. Here we have added processors available in the system to be distributed amongst jobs as required. If an image analysis program requires the use of ten processors it can do so and later return them to the pool. The idea here is to increase the CPU power per user to more than one which is what you would have on a standalone workstation. Amoeba is a DOS which incorporates this feature. There has been at least one argument against adopting such a design approach. In [Cheriton1988], the author gives his opinion on such an implementation which came as an outcome of his experience designing the V Distributed System. He states that his group originally considered having a pool of dedicated server machines to support remote and distributed parallel computation. However, they found that the utilisation of the workstation resources with a workstation per person to be sufficiently low that additional processors were not required. Moreover, the observation made was that each generation of workstations is so much faster than the previous generation. So, a dedicated pool of previous generation processors would likely get less use than a few idle workstations of the current generation. There is an argument here as to what would happen in an environment where massive computation is required for example when running graphics applications or neural network simulations. A solution might be to stay with the workstation pool model but also add to your system a few "number crunching" machines if there is a need for them. The final model in this category, is the **minicomputer model**. Here, a collection of minicomputers are connected. Such a system is DUNIX [Litman1988]. Another method for categorising DOSes is into **homogeneous** and **heterogeneous models**. The first requires homogeneous architecture amongst the computers used whereas the second does not. Obviously the heterogeneous model would be the most attractive from a first perspective, allowing us to incorporate into our system all sorts of machines from PC's and workstations to minicomputers and mainframes. There are arguments though which state that a homogeneous model provides better performance which is enough to substitute

for the advantages of the heterogeneous one. Such was the intention of the DUNIX DOS creators who preferred perfect integration of homogeneous computers over poor integration of heterogeneous computers [Litman1988].

Object oriented design in a DOS's architecture is another distinguishable feature. An object can be any resource, such as files, disks, disk blocks, processors, or processes. Each object is managed by a server which defines a set of operations to manipulate the object. Such operations might be Create, Delete, Read or Run. Clients can access an object only through its operations [Renesse et al.1988 p.822].

Some early systems were built on top of the Unix operating system (in most cases BSD4.3) to provide (as close as possible) binary compatibility with existing software. This has mostly a negative effect on performance in contrast to a DOS where the operating system is written from scratch. MACH has achieved binary compatibility with BSD4.3 Unix without using any code from BSD as we shall see later in the paper.

3. Interprocess Communication

Just as in a centralised system, processes need to communicate with each other in a DOS. Only in the DOS case shared memory is not available in the same form as it is in a centralised system. Processes cannot exchange information using shared memory, which were protected using monitors, semaphores and the likes in an easy manner, especially when they are executing on separate machines. Therefore other methods allowing processes to identify other processes, communicate and solve synchronisation problems, have been introduced. The three most popular are **message passing**, **remote procedure calls (RPCs)** and **transactions**. With message passing, the operating system basically offers two primitives, send(message) and receive(message). In the RPC model, communication between two remote processes resembles that of calls and interaction between conventional procedures. Finally, the transaction method is based on the idea that a single communication is not enough between interacting processes. Rather we have a sequence of communications and computations which make up what we call a transaction.

3.1 Message Passing

A **message** is a typed collection of data objects consisting of a fixed size header and a variable (sometimes constant) length body, which can be managed by a process, and delivered to its destination [Goscinski1991]. The exact structure of these messages depends on the way we are going to use them to communicate. In a heterogeneous network, structured messages are much easier to handle. Unstructured messages would require considerable processing overheads.

Messages can be split into data messages for transferring data and control messages which contain information for an action to be taken. It is convenient to think of this form of communication like that between a client process and a server process. The client sends a message to the server process requesting some service. A communication link is implemented between the processes that want to communicate. The first issue tackled here is how does the client identify the server? How does the server identify a client requesting a service? The two broad categories here are **direct** and **indirect communication**. In direct (or otherwise called **naming**) communication, the process ID's or name are used in communication between client and server. With direct, symmetric addressing, both the sender and receiver are explicitly named in the messages. In the other form of direct addressing called asymmetric addressing, only the client identifies the server by name in its message and the server uses the identification of the process that has called it for further communication.

The communication link created in direct symmetric addressing, has the following properties:

- Between every pair of processes that wish to communicate, a link is automatically established.
- A link is associated with exactly two processes and there can be only a single link for these two particular processes.

- Finally the link is bidirectional, that is information can flow in both directions.

The advantages of using naming schemes like this are that they are simple to implement and their use is straight forward. The drawback is that modularity is sacrificed. If a process's name is changed then we have to change all references to reflect the change. The major disadvantage is that only one client can communicate with a server at a time. That is because, at the very least, a receive would be required for each requester. The server process cannot reasonably anticipate the names of all potential clients. Similarly, direct naming does not make it possible to send one request to more than one identical server [Goscinski1991, p.152].

In indirect addressing we use ports (sometimes called **mailboxes**) for communication. **Ports** can be viewed as an abstraction where messages can be placed and retrieved. Each port has a unique identifier. Primitives used to operate on ports are send (P, message) and receive (P, message) where P is the port identifier. A port is like a repository where one process can leave a message and the other can pick it up. The communication link formed in this case has the following properties:

- The link is established between two processes if they share a port.
- The link can be associated with more than two processes.
- More than one link can be associated with a pair of processes with each link corresponding to a different port.
- The link can be either bidirectional or unidirectional.

A port can belong to the operating system or can be declared and belong to a process. In this case communication with the process is via this port and when the process terminates, the port must cease to exist. It is possible for ownership of the port to be given to some other process. There is also some exception and error handling that the system has to deal with. If a process owner of a port "dies", subsequent processes sending messages to this port must be notified.

If two processes are sharing a port and a message arrives the system must determine who receives the message. In many cases message queues are provided to buffer messages arriving at ports which have yet be to be received. We can have bounded or unbounded queues for this purpose. A process can send the message to the port to be queued and can return control to the user program immediately. We call this feature **non-blocking** message passing. It should be evident that if no queue is used, the two processes must be synchronised so when one forwards the message to a port, it must wait (sender is blocked) until the other process takes it. We call this synchronisation a **rendezvous**.

The actual data in a message, can be passed by reference or by value. By value is the easiest method. Here we put all the data into the message and send it across. This method is expensive though. With pass by reference, we only pass a pointer to the address of the start of the message. This means that some form of global shared memory is required. Also, in a heterogeneous system, shared memory techniques like the above may be difficult to implement. A balance between the two methods seems to be the best alternative.

Messages can be lost or scrambled in a distributed operating system because of network problems. Here the alternatives we have are: either that the operating system detects this and tries to resend that one message or, that the sending process itself is responsible totally for the monitoring and accurate delivery of the message. An in between solution would be for the operating system to detect whether a message has been lost and then to notify the sending process which tackles the problem from there onwards. To detect the loss or damage of a message we have what is called **reliable** send and receive primitives. The only difference using reliable primitives is that after a send has been executed, an acknowledgment must be received from the other side stating whether or not the message arrived in the correct format. The sender would continue work if the transmission was fine otherwise it will probably try to retransmit the message. Using unreliable primitives means that there will be no acknowledgment of the message. This method is simpler to implement and is used if failure detection is low on the priority list.

3.2 Remote Procedure Calls

The idea of using remote procedure calls for interprocess communication has been gaining acceptance because the concept seems simple enough for the average programmer to understand and use. The use of procedures in high level languages for programs running on single machines is well understood. This mechanism is extended to a distributed operating system environment with the difference being that the procedure call will activate and execute possibly on another machine on the system. This transition though is totally transparent to the user as the operating system handles the passing of data and the transmission of the message across the network to the required destination. There the message is decoded, the data passed to the required procedure which returns the result through the same path back to the original caller on the machine from which the message originated. The packing and unpacking of data in the call is often called **marshalling**. Execution then resumes as if we had just called a local procedure.

Major issues facing the design of a RPC facility include: the precise semantics of a call in the presence of machine and communications failures; the semantics of address-containing arguments in the (possible) absence of a shared address space; integration of remote calls into existing (or future) programming systems; binding (how a caller determines the location and identity of the callee); suitable protocols for transfer of data and control between caller and callee; and how to provide data integrity and security (if desired) in an open communication network [Birrell and Nelson1984]. The properties that RPCs should exhibit are summed up in [Goscinski1991 p.166]. He states: **uniform call semantics**: the semantics used for a local procedure call should be maintained in the case of remote procedures; **type checking**: the same level of static type checking is applied to remote procedures as it is for local ones, therefore the compiler makes no distinction; **full parameter functionality**: all data types including user defined, primitive and structured data types should be allowed to be passed with an RPC call; **concurrency control** and **exception handling**. For good RPC design, the programming language is required to support the following features: **distributed binding**: there must be support for compiling, binding and loading distributed programs onto the network by the programming language that uses RPCs and **orphan computations**: provide facilities for recovery from a failed RPC. Timeouts could be used to determine failed calls (i.e. no response within a given period of time).

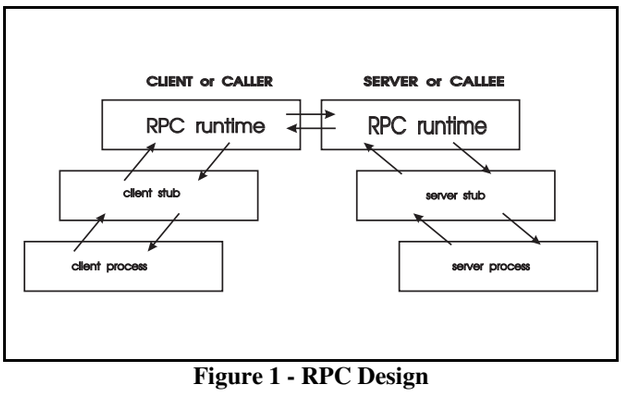

Figure 1 - RPC Design

There are mainly five pieces of program involved when making a remote call: **the user, the user-stub, the RPC communications package, the server-stub and the server** (see Figure 1). Remote Procedure Call is one of the most important aids for writing distributed applications today. The stub compilers take care of the details of transmitting the data structures, dealing with different byte ordering and creating threads of control in which the remote procedures execute [Mullender1988 pp.60]. The system and language used to implement the RPC mechanism can have a significant effect on the outcome. In [Almes1986], Modula-2 and the V distributed Operating System [Cheriton1988] where used to implement a RPC mechanism. V already uses a message-based interprocess communication mechanism. The goal of the work though was to provide a higher-level interface to V's interprocess communication mechanisms, while retaining the high performance of systems written to use the mechanism directly [Almes1986 pp.415]. The author stated that the specific concept of a module in the Modula-2 language was key to the success of the implementation. In [Souza and Miller1986], the authors tried to build a RPC system on top of the Berkeley BSD 4.3 Unix system. The results where less gratifying and the main cause seemed to be the UNIX kernel on which they built their system.

3.3 Transactions

The concept of a transaction has its roots in database theory. A transaction is started, a series of operations take place, if they are correct they are recorded and the transaction terminates. It is this series of operations together with the basic concepts of commitment, concurrency and recovery that define a transaction. **Commitment** refers to all the operations having successfully been executed and it is then that the transaction is accepted and identified as complete. **Concurrency** is associated with simultaneous access of data, or a service provided by a server by more than one client at a time. There must be some sort of concurrency control in situations like these to protect against deadlocks and inconsistencies. Finally, **recovery** means that we must be able to reverse the effects of a transaction which has not been committed due to some error, therefore maintaining the integrity of the system. The operations performed are primarily based on the message passing mechanism.

3.4 Performance

It is interesting to examine the performance of systems based on IPC using the methods described above. Actual results and test cases were presented by various authors on systems using the

RPC method. In the DUNIX system [Litman1988 pp.48], the authors conducted a test, compiling the Bourne shell which consisted of roughly three and a half thousand lines of C code and composed of twenty source files. They ran the job on three systems using DEC VAX 750s and an Ethernet connection. The three configurations were: a standalone computer running 4.2 BSD Unix, a standalone computer running DUNIX and finally a DUNIX system with all the relevant processes on one computer and the relevant files (which include /bin and /tmp) on another. The results were:

4.2 BSD Unix: 334 seconds

DUNIX-single machine: 261 seconds

DUNIX-remote files: 306 seconds

The results were achieved after performing a number of test runs and averaging out the results. From the results above, the authors claim that crossing computer boundaries using DUNIX does not penalise the user. Both DUNIX configurations were faster than 4.2 BSD running on a single machine. The main reason for this is the simplicity of the DUNIX kernel which is only 17K of source lines (compared to 45K for 4.2 BSD). In fact, simplicity of kernel design is a key issue in most DOS implementations. There is a tendency to move as much out of the kernel as possible and the results seem to be rewarding if we look at DUNIX's performance. In [Almes1986 p.417] where the V system and Modula2 were used, the author states that a null call results in roughly 10% overhead with this overhead becomes negligible when the computation within the remote procedure becomes non-trivial. The author believes that the performance is very good and the design provides a high degree of useability.

4. Services

4.1 Distributed File Service

One of the most useful parts of an operating system is the file handling and file system. It is through the use of the file system that a user mostly comes in contact with the operating system. In a distributed operating system environment, dedicated file servers offer their services to users. This can be contrasted with a centralised operating system where the file service can be integrated with the operating system. In the case of the distributed file system, the users must still be provided with the file services normally found in centralised systems in addition to file sharing, enhanced reliability and other features.

4.1.1 Goals

A distributed file system is designed to accomplish the following goals summed up by [Goscinski1991 p.712]:

- to be able to handle diverse applications;
- to provide easy to understand semantics;
- to provide and meet each users separate requirements;
- to use a consistent naming scheme for objects, independent of location;
- to provide transparent access mechanisms for processes accessing files throughout the system;
- file data integrity should be supported with the provision of concurrency control;
- the authentication of users should be consistent and irrelevant of physical location;
- to protect file information from unauthorised access;
- users should be allowed to access their files from anywhere in the system without performance degradation;
- hardware problems at one point in the system should not prevent file services to be provided to the majority of other users;
- the system should be scalable, allowing for growth without reduction in performance;
- several storage media should be supported;
- recovery mechanisms should be provided to enable servers to handle crashes in parts of the system, enhancing reliability.

Of all the design factors, this is the most important: to provide recoverability after a file server crash or a client crash. In [Coulouris and Dollimore1988 p.149], the concept of careful block transfers and checksums together with algorithms are described to implement recoverability. In [Tanenbaum and Renesse1985], the idea of **mirrored disks** is presented. Basically, a mirrored disk is an absolute copy of an original disk which will be used in the case the original crashes. A well known file replication technique employed by SUN which requires one central master disk and several secondary slave servers, is **The Sun YP**. This is called a master slave strategy because the master server holds copies of all files and services update requests. An appropriate slave holds a copy of a file and is sent changes to the file either from the master or else the slave

takes a copy of the master copy. In large distributed file systems we would probably have more than one disk and file server. Usually file servers in this case will handle a portion of all the files in the system and other file replication techniques are used to allow for recovery and reliability. Performance in a distributed file system is affected by the fact that there are overheads of transmission over the network and protocol handling.

4.1.2 Architecture

For a distributed file system to provide all the services mentioned the implementation can be easier handled if the system is split up into distinct modules as required. A set of modules is presented by [Goscinski1991 pp.715]: for the disk I/O and buffering we have a device module; the allocation and access of disk blocks is handled by the block module; for caching disk operations we have the caching module; file identifiers are associated to files by the file module; the file data is read and written by the file access module; file attributes are manipulated by the attribute access module; a transaction is handled by the transactions operations module; failure recovery mechanisms are provided by the recovery module; concurrency control is provided by a concurrency module; a data recovery and multiple copy update module is used for shared files; the mapping of logical file names to system names and file IDs is done by the directory module; access rights for operations are handled by an access control module; the physical location and placement of newly created files is handled by a placement module and these files are then located by a location module; a relocation module can be used finally to relocate files for improving performance in certain cases (see Figure 2). Obviously each DOS implementation may use its own terms to describe the modules performing a certain activity but basically all the above modules in one form or another should be deemed necessary for any file system identified as distributed.

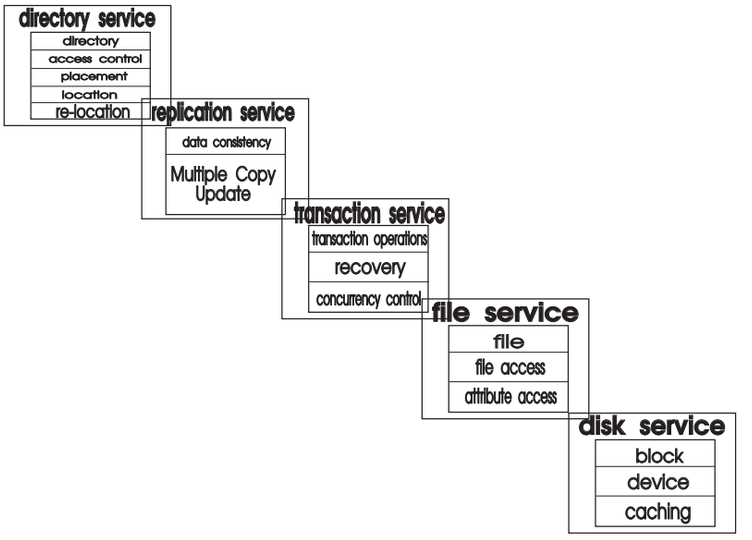

Figure 2-The modules (adapted from [Goscinski1991, p.71])

4.2 Process Management

The term **process** can be thought of as an abstraction of a program in execution. It consists of the executable program, the program's data and stack, its program counter, stack pointer, and other registers, and all the other information needed to run the program [Tanenbaum 1987 p.15]. In a DOS, the aim is to distribute the processing resources amongst all processes and to provide

mechanisms for handling these processes. Issues that need to be resolved are:

- how do we handle local and remote execution of processes?
- how do we tackle concurrency amongst processes competing for the same resource?, and
- how do we manage their states?

All the issues concerning processes in a centralised system (i.e. how to create, terminate, etc.) are applicable here in addition to many others.

4.2.1 Process Classification

An attempt to classify processes in DOSes into **lightweight**, **mediumweight** and **heavyweight processes** can be found in [Goscinski1991 p.381]. The division is based upon the degree of shared memory that a process uses. A lightweight process consists of one or more threads (in the case of a single thread we merely refer to the lightweight process as a thread). A **thread** can be described as a piece of executable code which has access to shared data and a shared stack. It dose not necessarily have an entry in a process control block (a PCB is known from centralised operating systems to keep information about a process). For **lightweight** processes to function, they must execute in the same memory partition and therefore require hardware support to be implemented. Effective communication can take place between threads because of this sharing, but this also leads to degradation of inter-thread security. The use of such threads allows the implementation of concurrent programming and parallel process creation methods. We have less overhead in manipulating processes,- shared memory wherever possible, is by far the fastest interprocess communication technique thus increasing the utilisation of our processors. A **mediumweight** process has several threads which do not share a stack and are considered as a separate process. This enhances protection amongst threads. **Heavyweight** processes have a single thread and their own entry in a process control block. Heavyweight processes will run in separate address spaces. Various DOSes use their own naming scheme for the processes described above and not all of them are supported in the different systems.

4.2.2 Scheduling, Execution, Migration

In distributed scheduling we deal both with how a local processor handles its awaiting processes and at a global level, how we will distribute the workload amongst the various processors using remote execution and process migration techniques. Local scheduling algorithms (i.e. round robin, priority scheduling) are covered in most operating system textbooks. In a DOS we have the option of scheduling a process anywhere in the system. **Load balancing** and **load sharing algorithms** are used for this purpose. We want to avoid having processes queued up waiting for one processor while another is idle. For this to be accomplished, the load sharing and balancing algorithms have to decide which process should be transferred and on which computer a process should run. The problem becomes more complex if we are dealing with a heterogeneous DOS. A process created for one architecture can of course only function when moved to an identical platform. In the V Distributed System a process can be executed on a randomly chosen computer by using a command of the form:

```
<the_command> <the_arguments>@any
```

Process **migration** is the transfer of a sufficient amount of a process's state from one machine to another for the process to execute on the target machine [Smith1988]. It is attractive in distributed systems because by allowing this flexibility, we can move processes that are more suited to a particular processor, to that processor. We can duplicate processes and move them to another processor to enhance system reliability. If the data that a process is performing operations upon is large (e.g. a database file), we can move a process to where the data is instead of moving the data to where a process is. Process migration requires at the least the transfer of the process's state information across the system. Depending on what sort of migration mechanism is chosen, a way to handle the messages still destined for the process at its previous node, is required. Basically, a process migration mechanism would involve collecting the various required statistics from the system and the handling of the process migration itself. The process migration mechanism is chosen so the interference with the kernel and other running processes is minimal or non-existent. Processes that may migrate are designed in such a way that they can execute transparently on any node. The migration facility also provides for the owner of a workstation to have priority over his machine. Therefore any foreign processes that have migrated to this machine should give way to local processes. Efficiency and reliability mechanisms are also implemented into process migration facilities. Programs themselves can be written in a special manner to support migration and load balancing. A **WORM** program is one which can take maximum advantage of the population of free processors by expanding its operation to encompass them as they become available, and according to its needs. Worms operate at a number of different machines and respond dynamically to changes in processor availability at run time [Kindberg et al.1987 p.355]. Writing programs that will support migration in the system is a new and exciting area of research.

4.3 Other Services

We would expect a DOS to also provide a printing service to handle the printing requirements in such a way as to utilise as efficiently as possible the available printers. A mail service is also important since e-mail (electronic mail) has become an important part of any distributed system. Also useful is a boot service which can be used to bring up workstations when they go down. Other services can be considered depending on the user requirements and system configuration.

CASE STUDY

1. Setup

The Macquarie University Computing Discipline decided a year ago to decommission its main central computer, a GOULD 6050 running a variant of BSD 4.3 Unix, in favour of a collection of SUN4 workstations. In addition to the Computing Laboratory which houses most of these SUNs for use by students, staff have also acquired such workstations to use on their desktop. There is a high speed ethernet connection between the computers. The SUN4 workstations use RISC technology, each have roughly a 200Mb local disk, high resolution colour screen display and come with a version of SunOS (a variant of Unix). The move from a centralised system to a distributed one has given rise to many ideas on how the services provided by the old centralised model can be preserved and how better utilisation of the new workstations can be achieved using DOS concepts.

2. File System

The handling of the file system was already in place from the moment the computers were purchased. SUN Microsystems supplies the Sun Network File System (NFS) integrated with the operating system. That is, the kernel on each computer has modifications of the basic Unix kernel to support NFS features. The system is also provided with the necessary library procedures for programs to make use of the facility. NFS has some of the properties of a distributed file system and lacks others. In such a system there are clients and servers. Servers are distinguished by the attribute of having entries in the file /etc/exports. This determines the directories that the other machines (clients) can mount i.e. use as if the directories were local to their own machine. Some of these mounts are automatically performed when a Unix system is initialised by executing the commands in the /etc/rc file. Manual user mounts can be performed if the files and directories already mounted are inadequate. The file servers in NFS are stateless, they do not keep any state information on behalf of clients. When the appropriate directories are mounted (we can even have the whole root directory mounted), the users and the client programs do not need to be aware that files they are using are distributed. This allows programs to access local files and remote files transparently. The department uses two machines to act as file servers, one keeping files for student use and the other for staff use. Because of the distribution of files, NFS uses a v-node number to identify files incorporating the normal i-nodes used in conventional Unix systems. The v-node amongst other information, contains an identifier for the computer on which the file is present. The server kernels perform requested operations by clients using the v-node.

The stateless character of the server has its positive and negative results. It is indeed essential for the server to keep operating when a client fails but when a server fails, the clients are left orphans. Therefore we are left with exactly the same outcome as if we were using a centralised system; when the server goes down, the system is practically useless. Caching is used on clients

and servers to enhance performance. There is some limited protection against inconsistent files due to concurrent access by other users. The time that a file has been accessed and modified is monitored for this purpose. A slight negative effect is that this goes on for all files regardless of whether or not they are being used concurrently. For this reason a write through facility is provided to enable modified files to be immediately written back to disk. In any case, the user is unaware of these functions that are happening in the background. Sun's implementation of NFS is satisfactory but there is substantial room for improvement. The staff would like to be able to continue using the system even if the main file server is down. This requires adopting a technique presented in distributed file systems either using mirrored disks or more than one file server.

3. Process Management

Research work done within the department often requires significant computing resources. With the old central computing model, a process would be left to run in the background, given some priority and hopefully would return with a useful result after hours and hours of heavy processing. With the new structure, all a member of staff has to work with is a single workstation to run the process on. This same workstation though could be occupied with other work that the owner has. Computing tasks such as image processing and neural network simulations require too much computing power for a single workstation to handle and sometimes a dozen other workstations might be sitting idle. Initially staff would look around the system for an idle workstation and then transfer the work there and execute it. This is when the need for a process migrating mechanism was detected and **Condor** was put to good use. Condor was developed at the University of Wisconsin-Madison. Its purpose is to hunt for idle workstations just as the users were doing.

Condor version 4.0.0 can be used on the following machines: VAX, i386, SPARC, MC68020 and MIPS. It supports the following operating systems: BSD4.3 , DUNIX, SunOs and Ultrix. Basic Condor design issues are that the placement and execution of background jobs is done in a transparent-to-the-user manner. If a remote computer fails, the job is placed on another idle workstation and work resumes. Remote workstations should be accessed on a fair basis. A user with a heavy computing requirement should not be permitted to seize all the resources and a user with minimal requirements should be given access to resources as soon as possible. The mechanism used to implement condor must use minimal computing power and resources. Long jobs should be handled and finally there should be no interference with users using a workstation locally [Goscinski1991 p.476].

On each workstation, a scheduler of background processes is running managing a queue of pending jobs. It also keeps checking to see if the workstation is idle. On one particular machine, a central coordinator executes. Its task is to monitor and gather information about idle workstations on which local schedulers are running. When a local scheduler detects that a local user has accessed the workstation, any background job is suspended. So as not to lose any information of the background process that has been suspended, Condor uses **checkpointing** which saves the state of the suspended process in order for execution to continue at a later stage or on a new machine. Condor uses the local disks for storing checkpoint files. The local disks

on the workstations in the student computing laboratory are not used by students therefore there is always space available on them. To allow a program to use the Condor facility, a user would have to link it with some special Condor libraries. Condor does not allow you to use shared libraries. The flexibility of Condor is that it allows you to specify the hardware and software requirements of your program and it will then proceed to find a workstation which fulfils those requirements. It also allows you to set priorities on those jobs and it handles notifying the user of errors and upon job completion.

The Condor system is a first introduction for us to process migration. What has been learnt is that process migration is not necessarily associated with faster program completion. For jobs that would run overnight, Condor handles the task of finding a workstation, running the job and returning the results. For very long jobs, the users may be disappointed. The reason is that if a job requires a few days to complete, it will take more now with Condor because a job will not be put onto a workstation that is not idle. This means that jobs are normally restarted towards the end of the day when the workstations are free. Users who were used to scheduling jobs on workstations either idle or not are hesitant to use the facility. To allow those users to regain the same level of performance whilst using Condor might mean that more workstations or workstations with more memory will be required. Overall, Condor is a step in the right direction as far as process migration and load balancing is concerned.

4. Future Options

The first step towards a DOS have been taken. The next move might be to switch to one of the DOSes that will be available for the SUN4s. The almost dominant criterion for such a switch in this case, would be binary compatibility with programs used on the SunOs platform with that of a new DOS. It seems that no matter how much performance will increase with a true DOS, it will come second to being able to use existing applications. The switch from the central Unix system to the SUN NFS was well accepted because despite the loss of a few applications, new ones were immediately available (provided by SunOs) or ported easily. The X-Windows environment added also a new category of possible applications. Following is a description of some likely DOS candidates.

4.1 Amoeba

Amoeba is a DOS developed at Vrije Universiteit for which a considerable amount of literature already exists. It is based on the processor pool model. Every machine in Amoeba runs a minimal kernel. Basic process, communication and object primitives, memory management and raw I/O, are supported by the kernel. Everything else is built on top of this in the form of server processes running in user space. The whole system is structured as a collection of independent processes. Clients are user processes running some application. The Bullet file server is a server process. The unique feature of the Bullet file server is that it caches a whole file into memory. Basically the microkernel provides the environment where servers and clients can run and communicate. This is a modular design enabling modification to be done by users thus tailor making the system to their requirements. Amoeba uses the RPC form of interprocess communication. The Amoeba Interface Language (AIL) handles the generation of stub routines and

hides the details from the user. Other key concepts are objects and capabilities. An object in Amoeba is a data structure on which a certain number of operations are defined. For example a directory is an object with the operations "enter file" and "look up name". Capabilities are associated with an object and are used for identifying the owner and permissions. Amoeba comes with a C-compiler, provides a Unix emulation library called Ajax, TCP/IP, the X-Windows user interface and has a Unix kernel connection utility. In Europe, several Amoeba systems connected via a high level gateway and spread across three countries have been in operation for some time [Renesse1988]. Applications developed for this system were a tele-conferencing service, a calendar service, and others. A version of Amoeba for SUN4 stations should be appearing by the end of 1991.

4.2 MACH

MACH was developed at Carnegie-Mellon University and followed earlier work on the Accent system. It runs on hundreds of heterogeneous computers at CMU, is the main operating system of the NEXT computer and has been adopted by the OSF (Open Software Foundation) as its main operating system. It features binary compatibility with BSD 4.3 Unix. MACH Release 3 is due and the size of its kernel should be a tenth of that of BSD. The intention is to make a small kernel with low level functions for supporting BSD and other operating systems. That is, it should be possible to have other operating system environments running on top of MACH. Process management and the file system operate outside of the kernel. MACH is at least as efficient as other major versions of Unix when performing similar tasks and it frequently is faster [Silberschatz et al.1990 pp.600]. Basic abstractions used in MACH are tasks, threads, ports, messages and memory objects. The conventional term process used in operating systems, is replaced by that of a task having a single thread of control. An important feature of MACH is that it allows shared memory even amongst tasks on separate computers by using external memory managers and memory objects (see Figure 3). Developers are provided with the option to use messages or shared memory for communication between threads and tasks. Communication takes place through ports. Threads can execute in parallel on multiprocessor machines. In fact MACH provides extended support for multiprocessors. Task migration is also supported. MACH will be available for the SUN4 architecture sometime in 1991.

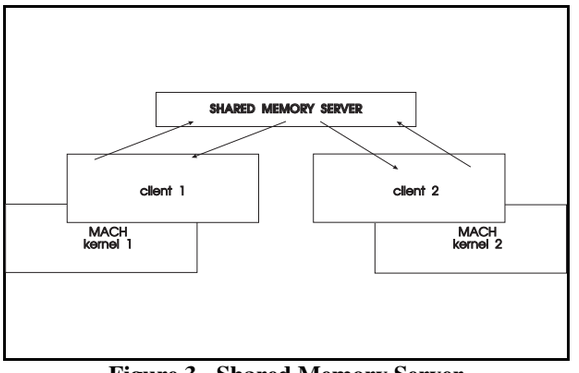

Figure 3 - Shared Memory Server

5. Conclusion

Not all aspects of Distributed Operating Systems were covered in this paper. Apart from the two systems examined, there are many others already developed or currently under development. It is hoped though that the information provided was enough to encourage thoughts about

utilising to a greater extent a system such as the one we have, the collection of SUN4s, by using a distributed operating system.

Acknowledgments

I thank Stephen Russel from The University of New South Wales for providing me with a wealth of references and information to get started. Dr. Andrezj Goscinski for sending me previews of sections from his book while it was being printed. Mike Johnson for his supervision. Len Hamey for always providing a helping hand. My friends, John Tassopoulos and Sandy Draganis, for assisting in the final stages of this paper.

References:

[Almes1986]: Guy T. Almes, The Impact of Language and System on Remote Procedure Call Design, 6th ICDCS IEEE 1986 pp.414-421.

[Armand et al.1989]: Francois Armand, Michel Gien, Frederic Herrman, Marc Rozier, CS/TR-89-36.1 Revolution 89 or "Distributing UNIX Brings it Back to its Original Virtues", Chorus systemes August 1989.

[Birrel1985]: Andrew D. Birrell, Secure Communications Using Remote Procedure Calls, ACM Transactions on Computer Systems, Vol.3, No. 1, February 1985, pp.1-14.

[Birrell and Nelson1984]: Andrew D. Birrell and Bruce Jay Nelson, Implementing Remote Procedure Calls, ACM Transactions on Computer Systems, Vol. 2, No. 1, February 1984, pp. 39-59.

[Cheriton1986]: David R. Cheriton, Problem-oriented Shared Memory: A Decentralised Approach to Distributed System Design, International Conference on Distributed Computing Systems, IEEE, 1986 pp.190-197.

[Cheriton1988]: David R. Cheriton, The V Distributed System, Communications of the ACM, March 1988, Vol.31, No.3, pp314-333.

[Coulouris and Dollimore1988]: G.F. Coulouris, S. Dollimore, Distributed Systems: Concepts and Design, Addison-Wesley 1988.

[Elmasri and Navathe1989]: Ramez Elmasri and Shamkant B. Navathe, Fundamentals of Database Systems, 1989, Benjamin/Cummings.

[Fleisch1983]: Brett D. Fleisch, Operating Systems: A Perspective on Future Trends, Operating System Review, Vol.17 1983 pp.14-17.

[Goscinski1991]: Andrzej Goscinski, Distributed Operating Systems, The Logical Design, 1991, Addison-Wesley.

[Guillemont1990]: Marc Guillemont, CHORUS: A Support For Distributed and Reconfigurable Ada Software, CS/TR-89-40.2, Chorus systemes, 14 November 1990.

[Kernighan and Pike1984]: Brian W. Kernighan and Rob Pike, The UNIX Programming Environment, Prentice-Hall, 1984.

[Kindberg et al.1987]: T.Kindeberg, A.V. Sahiner, Y.Paker, Worm Programs, NATO ASI Series, Vol. F28, Distributed Operating Systems. Theory and Practice, Edited by Paker et al. Springer-Verlag Berlin Heidelberg 1987.

[Kutti1984]: Swamy Kutti, Why a Distributed Kernel? Operating Systems Review, Vol 18, No 2, October 1984, pp.5-11.

[Litman1988]: Ami Litman, The DUNIX Distributed Operating System, Operating Systems Review, Vol.22, No.1, January 1988, pp.42-51.

[Morris et al.1986]: James H. Morris, Mahadev Satyanarayanan, Michael H. Conner et al. Andrew: A Distributed Personal Computing Environment, Communications of the ACM, March 1986 Vol.29 No.3 pp.184-201.

[Mullender1988]: Sape J. Mullender, Distributed Operating Systems, State-of-the-Art and Future Directions, pp.57-66, Research into Networks and Distributed Applications, R. Speth editor, Elsevier Science Publishers, 1988.

[Renesse at al.1988]: Robbert van Renesse, Hans van Staveren, MANDIS/Amoeba: A Widely Dispersed Object-Oriented Operating System, Research into Networks and Distributed Applications, R, Speth Editor, Elsevier Science Publishers, 1988.

[Rickard1989]: Doug Rickard, Project Athena at MIT, An Experiment in networked Workstations, Digital Equipment Corporation, Brisbane Australia.

[Rozier et al.1990]: M.Rozier, V.Abrossimov, F. Armand et al. Overview of the CHORUS Distributed Operating Systems CS/TR-90-25, Chorus systemes April 15, 1990.

[Russell 1990]: Stephen Russell, Single-User Capabilities in Interprocess Communication, School of Computer Science and Engineering, University of New South Wales, email: smr@spectrum.cs.unsw.oz.au.

[Russell1991]:Stephen Russell, Object Support in the Styx Operating System, School of Computer Sciences and Engineering, University of New South Wales, email: smr@spectrum.cs.unsw.oz.au.

[Shrivastava1982]: S.K.Shrivastava and F.Panzieri, The Design of a Reliable Remote Procedure Call Mechanism, IEEE Transactions on Computers, Vol C-31, No 7, July 1982, pp.692-697.

[Silberschatz et al.1991]: A. Silberschatz, J. Peterson, P.Galvin, Operating System Concepts, 3rd Edition, 1991, Addison-Wesley.

[Smith1988]: Jonathan M. Smith, A Survey of Process Migration Mechanisms, Operating Systems Review, Vol 22, No 3, July 1988, pp.28-40.

[Souza and Miller1986]: Robert J. Souza and Steven P. Miller, Unix and Remote Procedure Calls: A Peaceful Coexistence? Distributed Computing Systems IEEE 1986.

[Tanenbaum and Renesse1985]: Andrew S. Tanenbaum and Robert Van Renesse, Distributed Operating Systems, Computing Surveys, Vol. 17, No. 4, pp.419-470, December 1985.

[Tanenbaum1987]: Andrews S. Tanenbaum, Operating Systems, Design and Implementation, 1987, Prentice-Hall.

[Tanenbaum1989]: Andrew S. Tanenbaum, Computer Networks, Prentice-Hall 1989.

[Tilborg1987]: Andre M. van Tilborg, Critical Research Issues in Distributed Operating Systems, IEEE 7th International Conference on Distributed Computing Systems, 1987.