Resource sharing across heterogeneous networks

Scott Edward Hazelhurst

A dissertation submitted to the Faculty of Science, University of the Witwatersrand, Johannesburg, in fulfilment of the requirements for the degree of Master of Science.

Johannesburg 1988

Abstract

Sharing resources on a computer network, especially in heterogeneous environments, has many benefits: new applications become possible, and use of technology cheaper. This dissertation investigates how resources—in particular printing resources—may be shared.

While still incomplete, an existing theoretical framework for data communication and resource sharing, *the ISO-OSI Reference Model*, provides useful background information and tools for analysis.

A discussion of this framework complements a survey of the principles and current state of file and printer servers, and distributed systems. An analysis of the design and implementation of a printer server acting as a bridge between two networks illustrates problems and results found in distributed systems generally.

The dissertation concludes by analyzing the strengths and shortcomings of the Reference Model and distributed systems. This and developments in technology lead to a proposal of an extended model for printer services, and clarification of printer servers' needs and requirements.

Declaration

I declare that this dissertation is my own, unaided work. It is being submitted for the degree of Master of Science in the University of the Witwatersrand, Johannesburg. It has not been submitted before for any degree or examination in any other University.

Scott Hazelhurst

Scott Hazelhurst
4th day of February 1988

Let us speak of the light that comes with learning...

Preface

There are many people whom I owe thanks. Foremost are my parents, David and Ethel, whose support and encouragement in every way I have always valued.

My supervisor, Kees Hoogendoorn, has been very helpful. I am grateful for his assistance and advice over the last two years, and especially over the last six months.

Other members of the department — staff and postgraduate students — have made my working environment pleasant and stimulating. In particular, I want to thank Philip Machanick who read drafts of this dissertation, and Conrad Mueller, who helped me debug the MAT.

My friends have always supported me. I especially wish to thank Lili and Annette for their friendships.

Table of Contents

1. Introduction 1
1.1 Justification 1
1.2 Local-area networks 2
1.3 Applications 2
1.4 Resources 3
1.5 Aim of the research 4
1.6 Servers 5
Message passing 7
1.7 Hardware/Technological considerations 8
1.8 Outline of dissertation 8

2. Protocols and internetworking **10**
2.1 ISO-OSI model 11
Physical layer 12
Data-link layer 12
Network layer 13
Transport layer 13
Session layer 13
Presentation layer 14
Application layer 14
2.2 High-level protocols 14
File Transfer, Access and Management Protocols 17
2.3 Internetworking 21
2.4 Summary 23

3. Servers and Distributed Systems **24**
3.1 Introduction 24
3.2 Design principles for file and printer servers 25
Access control 26
Unit of data access 27
Atomic update 28
Concurrent access 29
Reliability and Robustness 31
Printer servers 31
3.3 Survey of existing servers 33
3.4 Distributed systems principles 35
3.5 Survey of existing distributed systems 38
3.6 Analysis 43

4. A case study—MAT **46**
4.1 Problem description 46
4.2 The MAT — Requirements 49
4.3 General design considerations 52
4.4 Design and implementation 54
Performing service for a PC-net client 56

Performing service on the AppleTalk 57
Protocol conversion 60
Reliability and Robustness 63
4.5 Summary ... 65

5. Assessment ... 66
5.1 Assessment of design decisions ... 66
5.2 Synchronization and interprocess communication ... 68
5.3 PostScript ... 70
5.4 Reliability and Robustness ... 72
5.5 Performance ... 74
5.6 The layering of protocols ... 78
5.7 Extensions, Criticisms and Changes ... 79
5.8 Summary ... 82

6. Conclusion ... 84
6.1. The ISO-OSI Reference Model ... 84
6.2. Distributed Systems ... 86
6.3. Printing Services ... 88
6.4. Summary ... 93

Appendix A. NETBIOS ... 94
A.1. Commands ... 95
General 96
Name support 96
Session support 97
Datagram support 97
A.2. Remote program Load ... 98

Appendix B. ... 99
B.1. AppleTalk protocols ... 99
Link Access Protocol 99
Datagram Delivery Protocol 100
AppleTalk Transaction Protocol 100
Name Binding Protocol 101
Printer Access Protocol 101
B.2. Macbridge ... 102

Appendix C. PostScript ... 105
C.1. PostScript ... 105
General 105
Drawing figures 107
Text 108
Summary 108
C.2. Adobe structuring convention ... 110

Appendix D. PostScript protocol coversion ... 114

References ... 117

List of figures

Figure 1.1 — Synchronous and asynchronous communication 7
Figure 2.1 — Application and Presentation Layer 17
Figure 2.2 — OSI File Service 20
Figure 3.1 — Summary of choices 31
Figure 3.2 — Migration in the V-system 43
Figure 4.1 — Configuration of the MAT 50
Figure 4.2 — NCB request block 53
Figure 4.3 — Outline of MAT Code 55
Figure 4.4 — *Servicing AppleTalk clients* 58
Figure 4.5 — An interaction with a PC-network client 61
Figure 5.1 — Printing times for LaserWriter 69
Figure 5.2 — Printing times for Proprinter 69
Figure 5.3 — State diagram for PC-network clients 73
Figure 5.4 — The effect of I/O buffer size on I/O speed 76
Figure 5.5 — Dialogue between user and client code 80
Figure 6.1 — The Printing Machine 90
Figure B1 — Data structure for ATP 104

1. Introduction

1.1 Justification

The 1970s saw the growth of individual computing applications. The development and cheap availability of personal computers allowed, for the first time, these individual computing applications to be cost effective. More and more applications which previously required large mainframes became easily accessible to microcomputers.

A second trend was the development from the mid-sixties of the networking of computers (usually minis or mainframes). By the time microcomputers were widely used, networking was a fairly mature branch of computer science.

An advantage that mainframes continued to have over microcomputers was that mainframes allowed sophisticated communication between users, and the sharing of peripherals. The latter reason is significant: microcomputers have brought down the cost of processing, and the cost of all computer hardware has dropped. However, reserving the use of an expensive printer for only one personal computer usually cannot be justified. A solution is the use of networking to allow communication between different computers and the sharing of expensive equipment or 'resources'.

Networks of small computers have become common, with sophisticated sharing of resources and communication between them. In many working environments, the wide variety of equipment in use complicates this communication. It is feasible that an organization could have two (or more) networks running in the same environment. Particularly in smaller organizations, the need and desirability of communication across different networks exists. This dissertation examines issues of resource sharing across heterogeneous networks.

1.2 Local-area networks

A network is an interconnected collection of autonomous computers [Tanenbaum 1981]. The network allows these computers to communicate with each other. There are many different types of networks, using different physical media to connect the computers, stretching over different geographical areas, used for different applications, with many different types of computers connected to them.

The distinctive features of local-area networks (LANs) are (i) they cover a small geographical area, a few square kilometres at the most, (ii) the high data transfer rate, 0.1-100 Mbps or more and (iii) a low error rate. These features allow simplified, fairly cheap designs and greater flexibility and possibility of services [Stallings 1984]. The typical physical connection between nodes on a LAN is provided by twisted pair, coaxial cable, or optical fibre.

1.3 Applications

Perhaps the greatest application of networking is making computing far more accessible and flexible. There are a number of applications which while theoretically possible, only become practical with the use of networking. Local-area networks are important because they are cheap and allow easy communication between local resources. They are also important because they can be the gateway to longer distance communication: either with other local area-networks, mainframe computers, or computers on the other side of the world.

Desktop publishing is a well known example of a use of networking which makes computing accessible. Newspapers or magazines may be produced by fairly cheap technology, the equivalent of which a few years ago would have cost millions of rands. This case is put forward in [Manein 1986]. A personal observation is that not only the cost of the technology is reduced, but because the users of the technology will have experience with the fairly user-friendly environments of personal computers, the type of service offered by the technology (hardware and software) will be changed. The social effects of this can be positive.

In any society, knowledge and the ability to communicate ideas is power.

Desktop publishing allows those who previously could not compete with the established media the possibility of being able to start competing. In South Africa this is particularly true, and there is evidence of the AppleTalk network with Macintoshes and the LaserWriter being popular with people who have this need [Leah 1987].

Office automation is another application of networking. There are various claims that office automation will lead to improved productivity, the removal of routine work, and greater control of work. However, the issues are not clear cut. Hirscheim reviews this area in detail [Hirscheim 1986].

Other applications are electronic mail, banking and financial applications, distributed information systems, and computer conferencing. It has even been suggested that once the use of networking is more prevalent, that computers could be used to enable people to have greater control of their own lives by having a direct say in government (participatory democracy) [Mayne 1986].

Unfortunately, like most technology, networking can have detrimental effects. Privacy is a topic for much concern; the United States which seems to have the most safeguards in this area still has alarming practices [Rosenberg 1987]. There are other examples of negative impact of networking, and more areas of concern [BSSRS 1985; Mayne 1986]. This issue is not addressed again, not because it is unimportant—all scientists have the responsibility for the work they do and its political implications—but because the purpose of this dissertation is to examine the technical issues.

1.4 Resources

A resource is a service which is provided by one part of a computer system for another. A resource can be hardware (a printer or a modem), or software (compiler). Section 1.6 will examine a mechanism for making resources available to users on the network.

In a computer network, there are a number of possible resources. Secondary memory and I/O devices are common examples of relatively expensive hardware which should be shared by a number of users to make their use affordable. For example, a laser printer might cost about R20 000.

Such a high cost could be shared if a large number of personal computers and their users could have direct access to the printer [Janson et al. 1983].

The use of some resources can be economically justified in many circumstances only if they are shared by a number of users. This is not a technical limitation, but a practical one — there is no technical reason why each user on a network should not have a hard disk and a laser printer. However, there are some important applications which need to have shared resources [Mitchell and Dion 1982].

A distributed database is one type of application (e.g. for a booking system which has users all over the country, or a hospital information system [Jackson et al. 1986]). For this type of application to work, common data must be shared. Each user on the network needs to see the same common data. Even if each user has a personal copy of all the data, there must be sophisticated communication between the users to keep the data consistent, no matter how crudely this is done. The same problems which any data base system faces in this regard must be faced by a distributed system.

1.5 Aim of the research

This dissertation is an exploration of how resources can be shared, with particular emphasis on printing resources. There are three phases to the dissertation:

- The more sophisticated sharing of resources across different networks is fairly new, and most advances have come through practical research and experimentation. Nevertheless, there has been substantial theoretical work in some areas. This work will be discussed because it provides useful background material, and tools for analysis.
- The present state of particular types of resource-sharing systems will then be surveyed. In the light of this discussion, the design and implementation of a system will be presented, highlighting important principles and issues from previous discussion.
- The final phase concludes the dissertation by analyzing overall

trends, and examining which issues — both theoretical and practical — need further work. Previous discussion allows the clarification of some problems.

The rest of this chapter presents background material for the rest of the dissertation.

1.6 Servers

Servers are processes which provide some service to other parts of a system; they are the collection of software and hardware which gives users of the system access to particular resources ("a hardware and/or software resource designed and designated to provide a specific service to a user community" [Abrams 1985]). Although it is one logical process, it can take many forms. The software which implements it can be distributed on many machines [Janson et al. 1983]. It may be implemented on one machine, which may or may not be dedicated to that task. A number of copies of the server may be present on several machines, with only one active at any one time — the others ready to take over if the active one fails.

The server's implementation affects its reliability, efficiency and security, rather than the way in which a server should be conceptualized. Details of reliability and efficiency will be examined more later.

The framework in which servers are designed is the client–server model. A 'client' — some user or process on the system — requests a service from the server which provides the service. The motivation for servers are that they promote fairness, efficiency, sharing and transparency of service *[Berglund 1986]*.

Fairness

By centralizing the access to a service it is possible to ensure that all users have fair access to the resource. In this context, the server acts as the operating system of the network, and the strategies which an operating system uses can be adapted by a server.

Efficiency

While it is true that any system which administers resources uses some of the resources in the process of administering them, there are some aspects

of servers which promote efficiency. In a large system, with many processors, particular jobs may run more efficiently on some processors than others. A server which can allocate processors to jobs according to their needs can provide a more efficient service than a human user explicitly allocating jobs to processors may do. This planned use of resources leads to more efficient utilization of resources. Using a server also allows analysis of a system's behaviour. Over a period of time, tuning a system on the basis of information obtained is beneficial.

Sharing and transparency

A server makes a resource easily accessible. The operations which manipulate the resource may be intrinsically quite complicated. The detail of this manipulation is hidden by the server. Sharing also can be promoted by the fact that the server will generally define a common representation of the resource. All users of the resource will have to understand how this representation works and what it means. This standardization increases the availability of the resource.

Transparency means that users of a resource access the resource in the same way as they would access a local resource. This has important implications for the design of networks, their implementation and extensibility.

A final point worth considering is that by only allowing one way for a resource to be accessed, some control of the use of resources can be enforced.

These and other considerations (including user needs and hardware availability) determine many design decisions which affect relative efficiency and reliability etc. One of the most important decisions to be made is whether the communication between the client and server should by synchronous or asynchronous [Tanenbaum and van Renesse 1985].

With the former paradigm, the communication can be described as a remote procedure call. The one communicator issues a procedure call (in the same way as any other procedure call) which is executed by the site at which the other communicator is resident. Execution at the caller is suspended until the callee finishes executing the called procedure. Typically, the procedure call would have the effect of providing the

requested services. Any relevant information would usually be sent or returned as parameters. The implementation of this can be found in [Notkin et al. 1987].

The second option — asynchronous communication — can be explained by message passing. One communicator sends a message to the other. The caller then can continue processing. The callee receives the message and processes it in some way. This processing may require that some message be passed back to the caller. Synchronization of this communication is implementation dependent, and requires careful design.

The explanations in the above two paragraphs are to some extent simplistic. How message passing and remote procedure calls are implemented in a network can be hidden from a user. Aspects of this will be discussed in detail at a later stage. The two methods are illustrated in figure 1.1.

Example

A network may have a real-time clock. A user wanting to obtain the time has to request the time from the time-server.

Remote procedure call	
client	server
issues request:	
time _request(time);	processes the request
execution halts until the server responds	transfers control back to the client
Message passing	
client	server
sends message asking for time	gets request
ask_for_time;	finds time
continues processing	sends message to the client
at a later stage gets time	
receive time	

Figure 1.1 — Synchronous and asynchronous communication

1.7 Hardware/Technological considerations

This section is a list of a few of the technological considerations which must be borne in mind.

- The cost of storage on floppy disks is still far more expensive than that on a hard disk. Disk access time is much faster, and hard disks are much more reliable than floppy disks.
- Other peripherals — e.g. printers — are still relatively expensive and need to be shared.
- Bandwidth is getting cheaper. Optical fibre local-area networks are becoming available.
- While secondary memory is becoming far cheaper, there are still applications where the amount of secondary memory is limited because of cost. There is increasing usage of applications which need to share data.
- Balancing the benefits of sharing the costs of some resources is the cost of the network equipment itself. In some makes of networks, this may be substantial. In the rest of this dissertation, however, it is taken as given that the use of networks does reduce the cost of computing, besides allowing the use of distributed applications.

1.8 Outline of dissertation

The purpose of this chapter is to set the context for examining how resources can be shared across networks, first of all explaining why it is necessary or useful, then what resources are and general principles involved, and finally to explain technological considerations. The rest of the dissertation is as follows.

Chapter 2 presents existing theoretical aspects of the area. Protocols — the rules by which communication takes place — are discussed in the context of the International Organization for Standardization Reference Model on Open Systems Interconnection (ISO-OSI Reference Model), and the impact of connecting different networks is examined. This context provides the necessary background material for the rest of the dissertation. Some of the concepts which are introduced here are useful in analysis and design. Where this theoretical

work is incomplete or inadequate will be pointed out here and in subsequent chapters.

On a single computer, the operating system provides the interface between a user and the resources of the computer. On a network, where resources are distributed, some sort of distributed system software is necessary to provide this function. Chapter 3 surveys the current state of distributed systems and servers. Links to the theoretical work discussed previously are made. This investigation clarifies what issues and principles are important in the design and implementation of resource-sharing systems. This field is characterized by the use of experimentation to develop principles and ideas; it is through the design, implementation and analysis of real systems that the area grows.

Chapters 4 and 5 present and assess a case study: a printer server which provides printing resources on two different networks. This printer server — which is an example of a fairly unsophisticated resource-sharing system — is designed to illustrate some of the points made in previous chapters. Some of the important principles, problems and results of distributed systems are clearly shown by this work.

The conclusions presented in chapter 6 assess

- the strength, weakness and applicability of the existing theoretical models,
- the needs and requirements of distributed systems in general, and printing services in particular,
- the success of general methods for coping with heterogeneity, and
- what future work needs to be done.

2. Protocols and internetworking

While the thrust of much research in resource sharing has been of an experimental nature, there has also been theoretical work. The models which at present form the basis for communication and resource sharing are not complete and are still developing. Nevertheless, there are certain areas where more theoretical research has been extensive and influential.

This chapter examines the theoretical models for two reasons. The theory is useful background for further discussion and debate. This also makes analysis and assessment easier. A second reason why it is included here is that later chapters in the dissertation show where the model is incomplete and needs extension, and where such work would be beneficial in the design of real systems.

Protocols are the sets of rules which regulate how communication should take place between communicating entities. Just as humans have different elaborate sets of rules for communicating in different circumstances, computers do too. The chapter discusses computer network protocols and their implications. The search for network protocol standards is an important area of research. The greater the variety of equipment and applications on a network, the greater the potential for problems in communicating.

Computer network protocols are the operating rules and procedures for conducting communication between different network users and devices. 'Communication' is not one concept: at one level it may mean the transmission of bits across a certain type of line, at another level it may be the transfer of a high-level construct or concept. As with all computer programs, dealing with different levels of abstraction can be difficult. It is for this reason that a layered model of protocols is suggested for computer network communication, with different layers being responsible for different levels of abstraction

This chapter is divided into three sections. The first describes the ISO-OSI Reference Model. The second examines high-level protocols, and the third the influence of internetworking on protocols generally.

2.1 ISO-OSI model

For any hierarchy of protocols, there should be a number of common principles. The purpose of the hierarchy is to reduce complexity at each level. Each layer takes some of the services offered by the layer below, adds some services to them, and offers these 'value added' services to the layer above [Linington 1983]. A process at each layer communicates with a process at its corresponding level. This peer communication is virtual communication for all layers except the bottom layer. A process at level n wanting to communicate with another process at level n gives its data to the next layer down, which performs some transformation on the data, and sends it to the next layer down. It is only at the bottom layer that the physical communication takes place. At the other computer, the reverse process takes place. The data is received at the bottom layer which performs the reverse transformation, and passes the data up to the layer above. This process is repeated until layer n is reached. A formal definition of this can be found in [Lam 1986].

The most common and widely-referenced model for computer network protocols is the International Organization for Standardization Reference Model for Open Systems Interconnection (ISO-OSI model) [Zimmerman 1982; Day and Zimmerman 1983]. A subcommittee of the ISO started working on this model in the late seventies, and by the early eighties had published the model. Work is continuing on defining protocols for each of the layers of the model, particularly the higher levels.

The ISO-OSI Reference model is examined here as example of a protocol hierarchy because it is the best known model. There are a number of other models [Mayne 1986]. The dissertation presents a critique of the ISO-OSI Reference Model as an example of a standard layered model rather than a comparative analysis of different models and standards.

The ISO-OSI model had a number of specific design principles [ibid.]:

- there must be another layer wherever a different level of abstraction

is needed

- each layer should perform well defined functions
- the function of each layer must facilitate the definition of internationally standardized protocols
- layer boundaries should be chosen to minimize information flow across interfaces
- the number of layers should be large enough so that distinct functions are in distinct layers, but not so large that they become unmanageable

The aim of the ISO-OSI model is to facilitate open systems interconnection: this would allow different systems to communicate with each other effectively, as long as they were designed within the parameters of the reference model. In the rest of this dissertation OSI means open systems interconnection. There are seven layers in the ISO-OSI model which are briefly outlined here.

Physical layer

The physical layer is concerned with transmitting raw bits over the communication line. Protocols at this level are responsible for ensuring that if a transmitter sends a '1', the receiver receives a '1'. Decisions like what particular voltages represent, and whether the communication is broadband or baseband are determined here.

There is also a choice to be made how the network will be switched — how the communicating parties will be connected physically. The two main categories are circuit and packet switching, although variations and alternatives do exist [Tanenbaum 1981].

Data-link layer

The data-link layer takes the services provided by the physical layer (the raw transmission facility), and provides the network layer with an error-free line. It breaks the data up into data frames and then transmits them on the physical channel. One of its main tasks is to provide a medium access protocol [Carlson 1982; Conrad 1982; Kurose et al. 1984].

Network layer

The network layer controls the operations of the subnet. It takes packets of data given to it by the transport layer, and forwards them to the appropriate node. On a general network, the network layer would be responsible for routing.

There are three sublayers here [Mayne 1986].
(1) The *subnetwork access layer* uses the existing data link layer directly to provide an abstract net.
(2) The *subnetwork enhancement layer* enhances the particular subnet to allow data transfer across it to meet required quality of service.
(3) The *internetting sublayer* is responsible for concatenation of subnets, together with global addressing and congestion control.

These latter two sublayers are mainly useful for dealing with internets, and will be dealt with in more detail later on in this chapter.

Transport layer

The transport layer is responsible for the end-to-end management of data. It is responsible for establishing and deleting connections across the network. It takes data from the session layer, splits it into smaller units or packets and then passes these to the network layer. The network layer is responsible for ensuring that the packets get to their destination — they might pass through a number of other computers. The transport layer is responsible for ensuring that all the packets arrive correctly at the other end. The transport layer can provide a datagram or virtual circuit service to the session layer.

Session layer

The session layer [Emmons and Chandler 1983] is the mechanism for organizing and structuring interaction between application processes. It is the way that the user or application program sees the network. These services can be provided to the presentation layer as either a virtual circuit, or a datagram service.

For convenience, in the rest of the dissertation, 'transport functions' refer to the functions of the network, transport and session layers (in some local-area networks, a protocol may be implemented at only one of these layers).

As will be seen in later chapters, the design and implementation of the transport functions make a significant impact on the performance of network systems [Watson and Mamrak 1987]. While the ISO-OSI model is a useful one to use in the design of a system, the implementation should take into account that layering in implementation can cause degradation in performance. The mechanisms of error detection and recovery, connection management, packet length and acknowledgment are important factors in the performance of the transport functions. Watson and Mamrak examine these issues in more detail. As computers' architectures and operating systems evolve and adapt for networking applications, the significance of the transport functions to overall system performance will increase.

Presentation layer

The general purpose of the presentation layer is to make the application processes independent of differences in data representation by performing functions that require a general solution. It provides any necessary transformations on the data.

Application layer

Users' applications reside at the application layer.

High level protocols (presentation and application layer protocols) will be examined in detail in the next section.

2.2 High-level protocols

Protocols in the layers between the physical and session layers provide the logical link between the communicating parties. The high level protocols — which are defined for the purposes of this dissertation to be those protocols in the presentation and application layers — are concerned with the sharing of the logical resources across the network and the managing of the computing process rather than with data communication itself [Sproull and Cohen 1978]. For the entities at this level, the fact that communication is taking place across a network or internetwork is irrelevant.

Some of the services which high-level protocols (HLPs) provide are remote job entry, file services, mail services, resource sharing, security, text compression and terminal handling [Tanenbaum 1981]. With the development of computer networks, the occurrence of heterogeneous networks and internetworks will become more common. It cannot be assumed that the computers on the network will have many features in common. Thus discussion of HLPs must take place in this context; meaningful communication must take place even though the underlying representation is completely different.

Three components of an HLP are:

- The *language* which describes the functional intent of the protocol, including the commands and other control information that the user would need to issue or know
- The *coding* which imposes structure on the stream of bits or bytes which the session layer provides.
- The *transport* which provides the communication protocol. The transport is implemented by the session and lower layers. Its functions will be used by the HLP to meet the requirements of the application [Sproull and Cohen 1978].

The ISO-OSI model describes the presentation layer as dealing with the representation of user information while between OSI systems. Together with the application layer, this layer describes shared information. The presentation layer describes the syntax, while the application layer describes the semantics [Hollis 1983]. For example, which character set the bytes represent is a presentation layer decision, while what they mean to the user is an application layer decision. The presentation layer facilities deal with connection establishment and termination, dialogue management and synchronization, information transfer and context definition. In Sproull's and Cohen's terms, all the facilities except the last one are transport ones — they are facilities of the session layer which are provided by the presentation layer to the application layer.

The context of the presentation layer includes what character set is being used or applicable at a particular time, or what particular bit patterns represent. Text compression, if necessary, would be performed at the presentation layer, and the context would define how it was done. Another

important element is encryption. As the ease of accessing data increases, the importance of safeguarding certain information also increases. This is an important area for work [Tanenbaum 1981; Voydock and Kent 1985]. These are examples of the coding element.

The presentation layer offers its services to the application layer which must assign meaning to the data presented to it. The application layer is divided into three elements, the user element, the common application service element, and the specific application service element [Bartoli 1983; Linington 1983]. The application layer protocols define the agreement of application entity communicators on the semantics of information exchange [Bartoli 1983]. The relation between these elements and the presentation layer is usefully shown in figure 2.1 [ibid.].

The common application service elements deal, in the main, with the transport functions: the establishing, managing and synchronizing of connections. One of these service elements allows the definition and selection of specific application elements. A service provided at the higher levels, therefore, would be a set of service elements: a particular user element, together with selected specific application service elements and the common application service elements. Communicating application entities would then use the appropriate protocol. For example, a file server and its client would both be application layer entities. The server would use the protocol to communicate with its client and thereby provide the file service.

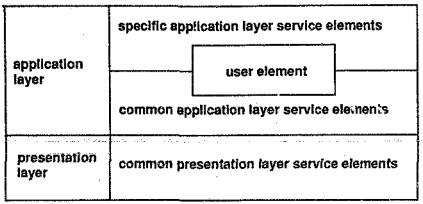


Figure 2.1 — Application and Presentation Layer

These specific application elements describe the functional intent (the language) of the protocol. There are a number of guide-lines for designing HLPs. Wherever possible, the protocols should be device independent (not rely on the characteristics of any particular hardware), and be independent of intervening networks [Lantz and Nowicki 1984]. Simplicity, generality and robustness are also important. When resolving the tension between generality and simplicity, the dictum "a design that provides a special feature in a device independent way is worthless if it is never used" should be kept in mind [Sproull and Cohen 1978].

As there will be a wide variety of equipment and system software on most networks, providing for device independence needs careful thought. There are two basic approaches to this problem. The first approach is to define a virtual device or representation. There then must be a mapping mechanism between all real devices and this virtual device. The alternative is the parametric approach. The device is controlled by setting various parameters. The latter approach has proved to be quite acceptable with terminals. However, with the development of OSI, it seems that the virtual device approach is more suitable [Day 1980]. Others have qualified this [Notkin et al. 1987], pointing out that adopting a single standard may lead to unnecessary inefficiency, and have advocated some potential for negotiation at bind time.

The rest of this section will illustrate the concept with discussion on file transfer protocols. File transfer protocols will also lay a useful basis for discussion in future chapters as file servers use file transfer protocols extensively.

Virtual Terminal Protocols (VTPs) are another important class of HLPs. They are related to printer servers because many of the issues which VTPs deal with in reproducing graphics or text must be dealt with by printer servers. Discussion on these may be found in other work [Day 1980; Lantz and Nowicki 1984; Lowe 1983; Mayne 1986; Tanenbaum 1981]. The analysis of a printer protocol in later chapters will use some of this work.

File Transfer, Access and Management Protocols

Conceptually a file is an entity with the following [Linington 1984]:

- naming attributes
- other descriptive attributes
- attributes describing logical structure and dimension
- associated data

All of these need to be represented by the file transfer, access and management protocol (FTAM), with mechanisms for managing collections of files. The next chapter will discuss the design decisions which have to be made in greater detail, and in a more concrete way. The point of this section is to describe an abstract model in which this may be done using the ISO-OSI model.

The OSI File Service can be described as defining a standard for transferring, accessing and managing information stored in or moved between open systems as files [Lewan and Long 1983]. It is the aim of the OSI file service to allow otherwise incompatible filing systems to work together. These protocols could be used by humans or by applications.

The OSI defines a virtual filestore which allows differences between local representations to be absorbed into local mapping functions, provides different services for different users, and allows a variety of different systems to be implemented [Linington 1984]. File transformation deals with the way in which files are represented. As stated earlier, the session layer just recognizes a stream of bytes or bits — it is the HLPs which are

responsible for assigning structure and meaning to these bytes. By defining a virtual filestore, a canonical representation of the file is developed, i.e. a representation of the file which is meaningful to all users of the file is defined.

At the syntactic level this will be dealt with at the presentation layer — which character sets are used, what control characters mean, and what conventions are used to represent digitized images. The protocols which do this will be invoked from the application layer.

How the files are stored at a macro level, and what they mean is the task of the application layer protocols.

Consider an application — the user — which wishes to access a file at a (remote) site. This file will be stored by a local file system in its own format. A local mapping function will be able to map between this and the virtual filestore's representation. An application entity at this site will communicate with the user's site using the relevant file transfer protocol. Users see this communication as the file transfer services, which is diagramatically represented in figure 2.2 [Aggarwal et al. 1985]:

The naming and other attributes are used to identify the file, and provide descriptive information (date of creation, modification, name of initiator etc.). They would also describe the file's access structure, and possibly give information like who can have access to the file and the mode of encryption. Work on the standardization of directory systems by ISO is in progress [Goodwin and McDonnel 1986].

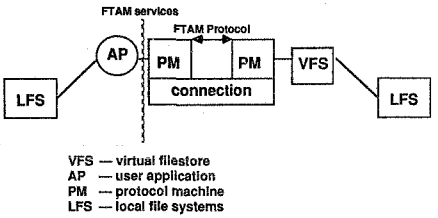


Figure 2.2 — OSI File Service

The virtual filestore also provides for the structuring of the subunits of the file. Abstract operations are defined for manipulating the entire file, and its contents.

A session between a user of the file service and the file service goes through several distinct phases [Lewan and Long 1983]: the file service initiation phase, file selection phase, file management phase, file access initiation phase, data transfer, file access termination phase, file deselection phase, and file service termination phase. The actual services provided could be agreed on between two communicators [Aggarwal et al. 1985].

As will be seen later, few real systems delineate the phases described above as rigorously as this. There are reasons though why it is useful to use a model like this. Firstly, a model is useful when designing and conceptualizing a system to help reduce complexity and generally make good design choices. This does not mean, however, that the implementation of a system needs to correspond exactly to this model. Another factor is that each of these phases is responsible for some tasks — tasks which are likely to have to be carried out. In analyzing the system, the model can be used as a tool to understand and improve what the system is doing. Device independence may also be promoted by adhering to the model closely.

This OSI file service description provides no more than a framework for

building a real system. Many of the real problems do not appear at first sight. The presentation layer services are not necessarily straightforward. If a file is being stored at a remote site merely for storage purposes, then it is quite acceptable for the file to be transferred bit by bit to that site, and recovered in the same way. However, consider the problem of a data file which could be run by different computers. The data could include floating point numbers or abstract data types — stored on different computers quite differently. Two bytes with the same bit pattern might mean completely different things.

Furthermore, there are decisions about the file service which fall outside the scope of the OSI model of file service. In the next chapter, some of these decisions will be discussed. The processes which implement these decisions are application layer protocols. However, work has not progressed very far on standardizing these protocols.

2.3 Internetworking

A natural progression from connecting computers together is connecting networks together. The same motivations for networking apply to internetworking. Research into internetworking aims at providing a uniform framework for heterogeneous computing where machines and networks differ in physical characteristics and geographical location [Boggs et al. 1980]. With a well defined model, models for internetworking can develop naturally from those for networking, as can be seen in [Lam 1986].

The level of the ISO-OSI model at which interconnection takes place depends primarily on the networks involved, and their physical interfaces. *If interconnection takes place at level n, then the networks involved must* share common protocols for all levels above level *n*. Interconnection at the physical and data-link layers is straightforward because at that level the mapping between two approaches is fairly simple. Interconnection at the presentation and application layers with different protocols is difficult, if not impossible. This is not because of any of the physical characteristics of the networks involved, but because there is increased state information about the different protocols, and there being much less probability of a

one-to-one relationship between these protocols — "success in protocol translation seems inversely correlated with the protocol level" [Postel 1980].

Gateways, which connect different networks, can either be computers dedicated to the task of bridging the networks, or simply hosts on more than one network which are prepared to pass packets between networks. There are two approaches to the tasks of gateways [ibid.]:

- Some gateways take messages from one network, unwrap any 'packaging' in that network's format, wrap the packet in the destination network's packaging, and then pass the packets to the destination network.
- Others translate the protocol by replacing packets from the source network with different packets with the same protocol semantics.

Internetworking does not introduce fundamentally new problems to networking. It certainly complicates matters, and requires careful thought. Most of the work on internetworking has been done at the network and transport layers of the *ISO-OSI* model. Corresponding to the two approaches mentioned in the previous paragraph are CCITT's [Gien and Zimmerman 1979] and ARPA's models [Cerf and Cain 1983].

CCITT's X.75 recommendations which is based on the X.25 protocol establishes a cascaded virtual circuit between the two communicating entities. Each network on the internet provides a virtual circuit for its part of the journey. As not all networks provide a virtual circuit service, some networks may have to be enhanced — the internetting and subnetwork enhancement sublayers of the network layer mentioned earlier would be responsible for this.

ARPA's research networks have adopted the approach of providing a datagram service as the Internetwork Protocol (IP) at the network layer. The Transmission Control Protocol at the transport layer provides end-to-end service.

The problems which the above two approaches try to solve are complications of problems of networking. Addressing and routing within the internetwork are problems which have not yet been satisfactorily resolved. Having fixed routing through an internet is more reliable, but ties down

resources needed while the virtual circuit is up. On the other hand, variable routing leads to sequencing problems whilst perhaps making better use of resources. Related to this is the problem of fragmentation. A packet going through different networks may have to be fragmented, and those fragments fragmented. The reassembly of these packets is not straightforward [Gien and Zimmerman 1979; Tanenbaum 1981].

In assessing the impact of internetworking on resource sharing it is worthwhile noting that problems caused by interconnecting heterogeneous networks are the domain of the lower layers (session layer and below). Among others, these problems are dealing with different physical media for communication, and different protocols for routing, fragmentation etc. However, at the presentation and application layers the problem of networking heterogeneous networks is reducible to the problem of a network with heterogeneous application entities, as these layers are responsible for resource sharing rather than data communication: the added problems of internetworking will be solved by the lower layers.

2.4 Summary

This chapter introduced a model for networking and resource sharing. The ISO-OSI Reference Model was discussed, paying particular attention to high-level protocols and file transfer protocols. This was related to internetworking and its problems. Some of the possible weaknesses of the model were noted, including the problem of excessive layering, the inefficiency of a single standard, the incomplete nature of the model, the difficulty of protocol conversion at the higher layers and the problem of addressing, routing and fragmentation in internetworks.

The next three chapters examine network systems more concretely, and in detail, including a case study. This leads to an assessment in chapter 6 of the usefulness of the theoretical work for real systems.

3. Servers and Distributed Systems

3.1 Introduction

This dissertation has so far explained, in a general way, what resources are, why sharing of resources is important, and has also described a general framework for communication. The purpose of this chapter is to explore existing systems, and the design and other decisions which need to be made, bearing in mind the experimental nature of the field..

There are different philosophies and ways of implementing distributed systems. This chapter looks first at the specifics of file and printing services before examining distributed systems — networks of computers in which resources are shared — more generally. File and printing services can stand on their own, as will be seen in the case study analyzed in the next two chapters. For those just interested in these aspects of this dissertation, it is unnecessary to look at more general cases More importantly, file and printing services are an integral part of distributed services, and are therefore a good way of introducing the topic.

Before describing servers and distributed systems, it is useful to place these things in the context of the ISO-OSI model examined in chapter 2. Some of the decisions which have to be made are related to what transport functions should be provided (i.e. what services should be provided by session layer and lower protocols). The majority of the decisions and and protocols discussed are related to the presentation and application layer protocols. The standardization of distributed systems as a whole is not complete, an aspect which will be covered in more detail in chapter 6.

3.2 Design principles for file and printer servers

Broadly, file servers have two purposes: to provide users of a system reliable, long-term storage of files (done for cost and efficiency), and to allow files to be shared. 'Sharing' can mean different things. For example, a compiler may be shared by all the users of the system. Conceptually though, each user could have their own copy of the compiler — it is shared because it would be too expensive to provide each user with their own copy. Sharing can also be dictated by the needs of a particular application rather than for the sake of reducing costs. If all the users of a file need to see the file in the same way as all the others do (if they need to see others' modifications of the file, and have others see their modifications), then a file has to be shared. Directories are good examples of files which need to be shared in this way.

The previous chapter's discussion of file transfer, access and management protocols covered the design considerations for file representation, and mapping between different file representations. This section covers, in a concrete way, how file servers provide file service. The file service is provided for within the framework of the OSI model; however, how it is accomplished is decided by each system.

Exactly what design decisions get made depends on what a file server's purpose is, and what it will be used for. File servers can be classified into three types: simple, universal and database management support [Janson et al. 1983], depending on their level of service.

A simple file server provides nothing more than remote storage, while a file server providing database support will have all the facilities of a filing system. A universal file server falls somewhere in the middle: it gives a client process some powerful tools, but does not dictate all the decisions — for example, the type of filing system — which the client must make.

The first decision which must be made is what service is needed, with what reliability and performance. With this in mind, some of the other issues are:

- access control
- unit of data access

- atomic update
- concurrent access

For example, a sophisticated database in a large office should provide access at the record (an arbitrary number of bytes) level while guarding data from unauthorized access, and would probably need to have a high degree of concurrency, while being highly reliable. However, a file server for a small school environment is likely to have lesser requirements. Access control may not be needed, while whole files would be uploaded or downloaded at a time, and there would be no need for concurrent access. A low degree of reliability would probably be acceptable.

Access control

All file servers collectively play the rôle in a distributed system which a filing system does in a centralized operating system. Operating Systems are responsible for ensuring that users of the system do not have access to files for which they have no authorization. There are two main approaches: capability and identity based control. Under capability based control, a client of the file server will be allowed access to a file if it can show permission (a capability) for the file. In this case, the server takes no account of the identity of the client; as long as the client presents a valid capability it will get access to the file.

An alternative is for the server to have a list of which clients have what access to which files. Thus, the server plays an active rôle in keeping control of access to resources — in the capability-based approach it is the clients which are responsible for ensuring that capabilities are correctly distributed.

Another way of protecting files is to provide no access control at all. Any client wanting to store a file on the server encrypts the file. Only clients knowing how to decode a file can make sense of it. While this cannot prevent malicious destruction of valuable files it can prevent unauthorized inspection of the file, and unauthorized modifications (as any modification will be easily detectable).

Capability-based and identity-based methods have been successfully implemented, and seem acceptable methods of imposing access control

[Mitchell and Dion 1982], but there are problems with both [Tanenbaum and van Renesse 1985]. While the capability-based approach seems easier to implement, especially with multiple file servers on a network, the identity-based approach does allow more flexibility in providing finer granularity of access control. It also seems that there may be problems in ensuring that capabilities do not fall into the wrong hands, especially when a file server is serving a number of connected networks. The encryption/decryption method may not be viable in terms of performance where parts of files rather than whole files are units of data access.

Unit of data access

The unit of data access varies between file servers. For those servers — perhaps better called disk or storage servers — which play no more a rôle than storing files remotely, accessing files as units is acceptable from the point of view of the client, and preferable as far as the overall design, efficiency and simplicity go. If a typical transaction is for the client to download an entire file, make any changes locally, and then send the *modified file back to the server, then this unit of data access is preferable,* as to provide a more sophisticated service would be an unnecessary complication.

A file server which provides a paged virtual memory system to its clients will need a page rather than the entire file as a unit of data access. This method has the advantage that if only a few changes are being made to a file, then only the relevant pages need be communicated over the network. More than providing a flexible service, page access raises the possibility of having intelligent workstations with no local secondary memory [Lazowski et al. 1986]. Even though the cost of secondary memory is decreasing rapidly, this is still important for two reasons. Even if each workstation on a network does have a large hard disk attached to it, it would be a major constraint if this disk were not accessible from other workstations (a user may then be tied to one machine). Further, there are still applications where the cost of secondary memory will continue to be significant in relation the overall cost of the system. An example of this will be discussed later.

Networks which connect more than one type of computer may have to

deal with more than one size of page.

A more generalized version of page access is to allow clients access to any sequence of bytes, and this allows the greatest flexibility [Svobodova 1984].

Atomic update

The issues of atomic update and concurrent access affect the complexity of design. They are related to the unit of data access, as the smaller the unit of data access, the greater the flexibility that can be offered. To examine these issues, the concept of a *transaction* has been developed. A transaction is a set of events which is atomic, i.e. the events as a whole are indivisible; either all of them happen, or none of them happen, and the data which it accesses will not be accessed by any other process while the transaction is being processed. These concepts are called atomic update and serializability respectively [Tanenbaum and van Renesse 1985]. In this section, atomic update is examined, in the next serializability.

The first question is mainly one of reliability: how safe should the system be against hardware and/or software failures? What is being assessed here is how the system should react if there is a failure of some sort while a client is accessing data. The rule is that files must always be in a consistent state So, if a client updates a file and the operation completes then the new, updated version of the file must be stored by the server. If there is a failure, the server must still try save the new version. If this is not possible though, the old version of the file must be stored. Another factor to consider is that a catastrophic event might permanently destroy a version of the file.

One way to implement reliability in this context is to use stable storage [Mitchell and Dion 1982]. Two copies of all data are kept. When data are modified, two physical modifications have to be made; the update to the second copy is only made once the update to the first copy has been made successfully. If there is a crash, the server can restore itself to a consistent state by examining both copies of the file it was accessing.

An alternative is to keep only a second copy of the data being modified [Dion 1984]. When a page is being updated, the server writes to a shadow page, and keeps track of its intentions. If the server crashes, it is abl to

recover to a consistent state by using the shadow pages and lists of intentions. A full description of this mechanism can be found in [Dion 1984; Walker and Popek, undated], and a comparison between the two in [Mitchell and Dion 1982]. There are a number of other mechanisms. For example, the server can write changes to a log. Once the user is satisfied with the changes, the server can update the file from the log. Even if the server crashes a number of times, a consistent state can be reached. A description of this and other techniques can be found in [Svobodova 1984].

Concurrent access

Concurrent access by several clients to the same file can also affect the consistency of a file. If client *A* is modifying a file while client *B* is reading it, it is quite possible for *B* to see an inconsistent version of the file. For this reason, the server has the task of ensuring that no concurrent access of a file leads to inconsistency.

The easiest way to prevent these unwelcome effects is to prevent any concurrent access of files. In many cases this is practical, and does not lead to degradation of service. Assuming this makes implementation of concurrent access comparatively simple.

Making such a simplification is not possible, however, in many cases. A distributed database might be unworkable if most data accesses were to one very large file. There is no need for whole files to be protected from other users if the changes being made are restricted to a small region of the file and the other users want access to unrelated parts of the file. Furthermore, there is no reason not to allow concurrent access to any number of clients who are only reading from a file. The unit of concurrency control (the portion of the file for which there must be concurrency control) is determined by the unit of data access only to the extent that the unit of concurrency control must be larger than the unit of data access.

The most common unit of concurrency control is the file [ibid.], but file servers which have page or dynamically variable size units also exist. Within the unit of concurrency control, there are several approaches: the single client (either writer or reader) within the unit of concurrency control; single writer or multiple readers within the unit of concurrency

control; and single writer and multiple readers within the unit of concurrency control. The last type is the most interesting. The writer updates a separate copy of the file, while the readers have access to the original. When the changes to the file are committed, the readers have the options of using the new or old version of the file.

Another approach is needed in the case of systems like the Locus distributed file system where several, replicated copies of the file may be present in a system on different servers [Walker and Popek, undated]. When a client wants to use a file it gets access to the one which it is most efficient for it to have. One of the servers which has a copy of the file acts as a synchronization site, and is responsible for ensuring that the appropriate restrictions on concurrent access are kept. The server which has the copy of file being used is responsible for ensuring that the file remains in a consistent state. If any changes are made to the file, this server must inform the other servers which have copies of this file that changes have been made. These servers in turn are responsible for ensuring that they update their versions of the files. A high level of transparency was found to help in implementing transactions [Weinstein et al. 1985].

Not only does the scope of an atomic transaction in a file have to be defined, but how many files can be involved in a transaction, and if more than one file can be involved in a transaction whether they can be on more than one server. By allowing more than one file on multiple servers to be included in a transaction, the efficiency of the filing system can be improved at the cost of increased complexity and the introduction of problems like deadlock.

As networks get more sophisticated, bigger and interconnected with other networks, the importance of distributed file systems (i.e. where a number of file servers together are responsible for the system) will grow. This is especially so for distributed file systems which are transparent — where the clients cannot tell where files are physically stored.

Reliability and Robustness

A discussion of how reliable storage can be implemented by a file server has already taken place. This is one aspect of reliability. Errors on the network and physical faults are other aspects. File transfer protocols must be correct. The discussion of how reliable a system should be will take place in a subsequent section in the context of distributed systems in general.

Of course, file servers also need to cope with possible crashes of the client machines, or faulty communication on the network. Being robust—able to tolerate errors—is another important property.

This has been a very brief overview of file servers. The design areas outlined have been concurrency control (unit of concurrency control, scope of concurrent transaction), unit of data access, reliability and access control). Figure 3.1 is a summary of the decisions and choices which can be made.

decision	choices
access	capability/identity based, encryption
unit of data access	file/page/arbitrary range of bytes
atomic update	stable storage/intentions list/logging
concurrent access	unit for locking; number of readers and writers allowed

Figure 3.1

Printer servers

The printer service defines a virtual printer which its clients see. The actual service is divided into two parts: spooling, and the control of the printing device [Janson et al. 1983]. These parts of the service may be implemented on one machine or on two machines, and can be considered separately for conceptual purposes. The spooler part is a simple extension of a file server. A file to be printed is sent to the spooler which then saves the file and does any queuing necessary.

An alternative approach would be for the client to send the spooler the name of the file to be printed. The spooler would then not save a copy of the file, but transfer it to the printing device when appropriate [Tanenbaum and van Renesse 1985]. This approach, however, demands some level of sophistication and multitasking from the client.

Usually, a client is relieved of any further tasks associated with printing as the server takes responsibility. The print device controller transfers the files from the spooler to the printing device.

During the operation of this service, protocol translation on the file can take place at three places.

- The client can do protocol conversion before sending it,
- the spooler can do protocol conversion before saving it, and
- the device controller can do protocol conversion before printing it.

The advantages and disadvantages of these different approaches will be seen in the particular case of the printer server described in chapters 4 and 5. In general, which is the best approach depends on the nature of the system and the typical print jobs which are run. For example, the spooler doing all the protocol conversion would relieve the clients from doing it. However, if there was much protocol conversion to be done, the printer server would become a bottle-neck in the system.

In general, the protocol conversion takes place twice. The client translates the file into the standard device-independent protocol, and the server translates from the standard protocol into the device dependent protocol for the specific printer.

It is also important for a printer server to report back to a user on the status of a print job. This could take place once the job had been spooled safely, when the job has completed, and when problems occur. The level of sophistication of the service, and the reliability of the system would determine where this happened.

In the next chapters, the design and implementation of a printer server is presented. In chapter 6, a set of requirements for printer servers will be proposed, paying attention to the question of where protocol conversion should be done, and the rôle of the spooler — this will include a proposed

extension to the model of Janson et al.

3.3 Survey of existing servers

This section will briefly examine some of the file servers, describing the design decisions which were made in their implementation.

Simple file servers

The Acorn File Server [Dellar 1983] is a file server designed to provide file storage on cheap networks, typically in a school environment, where the cost of disk drives for all the machines on the network cannot be justified.

The services which it provides are unsophisticated and simple. Although files can be accessed at the file or byte level, the file manipulation operations are only provided at a low level. No provision for concurrent access is made; users wanting to have concurrent access have to provide it themselves. The file server uses a version of stable storage to provide reliable storage of directory maps. No provision seems to be made for the reliable storage of any other data, and if the server crashes, all active data about its clients are lost. Access control is performed by the file server using a simple password system and capability-like handles.

A major constraint on the design of the system was the lack of features of the client machines — no secondary memory, and not much memory at all, emphasizing, in the context of a low-cost environment, the need for all protocols to be simple. This was accomplished by using a connection-less protocol, and relying on the idempotent nature of most of the operations. The system shows that reasonable performance can be achieved with little cost.

A similar approach was adopted in the construction of a Simple File Server (SFS) at the National University of Singapore [Srinivasan and Ananda 1986]. Like the Acorn File Server, concurrent access must be implemented by clients. Files are the units of data access. A checkpoint/recovery protocol (similar to the logging system) is used to provide reliable storage.

Universal file servers

One of the best known file servers is the Cambridge File Server [Dion 1984; Mitchell and Dion 1982; Svobodova 1984], which falls into the category of universal file servers.[1] Access to the file server is controlled using a capability based approach. Clients can access arbitrary subranges of bytes, and concurrent access is supported to the extent that each file can have a single writer or multiple readers within it. Reliability of data is guaranteed by using a combination of shadow pages and an intentions log: all changes are written to shadow pages, and a record of which blocks should be allocated or deallocated is also made. Using this record, the file server can reach a consistent state after a crash. The state of the disk is recorded by redundantly storing an object map (where objects are stored), and a cylinder map (what objects are stored on each cylinder). If either of these maps gets corrupted, it can be reconstructed from the other.

One of the reasons the Cambridge File Server is a good example of a universal file server is that two different file systems have been implemented on it, including the Tripos Filing System [Richardson and Needham 1983], which provides a filing system for diskless machines connected on a Cambridge ring. The secondary storage is provided by a Cambridge File Server, with a dedicated machine known as the Tripos Filing Machine giving clients of the Tripos filing system access to the file server. The advantage of dedicating a computer for this task is that it relieves the load on client machines, as well as allowing efficient use of caching, and the implementation of a better security mechanism than the Cambridge File Server itself has. While theoretically the Tripos Filing Machine could be implemented on the same machine as the file server, with the type of hardware available to the implementors of the system, it was necessary to use a different computer. By fine tuning the various protocols for the type of service the file system was giving, significant gains in service could be made by using caching and by reducing the cost of synchronization.

The Arca File Server [Muir et al. 1985] and the Amoeba File Server

[1]Recall that universal file servers are designed so that more than one filing system can be built on top of them, and coexist in the same distributed system at the same time. However, they also provide a greater sophistication of services to their clients than simple file servers do.

[Mullender and Tanenbaum 1985] are other examples of file servers which support different filing systems.

Filing system/database management

The Xerox Distributed File System was intended as the basis for database research [Mitchell and Dion 1982]. Access control is imposed using an identity-based approach. Data and directories are stored reliably using stable storage. A high degree of concurrency (single writer and multiple readers within a transaction) with byte level locking on multiple files within the same transaction is supported. As with more sophisticated systems, it is the server which is responsible for managing the concurrency, relieving clients of this task.

Printer servers

The printer server in the Cambridge Distributed System only accepts one file at a time [Needham and Herbert 1982]. Spooling has to be provided by a client. So, in the same way the Tripos Filing Machine was built on top of the Cambridge File Server, a printing service can be placed on top of the printer server. This concept will be expanded in chapter 6.

The MacServe printer server for the AppleTalk system [Infosphere 1986], allows all nodes on the network to send it files for printing on Apple's Imagewriter. Spooling is allowed depending on how much disk space has been allocated for the spooling function of MacServe.

3.4 Distributed systems principles

File servers are a useful introduction to distributed systems because the principles used in their implementation are also applicable to other areas. Access control, concurrency control and reliability are equally important for many shared resources, and distributed systems in general [Notkin et al. 1987].

The coherence of a distributed system depends on what extent the distributed system was planned or grafted on afterwards, and on the level of uniformity in the system. Clearly, heterogeneity is unavoidable, but the level of heterogeneity in the system is important in its design. It must be accepted that at the hardware level, heterogeneity must be catered for. A more open question is whether higher-level functions like operating systems should be homogeneous or not.

There are two basic divisions of distributed systems, *distributed operating systems*, and *network operating systems* [Fortier 1986; Tanenbaum and van Renesse 1985; Tripathi et al. 1987]. A network operating system is a distributed system where each of the computers on the system uses its own operating system. In addition to the local operating system, there is additional hardware and/or software which allows resources to be shared.

A distributed operating system, on the other hand, is not built on to existing software or operating system. It is a system-wide operating system. Each computer on the network has the same operating system (the hardware can be different, but the operating system must be the same). To sum up the difference: in a network operating system, resources are locally owned and managed, and are available on request to other users; in a distributed operating system, all resources are globally owned, managed and shared.

The design principles of distributed systems are not much different from those used when designing an operating system. Besides those principles mentioned above, performance, flexibility and extensibilty should also be considered. The sharing of resources is complicated by the fact there is no shared memory between all the parts of the system, and no global system state available as there is in a centralized operating system.

Transparency

Transparency — the ability of a network or internetwork to hide machine boundaries from users — is one of the most important principles. A completely transparent system would allow a user to access the same application from a variety of machines. Not only would this mean that resources could be effectively shared, but this could lead to optimal use of resources; files could be stored on the best (however that is measured) place for them to be stored, the actual processing could be done on whichever processor(s) in the system would provide the best service. All of this could be hidden from the user.

Full transparency means common or compatible high-level protocols (HLPs). This is not an inherent problem with internetworks, as HLPs do not have to deal with hardware considerations. As a practical problem though, HLPs on different networks are incompatible, making transformation between the two almost impossible. This means that a common HLP must be available on all the networks connected to the internet. (Of course, these protocols could be available in addition to others.) The difficulty of protocol translation at the high-levels was pointed out in chapter 2.

It is worth mentioning here that the transparency of a system is an indicator of whether it is a distributed or network operating system. A completely transparent system is a distributed operating system, while a non-transparent system is a network operating system. It is also true that this distinction can become blurred [Tanenbaum and van Renesse 1985]. A filing system may be completely transparent while the process management is not transparent at all.

One problem of a practical nature is that different networks have different performances. If an internetwork is completely transparent, this would not be immediately obvious to a user. Anomalies in service, or degradation of service are possible. So, for example, a user might see that a laser printer is available on the system, and decide to print a long file on it. If the laser printer is on another network which is only accessible through a 1200 bps line, the performance the user experiences may not be acceptable, and this may also tie down other resources of the network. Although this is a problem, it need not be insurmountable, and the advantages of

transparency far outweigh this disadvantage.

Naming

Naming is one of the more important factors in transparency. The name of a resource that a user sees should not imply its address. The system should resolve an entity's address given its name. Other issues are how names are acquired, and whether there should be a global name space [Notkin et al. 1987]. Work on the standardization of naming is in progress [Goodwin and McDonnel 1986].

Reliability

Users of a distributed system must be able to rely on it. Experience [Lampson 1983] shows that reliability is difficult to put onto a system after it has been implemented. This means that reliability must be an essential part of the design of a successful system. However, reliability must be kept in context, as there is often a tension between reliability on the one hand, and speed and simplicity on the other. And if a system crash is not disastrous, then the maxim "one crash a week is a small price to pay for 20% better performance" [ibid.] may well be true. This point will be well illustrated in the implementation of a printer server which is discussed in subsequent chapters.

One way of handling errors is to have an exception server to deal with problems. Tanenbaum and van Renesse point out that many systems tend to favour performance over reliability [Tanenbaum and van Renesse 1985].

General

As distributed operating systems develop, global network management of all the processors in the system will become more important. Scheduling and protection against deadlock are more difficult in a distributed operating system than a centralized operating system because there is little global information. Distributed deadlock detection and prevention algorithms can be used [Peterson and Silberschatz 1985].

Remote procedure calls are a fairly common way of doing interprocess communication; asynchronous communication can make programming messy [Tanenbaum and van Renesse 1985]. Language and system support for the automatic generation of stubs for remote procedure calls is useful [Notkin et al. 1987].

Whether transport functions should be assumed reliable or not is an open question. It has been argued [Watson and Mamrak 1987] that as higher level functions need to cater for errors whether or not the transport functions are reliable, making transport protocols reliable is unnecessary and expensive. Related to this is whether the ISO-OSI model is useful in distributed systems. Transport functions will be treated in more depth later.

3.5 Survey of existing distributed systems

Illustrative examples of some of the distributed systems principles are the the Locus system, the V-system, and the Grapevine system.

The Locus system [Walker and Popek, undated; Walker et al. 1983] initially linked VAX 11/750s over an ethernet, but has also been implemented on a variety of computers connected by different networks, and an Internet Locus [Sheltzer and Popek 1986] also exists. The primary aims of Locus were to provide a fully transparent system with high reliability, flexible replication of storage, and high performance. The file system is a superset of the Unix system. Both the file system and processes are fully transparent: a user does not know where a file which is being used is stored, nor where processes are being run. This allows a great deal of flexibility, and can greatly improve system performance as process migration can drastically reduce data transfer across the network.

A lot of effort went into providing error recovery, reliability and availability. Highly replicated data structures, particularly file directories, are one of the chief factors promoting reliability, availability and efficiency. (The Eden system [Black 1985] also found that replicating structures like directories improved performance.)

Much thought had to go into synchronization and reconciliation mechanisms to keep different copies of the same data consistent. For example, if there is a physical break in the network, different replicas of a file may be on different partitions of the network. The question which a system has to address is whether users should have access to these replicas while the network is divided. Locus does allow access. Its rationale is that

most shared files are shared in read-only mode, and that other shared files are system files like directories for which automatic reconciliation algorithms exist.

Dynamic reconfiguration of the system is allowed, so high is the degree of transparency. Transparency was also important for the implementations of transactions on the Locus system [Weinstein et al. 1985].

The Locus system has successfully provided a transparent system, even in the internet case. Sensitivity to the slow speed of internetwork links was diminished by using three design principles. The use of a semantics based protocol to raise the level of messages sent across the network substantially reduced network traffic, while exploiting locality (as in any operating system), and doing appropriate job migration to minimize data movement were also helpful.

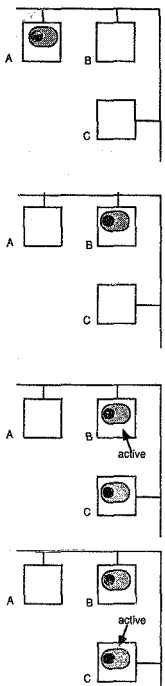
Stanford's V-system [Berglund 1986; Lantz and Nowicki 1984] is also a distributed system which allows transparent access to files, and transparent execution of programs. One of the underlying design principles of the V-system is that communication between different processes can be done using synchronous message passing, the calling process being suspended until it receives a response from the callee. This communication between processes is transparent to a user. The V-system is implemented on the Stanford University Network which is connected with a number of networks, including 10 Mbps ethernets, and has a number of different hosts including diskless workstations and timesharing computers. There are a number of dedicated servers on the network.

Like the Locus system, the V-system provides network-transparent execution of processes. V also allows processes to migrate while executing [Theimer et al. 1985]. This gives the advantage that a user can use spare processing power (the processors of workstations which are not being used heavily), but allow these spare processors to be reclaimed by their local applications or users. Migrating a process in mid-job may mean that a few megabytes of data must be transferred from one computer to another. To suspend a job for the time that this will take may be prohibitive. The solution used in the V-system is explained in figure 3.2. By adopting this approach, process migration becomes possible.

By allowing communication with, and control over groups of processes, the V kernel supports distributed server groups, distributed job control, and distributed parallel programs [Cheriton and Zwaenepoel 1985]. Processes may be grouped togther, and operations such as sending messages to the group or suspending the group may be performed easily and efficiently.

Some of the interesting results obtained from observing the behaviour of the V-system [Lantz et al. 1984] have been found in other systems, including Locus. For example, Internet Locus found it important to raise the level of the communication within the system. With the V-system, improving the high level protocols has far more effect on improving performance than tuning the underlying communication mechanism. Related to this, the study of the relative significance of different factors on performance [Lantz et al. 1985] showed that in order, the factors which performance is most sensitive to are.

- performance of the workstation,
- performance of the remote host if any,
- level of communication,
- choice and implementation of transport protocols, and lastly
- bandwidth.



1. A user at computer A decides to migrate a job to machine B. The data representing this process (including code) is transferred to B.

2. After a while someone else decides to use B. As a local process has priority, A's process is preempted, and the system decides to move it to C. First the data is moved to C.

3. Once the data has been moved, the system checks to see whether many of the pages transferred have been changed. If so, they are updated. This process is repeated until the number of pages that have been changed since the last copy is small.

4. If the number of pages changed is small, or the number of changes is not decreasing on each iteration of step 3, the job is suspended on B, and all the remaining data transferred to C. The job now starts running on C.

Figure 3.2 Migration in the V-system.

Grapevine [Birrell et al. 1982] is a distributed system with the primary function of providing a message service in a large internet. Besides delivering messages, it also has facilities for naming, access control and authentication, and resource location. These are very general functions which are needed by most services in any sort of operating system. Users of Grapevine do use these other facilities for services other than an electronic mail system. The design goals of the system were to have a replicated, distributed and reliable system with decentralized administration and reasonable performance. The physical system comprises ethernets (linked to each other directly by a gateway, or via telephone lines) which network personal computers and other dedicated servers. This is a large system supporting thousands of users.

One of the simplifications which Grapevine makes means that the system adopts a looser definition of consistency than does Locus, for example. In the Locus system, two users cannot modify different replicas of the same data except under unusual conditions (a physical partition in the network). However, for Grapevine's applications, it is acceptable for different users to access different versions of the same data. For example, if a user changes the location of its mailbox sites, not all Grapevine servers may be aware of the change immediately. If a server knowing the user's old address sends a message to it, then the message will be forwarded to the user at its new location, and the server's information will be updated. Thus, Grapevine takes the attitude that inconsistencies between replicated copies of the same data are acceptable because the copies will converge quite quickly, and that inconsistencies can be recovered from quite cheaply, and without any long-term ill-effect.

3.6 Analysis

Although a relatively new field, there are a number of sophisticated systems available. Reliable and efficient file servers with a wide variation of sophistication of service exist. Although their design can be complex, many of the lessons and principles of operating systems can be applied in the design of file servers — as with any distributed systems. Implementing reliability and concurrency properly are the areas which cause the most problems.

The range of sophistication shows the differences in philosophy of designers and the needs of users. Systems like the Simple File Server are examples of systems which allow sharing of resources through loose coupling. One particular service is shared; for the other services, the individual computers are responsible. The emphasis is on *cheap* sharing of resources.

On the other extreme, systems like the V-system and the Locus system are comparable in sophistication to an operating system on a large computer. It is important to note the advance of these systems over earlier systems, particularly as far as transparency is concerned. The Newcastle Connection [Brownbridge et al. 1982] is also a distributed UNIX-like system. However, the machine boundaries are not hidden. The name space is hierarchically organized, with the each component's file system appearing to the users as a directory of the global system. Furthermore, a user must negotiate rights with the administrator of each sub-system. Contrast this to the Locus system where the name space and access control are globally-managed. Directories are logically-organized — each directory can contain files which are on different machines.

Another interesting result has been the successful use of remote procedure calls or synchronous message passing in Locus and the V-system. With the appropriate system environment, remote procedure calls make development easier. The developers of the Eden system found in earlier versions of their system, that without proper support remote procedure calls were not appropriate [Black 1985]. The implementation discussed in the next two chapters shows the need for language and system support.

One of the more promising findings so far is that the performance of distributed systems can easily be improved by better hardware — doubling of the speed of the processor may double the performance. The apparent performance of a workstation is dependent on other things. Larger primary and secondary memory will allow more effective use of strategies like caching and buffering and thereby also improve the performance of the workstation [Lampson 1983].

As the cost/performance ratio of processors drops, and disk space becomes cheaper, many of the constraints which applied to some of the early

systems will no longer apply. Replication of data improves both availability, performance and reliability although significant effort is required to ensure that the different copies of the data stay within certain limits of consistency.

Expansion and proliferation of network services will emphasize the need for transparency. Dynamic reconfiguration of networks and services on them is needed both to cope with growth and moving, and to allow greater availability (if one file server crashes, users should be able to access their data from other file servers).

Printer services have not received that much attention [Tanenbaum and van Renesse 1985]. In the implementation discussed in the following chapters, the importance of printer protocols will be seen. The spooler/device controller model will be discussed in chapter 6, using the experience gained in the implementation, and an extension to the model will be proposed.

The next two chapters examine a case study of a small distributed system. This is a printer server which acts as a bridge between two different computers. After that, distributed systems will be examined in the context of resource sharing and communication generally.

4. A case study—MAT

This dissertation has surveyed both the theoretical and practical aspects of resource sharing and distributed systems. In this chapter, the design and implementation of a printer server are presented and discussed. This is an experimental system which highlights issues and shortcomings of distributed systems. In the next chapter, an assessment of the server is made both as a system and as an example of a distributed system. Some of the aspects of distributed systems which these two chapters examine are performance, concurrency, the need for language and system support, design of printer protocols and requirements for printer servers.

4.1 Problem description

In an environment with relatively varied equipment, and a fairly small number of instances of each type of equipment, expensive resources such as a printer can be underutilized. Where there are several networks in an organization, an expensive printer on one network should be available from the others to make full use of the machine. A good example of this situation is shown in the Department of Computer Science at the University of the Witwatersrand, which has:

- An AppleTalk network, with Apple Macintoshes and an Apple LaserWriter attached to it
- An IBM PC-network, with four IBM ATs and one IBM XT attached to it. An IBM Proprinter is attached to one of the IBM machines
- A Macbridge—a card inserted into a slot on an IBM XT or AT. This makes one of the machines on the IBM PC-net a node on the AppleTalk network. Using appropriate software, this machine can print files directly to the Apple LaserWriter, and files can be transferred between this IBM machine, and any of the Apple Macintoshes. Note that none of the other machines on the IBM PC-net can use any of the facilities of Macbridge.

Two possibili[illegible]ear almost immediately. Ideally, a printer on either network should be available from anywhere on the combined networks.

Secondly, a printer spooler would be useful. Before discussing a proposed solution, a technical introduction to the equipment is given to set the context.

PC-network

The IBM PC-network [IBM 1984] is a broadband bus local-area network running at 2 Mbps. Although the network is logically a bus, its physical configuration is that of a star network. A network adapter card is installed in each machine connected to the network. This adapter card implements protocols up to the session layer. Session layer facilities are offered to the user application by NETBIOS. NETBIOS is the interface between the computer and the network card. More detail on NETBIOS can be found in appendix A; the basic facilities it offers are session [illegible] name support, datagram support and monitoring functions.

Briefly, a program executes a NETBIOS command by [illegible]zing a block in memory — called a network control block — with appropriate data, depending on the function to be invoked. The program then raises an interrupt which has the effect of passing the address of this block to the network card, and transferring control to the card. The card then performs the appropriate function. There are three modes of operation:

- the NETBIOS card can block the calling program until the function is completed,
- the NETBIOS card can perform some initial checks before returning control back to the calling program, after which it processes the command asynchronously. The calling program must then check an appropriate variable to find out when the operation has been performed, and
- the NETBIOS card can perform some initial checks before returning control back to the calling program, after which it processes the command asynchronously. When the NETBIOS command completes, the NETBIOS card raises an interrupt which transfers control to a routine specified in the network control block by the calling program.

The advantage of using PC-network is that NETBIOS removes many of the detail problems which would otherwise have to be solved, as all protocols up to the session layer have already been implemented. Even though the IBM PC-network has not been very successful [Data Communications

1986], IBM are using the same NETBIOS for its token ring network. The University of Pretoria are implementing a NETBIOS interface for their Novell network. Therefore, the work described here does not rely on the physical characteristics of the network.

AppleTalk network

The AppleTalk network [Apple undated] is a baseband local-area network, running at approximately 230Kbps. A hierarchy of protocols has been implemented by Apple, broadly implemented within the ISO-OSI framework. The protocols of interest are the Link Access Protocol (ALAP) which is implemented at the Data-link layer, the Datagram Delivery protocol (DDP) at the network layer, the AppleTalk Transaction Protocol (ATP) at the transport layer, the Name Binding Protocol (NBP) at the session layer, and the Printer Access Protocol (PAP) at the presentation layer. Appendix B gives a more detailed account of AppleTalk protocols. Network commands are issued on the Macintoshes using standard operating system calls.

Apple LaserWriter

The Apple LaserWriter [Apple 1984] is an intelligent printer which is an independent node on the AppleTalk network. The LaserWriter used in this implementation has a Motorola 68000 chip, and is controlled using the Printer Access Protocol [Sidhu and Oppenheimer 1985]. The LaserWriter is a powerful computer in its own right: it has 0.5Mbytes of ROM and 1.5Mbytes of RAM; several fonts are built in to the ROM as well as an interpreter for the PostScript language. The Printer Access Protocols make the LaserWriter a server on the network. PostScript files [Adobe 1985a, Adobe 1985b] are accepted from clients and then executed. The PostScript language is a high-level device-independent language for page description. It is one of the purposes of the implementation to evaluate Postscript as an application layer protocol. Appendix C gives an introduction to PostScript. The LaserWriter itself is a monochrome printer giving a resolution of 300 dots per inch.

Macbridge

Macbridge [Tangent 1987] has been developed by Tangent Technologies, and consists of a card which is placed into an IBM compatible machine,

and software. This card is connected to the AppleTalk network. Its manner of operation is very similar to that of the NETBIOS: a program using the Macbridge sets up a block in memory with appropriate data, and then raises an interrupt which passes the address of the block to the AppleTalk card. The card then executes the appropriate command. There are two modes of operation: the card can block the calling program, or can allow the calling program to continue — a part of the block can be examined to check when the command completes. In addition, the program can pass the card the address of a routine to be executed once the card has finished processing the command. This is done by the card raising an interrupt when it completes. An interrupt handler of the card then calls the program defined procedure. A more detailed account of Macbridge can be found in Appendix B.

Throughout the next two chapters, *card* or *adapter card* is used as a generic description for the NETBIOS and Macbridge cards.

4.2 The *MAT* — Requirements

The possibilities can be fulfilled by a printer server which acts as a bridge between the two networks. The printer server is called the MAT. (This stands for *M*ain *AT*, or *MA*in XT.) This is an AT or XT which is a node on the PC-net, and contains the link (Macbridge card) to the AppleTalk network. The server consists of two parts, a simple file server which accepts files from clients of either network, and saves them, and a printer controller which is responsible for any protocol translation and control of the actual printing device. Figure 4.1 shows the configuration of the system.

The service offered to the clients allows them to print files to any printer on the network; in the actual configuration, there are only two printers on the network, the IBM Proprinter and the LaserWriter. The Proprinter is attached directly to the MAT[1].

[1]This does not limit the generality of the work. The device controller part of the printer

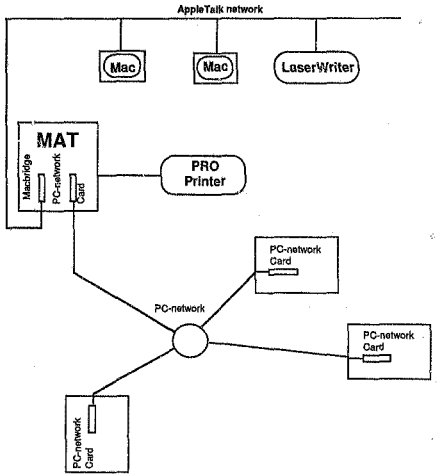


Figure 4.1 — Configuration of the MAT

While reliability is a desirable feature for the MAT, bearing in mind the advice on reliability stated in the previous chapter [Lampson 1983], the system will not have to be completely reliable. For the server to be reliable, it would have the following properties:

---

server must be intelligent. In the case of the LaserWriter, the printer can control itself. In the case of the Proprinter, the MAT itself acts as the device controller — another node on the PC-network could have been set up to do this.

(a) if a file is sent to it, the file must be safely stored on disk,

(b) once a file is saved on disk, it must be printed (even if the MAT crashes, particularly if it crashes due to some external factor),

(c) if a file is printed, it should be deleted from the disk, or if the file cannot be printed the user should be informed and the file deleted from the disk and

(d) a file should only be printed once. (This restriction does not reduce the generality of the MAT as will be seen.)

The extent of reliability is determined by what is possible in the environment, and a trade-off between reliability on the one hand, and simplicity of design and performance on the other. Of course, if a system is very unreliable then its performance *ipso facto* degrades. It is unlikely that a server which fails to perform its function regularly will satisfy the requirements of a user.

It is acceptable not to make a requirement that the server be completely reliable because printers, at least in the environment described, do not deal with irreplaceable data: users can just reprint files if they do not appear. Note that if confidential documents were being printed it would be undesirable for files to be 'lost' and then reappear a few hours later. There are also practical advantages of not having a completely reliable system which make this decision expedient. These will be seen later.

It is a requirement that the MAT be robust. The MAT should be safe against client crashes or misbehaviour. The MAT's crashing (for example, if the power fails) should not cause incorrect behaviour when it is restarted.

A final requirement is that the MAT should be useful and efficient. Client code's size and data requirement must be relatively small. To define 'usefulness' in objective terms is difficult. At the very least, considering the MAT acts as a spooler, using the MAT to print a file on any printer should require significantly less processing time on the client than printing a file to a local printer. The MAT should be designed so that when it is not performing useful work the processor should be available for any other tasks running on the machine.

The MAT is not an end in itself. It is also designed to explore more general issues of distributed systems. The requirements stated here will allow the discussion of the design and implementation to highlight issues of performance, concurrency, the need for language and system support, and design of printer protocols. It will also be possible to state requirements for printer servers with greater clarity and precision after the design and implementation of this server have been discussed and assessed.

4.3 General design considerations

Flexibility

The intention that the research described here could be generalized, rather than an actual need to make extensions later, guided the decision to make the MAT flexible and extendible. The extent to which this intention was achieved will be examined in the next chapter.

Multitasking

Allowing multitasking is important for a number of different reasons. A system like the MAT has different tasks to deal with concurrently — there are multiple threads of control within the program. Recognizing and supporting multitasking simplifies design and implementation of distributed systems [Black 1985].

From a performance point of view, multitasking is vital: if it is not done, a situation could arise where a processor is idle while servicing one task, and another task which needs service is not getting service. Such a situation would also be undesirable from a client's perspective. It is undesirable for a client which needs a few seconds of service to wait two minutes for the MAT to perform some other task.

The above reasons guided the decision to exploit concurrency wherever possible (by using interrupts), even though the environment of the MAT does not support multitasking.

File server decisions

The previous chapter discussed file servers in detail, and pointed out some of the sophisticated facilities that servers can provide, and the

correspondingly difficult design decision which have to be made (also see [Svobodova 1984]). For the MAT, the decisions are quite easy, as the file server part of the MAT has relatively few and unsophisticated tasks.

Conceptually, to the MAT and its clients, a file consists of a request block (see Figure 4.2) and a packed array of ASCII characters. Clients are responsible for ensuring that the request block is presented to the MAT in the correct format. How and where the MAT stores the file is irrelevant to the clients, and vice-versa. This conforms to the OSI File Service discussed in chapter 2: the differences between local representations is absorbed into a local mapping function.

```
fmtblk  = packed array[1..8] of byte;

tofiles     = ^print_file;

print_file = packed record
   fname : string[29];              {name of file}
   from_fmt, to_fmt : byte;         {source, dest file formats}
   next  : tofiles;
   lngth : integer;                 {length of file}
   ptr   : integer;                 {pointer to the data}
   fmt   : fmtblk;                  {format information}
   prtn  : AddrBlk;                 {name of printer}
   filler: byte;
   case net : net_type of
      AT_net: (                     {variant for AppleTalk}
               addr  : AddrBlk;
               sparea: packed array[1..12] of byte;);
      PC_net: (origin : names;
               );                   {variant for PC-net}
end;
```

Figure 4.2 — Definition of request block

There is limited concurrent access by different clients to the same data structures. The limited present needs do not require the support of concurrent access. Possible concurrent access in the future would be to allow a client to cancel a job before it finishes printing.

Remote storage of entire files is the file server's only real task. Thus, it is reasonable that the unit of data access is the file.

How reliability and robustness are implemented will be discussed in the next section.

4.4 Design and implementation

The MAT is written in Turbo Pascal under DOS 3.10. There is no provision for multitasking with DOS, so the machine will run as a dedicated server. However, the limitations of DOS (something which other have found [Morris et al. 1986]), forced this, and not the inherent needs of the MAT or printer servers in general. One of the purposes of the implementation will be to examine whether it is necessary for the MAT to be a dedicated machine. Personal experience with other non-dedicated servers (viz. an Apple Macintosh running as a MacServe [Infosphere 1986] server) has shown that there can be noticeable degradation in service experienced by a user of a server machine, and by users of clients of the server machine when the server machine is not dedicated. There are a number of factors which could lead to this, and these factors together with the experiences of the MAT will be presented in the next chapter.

The advantage of using Turbo Pascal is that most (99%) of the program is written in a high level language. Turbo Pascal allows machine code to be embedded within the program, or for assembler routines to be called as subroutines. There are restrictions on the size of the Turbo program; however, this is not a problem in the case of the MAT. A greater disadvantage is that there are no mutual exclusion routines (e.g. *wait*) to allow separate threads of control to be present, and that modularity is limited (no facilities for separate compilation).

Logically, the code in figure 4.3 represents the design of the MAT code. Implementation limitations necessitated changes to this outline.

The MAT's code can be divided into three sections: initialization code, the main body, and clean-up code.

The initialization code is responsible for initializing the NETBIOS and AppleTalk cards, and registering the name of the MAT on both networks. The clean-up code is responsible for undoing this, and ensuring that any pending network commands are canceled.

```
coroutine AppleTalk;
 begin
 repeat forever
      wait for request;
      if can service request then
         transfer file into memory
         place file on linked list in memory for saving
      else
        refuse request
 end
```

```
coroutine PC-net
 begin
   repeat forever
      wait for request;
      if can service request then
         transfer file into memory
         place file on linked list in memory for saving
      else
        refuse request
 end
```

```
coroutine main_loop
 begin
      repeat forever
            if files in linked list in memory then
                  for all files in list
                        perform protocol conversion
                        save file on disk
                        update list of saved files on disk
            print first file in linked list on disk
            delete file printed
 end
```

Figure 4.3 — Outline of MAT Code

The rest of the code is designed to be interrupt-driven wherever possible. By this means, the MAT need only be doing work when requested to by the network cards, and thereby partial concurrency is implemented.

The main event loop of the program has two main tasks. It checks to see whether the MAT has had any files sent to it, and if it has, saves these files to disk, if necessary doing protocol conversion. Its second task is to print any files which have been saved. While it is doing this, it may be interrupted, and the MAT's interrupt handlers[2] will perform service for a

[2] In fact, when one of the network cards completes a command, and issues an interrupt, an interrupt handler defined by the network card gets called. This routine then calls the MAT's routine. There are minor differences in the way the two network cards manage

client on one of the networks.

When a request from a client arrives, an interrupt is generated — this may occur while the computer is printing another file, or performing protocol conversion. The interrupt is serviced to examine the request. If the file is smaller than the buffer available on the MAT, it is received from the client immediately. The contents of the file is kept in memory, and the MAT returns to its previous task. Once that job is finished, the file is transferred from memory to disk.

If the file is bigger than the buffer available for it on the MAT, the client is told that the MAT is busy. It must then retransmit its request.

Performing service for a PC-net client

Communication with MAT clients on the PC-net is done using the PC-network card via NETBIOS. The MAT establishes its name on the card. Two interrupt handlers are then used to handle requests from PC-clients. Before they can send requests to the MAT, clients must find its identity and address. To do this, a packet is broadcast on the network, asking for the MAT's address. The MAT has issued an asynchronous command which can receive this type of broadcast datagram. When the card receives the datagram, the command completes, and control is transferred to the MAT-defined interrupt handler which sends back a response packet to the client. Control is then returned to the place the MAT was executing when it was interrupted. There need be no further communication between the MAT and the client: the client could have been a utility searching for devices on the network.

Once a client has the server's address, if it wishes to print a file, it sends a datagram to the MAT with details of the service wanted (e.g. file name, length, printer wanted) — this is the request block shown in Figure 4.2. Again, the MAT has a pending command ready to receive this datagram. It completes when the datagram arrives, invoking another interrupt handler. This examines the request to see whether it can provide the service requested. If it cannot, the client is so informed, and the communication is broken off. A typical reason for this happening is that the MAT's buffer is

---

this.

client on one of the networks.

When a request from a client arrives, an interrupt is generated — this may occur while the computer is printing another file, or performing protocol conversion. The interrupt is serviced to examine the request. If the file is smaller than the buffer available on the MAT, it is received from the client immediately. The contents of the file is kept in memory, and the MAT returns to its previous task. Once that job is finished, the file is transferred from memory to disk.

If the file is bigger than the buffer available for it on the MAT, the client is told that the MAT is busy. It must then retransmit its request.

Performing service for a PC-net client

Communication with MAT clients on the PC-net is done using the PC-network card via NETBIOS. The MAT establishes its name on the card. Two interrupt handlers are then used to handle requests from PC-clients. Before they can send requests to the MAT, clients must find its identity and address. To do this, a packet is broadcast on the network, asking for the MAT's address. The MAT has issued an asynchronous command which can receive this type of broadcast datagram. When the card receives the datagram, the command completes, and control is transferred to the MAT-defined interrupt handler which sends back a response packet to the client. Control is then returned to the place the MAT was executing when it was interrupted. There need be no further communication between the MAT and the client: the client could have been a utility searching for devices on the network.

Once a client has the server's address, if it wishes to print a file, it sends a datagram to the MAT with details of the service wanted (e.g. file name, length, printer wanted) — this is the request block shown in Figure 4.2. Again, the MAT has a pending command ready to receive this datagram. It completes when the datagram arrives, invoking another interrupt handler. This examines the request to see whether it can provide the service requested. If it cannot, the client is so informed, and the communication is broken off. A typical reason for this happening is that the MAT's buffer is

---

this.

full.

There are two approaches to fulfilling a client's requests in these circumstances. The one is to keep a record of the request, and when the MAT's resources are available, meet the request. The MAT — for reasons of simplicity — just informs the client why the request cannot be met. The client could then decide for itself whether transmitting the request again in a few seconds might be fruitful. Note that this could happen at the client code layer, and the user need only notice a few seconds of delay before the service requested is provided.

On the other hand, if the MAT can honour the request, a session is immediately opened, and the file is transferred across the network directly into memory. The request is placed in a linked list of files which have been received. This linked list is ordered on a first come-first serve basis, but a priority scheme could easily be implemented here. The file cannot be saved immediately because all of this is done within an interrupt handler, and so no I/O is allowed[3]. The interrupt handler then completes by closing the session, and issuing another asynchronous command to receive future request datagrams.

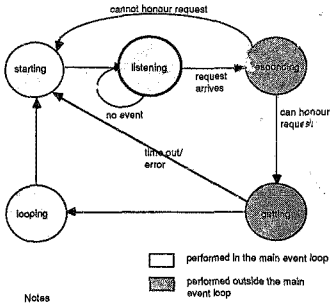
Performing service on the AppleTalk

The major difference between the routines to handle the clients on the AppleTalk, and the routines already discussed is that interrupt handlers can only be used in a limited way. As already noted, the MAT's 'interrupt handler' is not called directly when the AppleTalk card issues an interrupt, but is called from an interrupt handler defined by the AppleTalk card. This interrupt handler uses its own stack which it places adjacent to the heap. Thus, the MAT's 'interrupt handler' routine cannot call any procedure because then the stack would run into the heap. With the limits of the development environment, this meant that interrupt driven mechanism could not be used to the same extent as with the NETBIOS card. This and other problems of the development environment are discussed in greater detail in chapter 5.

The use of interrupts is simulated in the following way. A state variable

[3] A limitation of DOS.

keeps track of what state the MAT is in in relation to receiving a file on the AppleTalk network. On each iteration of the main event loop, the condition of the state variable is tested, and appropriate actions taken. Figure 4.4 illustrates how it is used. States inside the main event loop are checked there explicitly because interrupts cannot be used to test the state implicitly. (One of the implications of this strategy is that the MAT cannot service an AppleTalk request while it is performing another service such as printing another file.)



Notes

starting: prepares Macbridge card to receive request

listening: MAT polls for request.

responding: MAT checks to see whether it can meet request.

getting: the file is transferred in packets to the MAT. When the command to receive the last packet is issued, the MAT goes to the 'looping' state.

looping: MAT waits for the last packet to be transferred

Figure 4.4 — Servicing the AppleTalk clients

Rationale for decisions for both networks

The assumptions behind these decisions are:

(1) The usual cycle of operation is idle-receive-print-idle-receive-print...

(2) The server is able to deal with most requests that it receives while printing a file. It does not seem unreasonable to reserve up to 128K of memory on a 1M machine for this purpose. Most files will be smaller than this, and the MAT should be able to hold a couple of files while printing. In this implementation, the MAT reserves a 32K buffer for this purpose. Extending this to a larger size, within the limit of 64K which Turbo Pascal places on the data segment, is straightforward.

(3) To provide the best service on average, a client should not be made to wait for a long time. On average, most files will be small, and therefore can be placed in memory immediately.

(4) Interrupt service providers cannot do any I/O. This is a limitation of the BIOS (Basic Input Output System). Thus, interrupts must either defer performing service to a routine not within the scope of the interrupt handler, or not perform any I/O, keeping data in memory. There seems to be no easy way to get around this.

(5) All files will be handled on a FIFO basis. However, it should not be difficult to extend this to print files on a priority basis — priorities could either be explicit or implicit

(6) To simulate multitasking by providing a quantum of time for each task which must be performed is not desirable. In effect, this would be performing functions (scheduling and dispatching) which should be provided by the operating system. Unless it is done properly (including programming at a very low level), there is little to suggest that this would lead to good performance. It would lead to messy programming. This consideration outweighs the fact that in the present solution, the CPU may be idle some of the time when work needs to be done.

(7) To reduce network contention, and make better use of [illegible] data transfers should be done as quickly as possible. This means [illegible] no

Rationale for decisions for both networks

The assumptions behind these decisions are:

(1) The usual cycle of operation is idle-receive-print-idle-receive-print...

(2) The server is able to deal with most requests that it receives while printing a file. It does not seem unreasonable to reserve up to 128K of memory on a 1M machine for this purpose. Most files will be smaller than this, and the MAT should be able to hold a couple of files while printing. In this implementation, the MAT reserves a 32K buffer for this purpose. Extending this to a larger size, within the limit of 64K which Turbo Pascal places on the data segment, is straightforward.

(3) To provide the best service on average, a client should not be made to wait for a long time. On average, most files will be small, and therefore can be placed in memory immediately.

(4) Interrupt service providers cannot do any I/O. This is a limitation of the BIOS (Basic Input Output System). Thus, interrupts must either defer performing service to a routine not within the scope of the interrupt handler, or not perform any I/O, keeping data in memory. There seems to be no easy way to get around this.

(5) All files will be handled on a FIFO basis. However, it should not be difficult to extend this to print files on a priority basis — priorities could either be explicit or implicit

(6) To simulate multitasking by providing a quantum of time for each task which must be performed is not desirable. In effect, this would be performing functions (scheduling and dispatching) which should be provided by the operating system. Unless it is done properly (including programming at a very low level), there is little to suggest that this would lead to good performance. It would lead to messy programming. This consideration outweighs the fact that in the present solution, the CPU may be idle some of the time when work needs to be done.

(7) To reduce network contention, and make better use of CPU time, data transfers should be done as quickly as possible. This means that no

protocol conversion should be done on the fly. The procedure must be: do any necessary conversion on the whole file in the client and then do data transfer, or do the data transfer and then perform protocol conversion on whole file on the server.

(8) The system should be flexible so that either of the above approaches can be done. The file must be accompanied by state information like the format the file is in, and what printer it is going into. This means that files can either be stored in the universal format (Postscript), or the format of the printer it is going to. This will mean that no unnecessary translation needs be done.

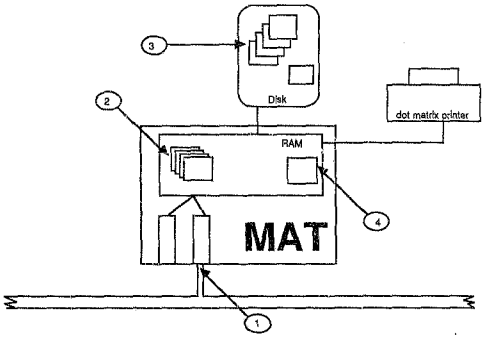
To summarize, figure 4.5 shows a typical transaction between a client on the PC-network and the MAT.

Protocol conversion

The MAT is designed to store files in the form they will be printed, or in PostScript format. A file undergoes protocol conversion if it is received in the format in which it will be printed.

An important decision to be made is whether protocol conversion should take place on the MAT or on the client. The advantage of it happening on the MAT is that the client is relieved of this, reducing the complexity of the client and the resources needed to use the MAT, and increasing flexibility. It does mean that the MAT will become more complex, but this is a small price to pay as the resources and complexity only need to be dealt with once — on the MAT — instead of every possible client. A more serious disadvantage is that the protocol conversion requires relatively more processing than other parts of the MAT's operation[4]. Although, with the use of interrupts, other types of processing can be overlapped with the protocol conversion, the MAT's throughput will degrade at heavy — or even modest — loads; the MAT's ability to operate as a non-dedicated machine will also be impacted.

[4]See chapter 5, figure 5.1



1. A request arrives on the network, generating an interrupt. The current job on the MAT is blocked until the interrupt has been serviced. If the MAT cannot service the request, a message refusing service is sent.

2. If the MAT can honour the request, a session is opened with the client, and the file is transferred into memory. All this is done within the interrupt. When the transfer is complete, the interrupt handler is exited.

3. When the MAT completes printing the current job, any files in memory are saved on disk. A file containing a list of all files currently on the disk is also saved.

4. The files on disk are printed in FIFO order. Once the file has been printed, it is erased from disk.

A typical interaction with a client on the PC-network.

Figure 4.5

The two approaches are not mutually exclusive. The overhead of just having the extra-code on the MAT is unlikely to be significant. Thus,

clients which can afford to have the appropriate protocol converters should be encouraged to do so; if they do not wish to use them (either because of their need to improve performance or because of shortage of disk space), the MAT can provide the facility, which if not used often will not degrade performance.

The current version of the MAT only does protocol conversion from an ASCII file with carriage return and line feed characters to PostScript. Integrating new protocol converters is easy. Stubs have been written, and the extensions are easily integrated.

The ASCII to PostScript converter produces a PostScript program. The converter keeps the text lines as the user enters them. No justification is done. The algorithm is:

```
Move to the top of the page
while not end of file
   print the next line
   move to the next line
   if end of page, print page & go to top of new page
end;
```

Chapter 5 presents alternative strategies for protocol conversion. These decisions influence where processing takes place (on the LaserWriter or on the MAT), and flexibility available to a client.

The above discussion refers to application layer protocol conversion. However, there was also a presentation layer protocol conversion problem encountered. The different byte-ordering approaches used by the different processors causes the problem. The MC68000 stores integers (two bytes) in the high byte-low byte format. The Intel 8086 family stores integers (two bytes) in the low byte-high byte format. The request block which is sent to the MAT from both the AppleTalk clients and the PC-net clients includes an integer field stating the length of the file to be printed.

The solution adopted uses the MAT's definition of the request block as the definitive one: the virtual file description. Thus, clients must send the MAT the request block in the byte order which the MAT expects. This was fairly straightforward to implement in the case of the MAT. Any byte swapping is done on the clients, and the MAT is not even aware of the problem. The solution seems satisfactory and could be applied to new clients—for this

reason, the issue is not discussed again.

Reliability and Robustness

There are four factors for the MAT to be reliable. (These were first mentioned in section 4.2).

(a) *If a file is sent to it, the file must be safely stored on disk:* A client cannot be guaranteed that once a file has been transferred to the MAT that either the file will be printed, or the client informed that it could not print the file. No provision is made for reliability before the file is saved. If the MAT crashes before the file is saved, all record of the file is lost. The main reason for this is that I/O cannot be performed in an interrupt handler. No provision is made for problems like the disk being damaged, or a file being corrupted.

(b) *Once a file is saved on disk, it must be printed:* Once the file has been saved, the MAT will print it, even if the MAT crashes. A list of files to be printed is kept on disk. When starting (or restarting), the MAT checks this file to see if there are any files to be printed.

In due course, the file will be printed if the printing protocols work and the printers are on-line. Clients are *not* informed if the file fails to print.

(c) *If a file is printed, it should be deleted from the disk:* A file is deleted when it has completed printing. If, for some reason, the MAT always crashes once the file has printed, but before it is deleted from the disk, on restarting the MAT will start printing the file again.

(d) *A file should only be printed once:* If the event described in (c) occurs, then the file will be printed more than once. This is what was meant by the requirements of (b), (c) and (d) being in conflict: a file may be printed more than once — violating (c) and (d) — to ensure that it has printed.

There is only limited feedback to the client. When the client first requests service, it is informed whether the service can be met or not. If it can be, it transfers the file across to the MAT. If this transfer is successful, this is the last communication between the client and the MAT. The main

justification of this is ease of implementation. In addition, the MAT cannot be sure that the user who sent the file will still be there after this time.

Robustness is accomplished by using time-out values, and instead of using synchronous commands, using asynchronous commands with a loop immediately after the command waiting for it to complete. This allows increased flexibility.

Another issue is ensuring that the MAT and its clients are synchronized. For example, a client sending the MAT a datagram when the MAT had not issued a command to receive a datagram could lead to problems — the client could mistake this for the situation where the MAT was dead. A typical situation where this happens is where one side of the communicating parties first requests some service from a user, and then must issue another command to receive the request, as in

MAT

```
issue send datagram asking for data
.
.
.
issue receive datagram command for data
```

Client

```
issue receive datagram to wait for request
while request is pending do (nothing);
send datagram with data
```

The correctness of the above depends on the implementation of the receive and send datagram commands. For example, a send datagram command may fail if a receive datagram command has not been issued by the other communicating party, or the send datagram may be queued at the receiving end until a receive command is issued.

The approach the MAT takes is to issue a receive datagram command for the data requested *before* sending the datagram requesting the data, as in:-

justification of this is ease of implementation. In addition, the MAT cannot be sure that the user who sent the file will still be there after this time.

Robustness is accomplished by using time-out values, and instead of using synchronous commands, using asynchronous commands with a loop immediately after the command waiting for it to complete. This allows increased flexibility.

Another issue is ensuring that the MAT and its clients are synchronized. For example, a client sending the MAT a datagram when the MAT had not issued a command to receive a datagram could lead to problems — the client could mistake this for the situation where the MAT was dead. A typical situation where this happens is where one side of the communicating parties first requests some service from a user, and then must issue another command to receive the request, as in

MAT

```
issue send datagram asking for data
.
.
.
issue receive datagram command for data
```

Client

```
issue receive datagram to wait for request
while request is pending do {nothing};
send datagram with data
```

The correctness of the above depends on the implementation of the receive and send datagram commands. For example, a send datagram command may fail if a receive datagram command has not been issued by the other communicating party, or the send datagram may be queued at the receiving end until a receive command is issued.

The approach the MAT takes is to issue a receive datagram command for the data requested *before* sending the datagram requesting the data, as in:-

MAT

```
issue receive datagram command for data
issue send datagram asking for the data
.
.
.
```

Another mechanism used to avoid timing problems is to prepare all communication beforehand wherever possible. The clients of the MAT, for example, read the file that they want to send to the MAT from disk into memory before asking for service. This also has the advantage that the total time spent by the MAT communicating with a client is shortened. This is even more important if the client is doing protocol conversion.

4.5 Summary

To summarize, the MAT is a node on two networks, the AppleTalk network and the IBM PC-net. It has the capability to print files sent to it to two printers, a dot-matrix printer which is attached directly to it (a printer on the PC-network), and to an Apple LaserWriter which it communicates with via the AppleTalk network.

This chapter describes the hardware components of the system, and its requirements. It then looks at issues which arose in the design and implementation phase. The main focus of the chapter is on the discussion of the MAT as a printer server. However a number of problems arose in the design and implementation. Several of these problems reflect problems or issues in distributed systems or printer servers more generally, including

- problems with the language and system
- attempts to improve performance: protocol conversion and exploiting concurrency
- what level of reliability and robustness to aim for and how to achieve it.
- the rôle of the printer server

5. Assessment

This chapter assesses the MAT in three ways. It assesses the MAT as a printer server (an end in itself), as an example of a distributed system (as a tool to explore distributed systems), and thirdly in the light of the above, it examines what changes and extensions can be made to the MAT both as a printer server and as a tool for examining distributed systems.

5.1 Assessment of design decisions

Development Environment

The choice of hardware, operating system and language must be assessed. As far as the hardware is concerned, the AT or XT with the Macbridge card had to play a central rôle. The MAT could have been an Apple Macintosh, with the XT or AT acting as a bridge, merely transferring files between the two networks, but the more logical choice was for the AT or XT which was the bridge to play this rôle.

The choice of operating system and language for the MAT was an important one. It was a decision which had major ramifications for the design, implementation, reliability and performance of the system. Other research in distributed systems has also found the system environment to be important in this regard [Brown 1985; Fry 1987; Morris et al. 1986].

There was not much freedom when choosing DOS as the operating system with which to implement the system — it was the only operating system available in the department at the time. Immediately obvious was DOS's lack of multiprocessing facilities. This means that without considerable effort, including writing low level routines to interact with DOS and overcome some of its problems, the MAT would have to be a dedicated server. This makes direct measurement of the effect of the operation of the MAT on other tasks impossible. Not being able to do I/O within an interrupt handler was also a major limitation.

After initial experimentation within assembler, Turbo Pascal was chosen for the reasons explained in chapter 4. There were serious drawbacks to

using Turbo Pascal besides the ones mentioned earlier. The use of *inline* machine code is limited when Turbo Pascal runs under DOS.

Individually, all these problems may not seem so serious. Together with some of the quirks of the network card, they make a formidable obstacle to clean programming, simple design and, in some cases, implementation of desired features.

One aspect of this is mentioned in detail as an example to show that some of the limitations of the MAT are caused by a poor environment, and not by the design. Nor are the limitations inherent limitations of printer servers.

When a program defined interrupt occurs, and control gets transferred to a program-defined interrupt handler, the DS register no longer necessarily points to the global variables. This can be overcome, if slightly messily. The interrupt handler can then call other routines. Unfortunately, this does not work with interrupts caused by the AppleTalk card, because of the way the AppleTalk card manages its stack[1]. Given the other limitations, this makes a design decision to use interrupts as much as possible difficult to implement.

In the light of this, if the MAT were to be reimplemented, languages which support synchronization or concurrency and have better support for low-level access such as Modula-2, Ada or C should be considered. The use of another operating system (like UNIX) would also be an improvement.

Limitation of file size

The MAT limited the size of files to be printed to 16K. This is an unsatisfactory limit as many documents exceed this size. The limit was placed at an early stage of the program development for a technical reason for simplicity. However, allowing files to be longer would be a straightforward development. For this reason, the restriction on the size of the file does not impinge on the generality of the MAT.

[1]See chapter 4 for a detailed explanation

5.2 Synchronization and interprocess communication

There are two issues of concern here: the implementation of concurrency within a program, and communication between the MAT program and its clients. The mutual exclusion and synchronization problems are related, and so are discussed together.

The approach of using interrupt handlers to do much of the work of the MAT means that there is partial concurrency within the program. For example, if the MAT is saving a file from memory to disk while doing some protocol conversion, and a client requests service, an interrupt handler could bring the file into memory at a critical time when both the interrupt handler and the routine being interrupted are accessing the same data structures. Without proper synchronization primitives (lil.e *wait* and *signal*), supporting mutual exclusion, either the solution adopted will be incorrect, or a client may be refused service when the MAT is, in fact, able to provide service.

In assessing whether allowing this partial concurrency is worthwhile, it is necessary to remember that the greatest motivation for the use of the interrupt handlers was efficiency both in terms of throughput, and apparent execution time for users. The breakdown of execution time is presented in figures 5.1 and 5.2.

The time which the client experiences is the time for the file transfer— less than one second in the case of PC-net clients, and about 20s in the case of AppleTalk clients. From the user point of view, the decision to use interrupts to provide partial concurrency is well-justified. For clients on the PC-net, it would be extremely undesirable to have to wait for other jobs to complete before getting service, as the MAT spends at most 2.5% of the time taken to process a job on the file transfer. Even in the case of AppleTalk clients, the proportion of time spent (±20%) on the file transfer phase would well justify the interrupt-driven approach if this were possible. At present, a client might have to wait for a printing job to complete (which for an 11K file takes about 80s to do).

Printing to the LaserWriter[2]

Activity	PC-net client	AppleTalk client
File transfer to MAT	<0.8 *	19 *
Saving to disk	1.64	1.64
Protocol translation	5.6	5.6
MAT transfer to LaserWriter	85 *	85 *
Wait for acknowledgement	9 *	9 *
Total	101 *	119 *
LaserWriter busy for	95s*	95s*

Note: These figures represent average times printing of an 11K file to the Apple LaserWriter in Times-Roman 12pt. All figures in seconds.

Figure 5.1

Printing to the Proprinter

Activity	PC-net client	AppleTalk client
File transfer to MAT	<0.8 *	19 *
Saving to disk	1.64	1.64
MAT transfer to Proprinter	78 *	78 *
Total	80 *	98 *
Proprinter busy for	140s *	140s *

Note: These figures represent average times printing of an 11K file to the IBM Proprinter. All figures in seconds.

Figure 5.2

Performance is likely to be improved in other ways by the approach. By servicing jobs immediately a request for service is made, the MAT is likely to reduce the average time spent servicing each request. For example, in the case of AppleTalk clients which request service from the MAT while it is busy with some other task like printing, the client keeps polling the MAT until the MAT is free. This increases the load on the network and may also reduce throughput in that there may be a gap between the MAT becoming free and the client discovering the MAT is free.

From the point of view of simplicity of design, the use of the interrupt mechanism is much better than using a polling mechanism. The technical limitations of DOS and Turbo do make the implementation of interrupts

[2]In this and all other results shown, the following applies. The times marked * were timed with a stop-watch, all other times were timed by the internal clock of the MAT. The same file is used in all measurements unless otherwise stated. All files contain only printable ASCII characters and the carriage return and line-feed characters. All measurements were taken when the MAT and its client were the only entities on the networks.

slightly tricky. However the use of the *state* variable for the communication on the Apple showed that interrupts are better to use because once they are working it is far easier to understand the logic of the program. Programming and maintaining are easier. The MICROLAN System [Fry 1987], operating with similar system limitations (CP/M) as the MAT, also used an interrupt driven approach.

The general problem of synchronizing distributed processes is known as a difficult one. The communication between the MAT and its clients does not need a high degree of synchronization. The method outlined in the previous chapter seems generally satisfactory. However, the protocol has not been rigorously tested. With proper language support, this would be easier.

5.3 PostScript

The type of Postscript file generated by the protocol converter is significant. In sending a text-only file to the printer, there are different strategies which can be adopted for protocol conversion. Different strategies give PostScript files of different sizes which take different times to run. Appendix D considers three strategies — called *B*, *C* and *P* — in detail.

The main difference between these strategies is the division of work between the MAT and the LaserWriter. For example, in strategy *B*, the MAT would produce PostScript code which consists of a line of code printing a line of text, followed by a line of code instructing the printer where the next line is to be printed. This is repeated for each line of text.

Under strategy C — the strategy adopted by the MAT — the code produced consists of: the definition of a procedure which when executed calculates where the next line of text should be printed; and for each line of text a line of code which prints the line of text and invokes the procedure to calculate where the next line should go.

Strategy *P* produces a PostScript program which has the text imbedded in it as data.

For any file which has more than 250 characters, method *C* will be more

efficient in terms of disk space used, and amount of network traffic than method B. For files larger than 3K, the saving will be between 18 and 20%. These figures are also conservative figures; they relate to pages which are full of text. Where there is more white space, the savings will be greater, as the ratio of overhead to text will increase.

As far as space is concerned, strategy *P* only betters the efficiency of *B* for files which are larger than 12K, although the larger the file gets, the greater the saving over *B* and *C*.

The advantage of off-loading work to the LaserWriter is that for large files there is saving in space. Unless a proper PostScript implementation is done on the MAT, the MAT is not going to be able to offer the full range of facilities which users will need. For example, justification and kerning[3] can only be done properly using PostScript features. On the other hand there is a price to be paid in performance. Section 5.5 examines the effect of different strategies on performance.

For the task of the MAT, PostScript proved powerful and versatile. Its use of escape sequences is annoying (some characters can not be referenced directly, but have to be referenced using their octal code), but this is not a serious flaw. Comparing the time taken by the LaserWriter (a PostScript implementation) and the Proprinter (a dot-matrix printer) presented in figures 5.1 and 5.2 shows that PostScript implementations can be slow. In this case, the PostScript implementation is approximately as fast as the dot-matrix printer (in fact, the Proprinter needs 78s before the MAT can continue processing, and the LaserWriter 85s).

The experience of the MAT is not sufficient to make a judgement on PostScript's use as a general page description language. PostScript is general and powerful: what needs to be assessed are questions of performance, and ease of use (by humans and applications), and translation between PostScript and other printer protocols.

[3] In many fonts, the space between characters is not constant, but depends on the characters concerned. *Kerning* is the process by which the correct inter-character space is left between letters.

5.4 Reliabili[ity] and Robustness

There are several aspects to reliability and robustness in the MAT which will be assessed separately. The MAT is not completely reliable, some of the problems being inherent in the design of the MAT, and others in the implementation.

Errors in the network

All communication between the MAT and its clients is highly reliable, that is, if the data reaches its destination, it has not been corrupted. Some of the communication is done using even more reliable protocols. The inherent reliability of local-area networks makes transient errors in the network extremely unlikely. Should packets of data be lost, the MAT would interpret this as a client crashing.

Failure of a client

If a client fails, then for the MAT to operate reliably and robustly, it must ensure that the failure of the client will not cause the effective loss of some resource (like memory or disk space). The current implementation of the MAT is robust and reliable in this case. Depending on the circumstance, the MAT will either deal cleanly with the problem, or print a junk file: space is reserved for the data from the client, even if the client fails, the printing procedure still prints out the contents of this reserved buffer.

The problem of printing a junk file is to some extent caused by the shortcoming of the Macbridge card (having to simulate interrupts). Nevertheless, an algorithm to ensure that it does not happen could be implemented with some cost to the understandability of the program. Far more important is whether the MAT is robust: the MAT should not fail because a client fails. Experience with the MAT shows that it is robust. Figure 4.4 shows a state diagram for the MAT's interaction with its AppleTalk clients, and figure 5.3 shows a similar diagram for the interactions between the MAT and its clients on the PC-net. These charts illustrate how the MAT is safe against its clients crashing. A more rigorous analysis would be necessary to prove that the MAT is robust. Furthermore, the experience of using the MAT has been gained in an experimental environment where it is difficult to duplicate 'typical' user use of the MAT.

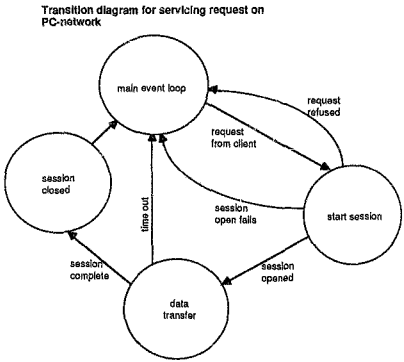


Figure 5.3 — state diagram for MAT

An inherent lack of reliability is caused by the design decision to use an interrupt handler as the process which transfers a file from the network into memory. If the machine should crash (say the power was switched off) before the file is saved to disk, the file would be lost, and there would also be no way of informing a client that this had happened. Given the constraints of the environment, this is acceptable. More importantly, the time in which the MAT is vulnerable to this sort of failure is relatively small. The type of use that the MAT is subject to would not warrant the complete change in design which would be necessary to obviate this risk.

Once the MAT has saved a file to disk, then the file will be printed, even if the MAT crashes. A list of files still to be printed is saved on disk. On starting, the MAT checks this list to see whether any files are still to be printed. This feature also allows the MAT to be brought down cleanly — a user on the MAT machine could stop the MAT program, and then use the

MAT machine for some other purpose. When the MAT program is restarted, files previously enqueued for printing will be printed. A corollary of this is that all files enqueued will be erased eventually, meeting another requirement for reliability.

Failure of a printer

A printer not working also causes unreliability, as printers often do not inform their users if something goes wrong. The MAT does not introduce this problem; it is there already (an IBM PC will report that it has printed a file, even if no printer is attached to it). Therefore, this does not affect the MAT's measure as a reliable system. One aspect where the MAT is occasionally unreliable is that it sometimes fails to print a file destined for the LaserWriter. The probable cause of this is that the Printer Access Protocols are not being used properly by the MAT. As there is no documentation for the use of these protocols with the Apple LaserWriter, this is to be expected. The MAT is however robust in these circumstances, and times out after a suitable interval.

5.5 Performance

Some of the discussion in this chapter has already had a relationship to the question of performance. A highly reliable system may lose some performance to attain a high level of reliability. In this section these factors will be examined again. There are a number of criteria for assessing a distributed system's performance. The ones which are discussed here are:

- network performance
- response time
- throughput
- use of resources

In chapter 3, the factors which network performance is most sensitive to were stated as: speed of the host and remote machine, level of high-level protocols, choice of transport layer protocols, and finally speed of the network [Lantz et al. 1985].

The load on the network is important for several reasons. Under heavy

loads, the use of the network will become a bottleneck (collision-sense multiple access protocols on buses are known not to be the most suitable ones at high-load [Stallings 1984]). For the type of network traffic which MAT-transactions generate, the number of bytes transferred is only affected significantly by the format of the file — other protocol information is a small proportion of total data sent (less than 100 bytes). The previous discussion of PostScript has shown how sensitive the length of a file to its format can be. (Note that even before this stage an important decision has been made — the characters are transferred to the printer rather than their bit-map representation. How information like this is transmitted is very important [Lantz and Nowicki 1984].)

What are the trade-offs as far as performance is concerned? To a large extent, this will depend on what a typical MAT transaction is.

If the LaserWriter becomes a bottleneck, then increasing the load on the LaserWriter by offloading processing to it may become counterproductive. The time saved on protocol conversion on the MAT, and any savings in network traffic will be lost by the MAT's throughput dropping, and increased network traffic caused by other clients having to retransmit requests. On the other hand, if there are a few, big files being sent to the LaserWriter, then the MAT may do well to offload all processing to the LaserWriter.

The MAT's strategy is a compromise one — network traffic and disk space is saved at the cost of the LaserWriter being forced to do more processing.

There are two times relevant to users. The time that the client program takes to send the file to the MAT, and the time that a job takes to print. Sending a file across the network takes a few seconds. The time that a user experiences could be significantly increased if the client instead of the MAT was to perform protocol conversion.

One of the important potential bottlenecks affecting the performance of a computer is the I/O speed. The MAT performs I/O when it saves files to disk after performing protocol conversion, and when it retrieves files for printing. I/O speed is very sensitive to the size of the I/O buffer. Figure 5.4 shows the effect of changing the size of the I/O buffer on time taken for I/O. Reserving 4K for this buffer to improve performance by almost an

order of magnitude is clearly worthwhile.

Figure 5.4 — Saving 11K to disk on an IBM AT.

Buffer size	1 byte	255byte	4K
Saving file (s)	13*	2.25*	1.64

Performance does not just depend on raw processor speed, but other factors such as memory available, and intelligent use of strategies like caching and buffering [Lampson 1983]. One example of this can be seen in the effect of caching on the LaserWriter. Some of the font faces on the LaserWriter are permanently stored in ROM (for example Times-Roman 12pt). Other fonts are only defined in one point size, and the LaserWriter has to calculate the other sizes. A further factor in the operation of the LaserWriter is that the LaserWriter caches the description of letters as it prints them.

When the LaserWriter, after having been switched on, first prints an 11K file in Times-Roman 12pt it takes about 112s to print, dropping to 95s the second and subsequent times the same file gets printed. The corresponding figures for Times-Roman 11pt are 138s and 101s[4]. As an 11pt document requires less work to print as it is smaller than a 12pt document, the use here of caching shows that significant improvements can be made with this strategy.

The choice of transport protocols is illuminating. Different protocols are used for communicating across the different networks. While this means that a different design has to be used for the routines responsible for the file transfer in the MAT, it does have the advantage of increased flexibility, allowing each network to use the most appropriate protocols, and allowing changes to occur where necessary. The use of different protocols also shows how important the performance of transport protocols can be (see also [Watson and Mamrak 1987]).

Two transport protocols are used on the PC-net. Datagrams are used to find the identity of the MAT, and to request service from the MAT. When the MAT accepts a request, a session is set up between the MAT and the

[4]These figures represent the time it takes for the file to print, rather than the time that the MAT waits for the LaserWriter to inform it that it can start with another job.

client. This session allows any number of *send* and *receive* commands to be issued between the MAT and client. Each *send* and *receive* command can transfer up to 64K of data. The experience of the MAT has shown that these are the appropriate protocols to use.

The AppleTalk Name Binding Protocol is used to find the identity of the MAT. It is much slower than using the broadcast mechanism that is used on the PC-net. For the request and the data transfer, the AppleTalk Transaction Protocol is used. Each transaction is capable of transferring about 4K of data. There seems to be a fixed overhead of approximately 6s for each transaction, with a slight variation depending on the length of the file. For the type of traffic which the MAT is used, these protocols are inefficient because of the length of time it takes to set up a transaction. A future implementation of the MAT should investigate defining an 'AppleMat' protocol built on lower-level AppleTalk protocols.

While the file transfer phase of the MAT takes much longer for clients on the AppleTalk than clients on the PC-net, the networks' bandwidths do not seem significant in this. The AppleTalk is a slow network for a local-area network, running at approximately 230Kbits.s^{-1}. But even this means that a file of 11K could be transferred in under a half a second using maximum network bandwidth. In practice a file transfer of a 4K file takes longer than 6s.

Figure 5.1 shows that for the PC-net clients, file transfer and saving to disk takes about 2.2s. The file transfer program supplied with the Macbridge performs the transfer of an 11K file between an IBM AT and the Apple Macintosh in approximately 3.5s (including the time for disk access). This compares well with the 2.2s needed for the MAT's PC-clients. Using exactly the same hardware, about 20s is needed to transfer an 11K file from one of the MAT's AppleTalk clients. This substantiates other research that bandwidth, although a factor, is not as important as other factors like level of communication and choice of transport protocols.

The results quoted in figures 5.1 and 5.2 underlie the small proportion of time spend on the data transfer phase. Of the time spent in the data transfer phase, only a small proportion of this time is spent transferring the data across the network. Improving the level of communication — for

example, the transport protocol — will be more valuable than improving the bandwidth of the network. Improving the speed of the processors (the MAT, the computer which does protocol conversion, and the LaserWriter) is also more important than improving network speed.

One of the goals of the MAT was to investigate whether such a printer server needs to be dedicated. The size of the MAT code is about 31K, and approximately 40K is reserved for data. From this point of view, the MAT does not have to be running on a dedicated machine. The processing requirements are more difficult to estimate. The only really processor intensive part of the MAT operation is the protocol conversion. Most of the rest of the time, the MAT is waiting for the printer to complete. In a true multitasking system, this time would be available to other tasks. Whether enough processing time is available for other tasks needs further research.

5.6 The layering of protocols

One of the aims of the ISO-OSI Reference Model is to hide lower level problems from users of higher-level protocols. It has been pointed out that implementing the full ISO-OSI Reference Model can cause performance to degrade [Tanenbaum and van Renesse 1985]. The experience of the MAT with the different transport protocols shows the positive value of abstraction and layering.

The NETBIOS session protocols, which were used to transfer files for PC-clients, allow data of up to 64K in size to be transferred with one *send* or *receive* command. The user designates the buffer where the data is or is to be placed by passing a pointer to this buffer in the network control block.

The AppleTalk Transaction Protocol place a limit of 4K on the amount of data which may be sent in one transaction. This in itself is a substantial limitation of the use of the protocol: it is reasonable to assume in the MAT environment that files will be smaller than 64K; it is unreasonable to assume that files will be smaller than 4K. Watson and Mamrak [1987] advocate that an arbitrary limitation on the size of transport packets should not be made: the transport protocols should take full responsibility for breaking the packet down into smaller packets for the network layer protocols.

However, what is worse is that the AppleTalk Transaction Protocol does not hide the underlying Datagram Delivery Protocol from users of the AppleTalk Transaction Protocol. A full AppleTalk Transaction Protocol packet comprises eight Datagram Delivery Protocol packets. To use the AppleTalk Transaction Protocol, a user must set up each sub-packet explicitly, and pass a pointer to each with the network control block. The data structures for this is shown in figure B1 in appendix B.

The AppleTalk Transaction Protocol, besides being slow, is unsuitable therefore because the service it offers is not adequate (the limitation on packet size), and is difficult to use as it violates the layering principles. However, it is unfair to write off the AppleTalk Transaction Protocol completely without knowing what its designers and implementors thought it would be used for.

5.7 Extensions, Criticisms and Changes

Obviously, the MAT has many flaws — some caused by the design, others by its implementation. Some are unavoidable because of hardware and/or software limits. These have been discussed already. In this section, what changes should and could be made to the MAT, and how they could be made will be discussed. This is useful for examining how flexible a system the MAT is.

The fundamental limitations of the MAT as a tool to assess distributed systems and printer servers generally are:

- the MAT is not typical of general distributed systems because it does work of a very specific type,
- PostScript was used for a only limited range of documents.

On the positive side, the heterogeneity of the environment and the flexibility of the MAT (to change the way protocol conversion is done, to use different transport protocols, to use different caching and buffering strategies) contribute to the MAT's utility as a tool to explore distributed systems.

An important part of any system is the user interface. However, this is not discussed in depth as it is largely a separate issue. For clients, the MAT should operate transparently: the usual print facility should be used,

which could be intercepted by MAT softwa e. Work needs to be done on the user interface, particularly on the Macintosh where the prompt driven approach is very clumsy. However, this is a user interface question: the MAT provides the basic underlying facilities. Figure 5.5 shows the dialogue between a user and the PC-net client code.

```
Entered the printing facility (MAT2.0)

Enter file name: client.pas

Enter file type:
ASCII, Screen, Pro: L

Enter font:
Times, Helvetica, Symbol, Courier: T

Enter font size: 12
```

The text in Roman (plain) font is produced by the client program, the text in italics is typed in by the user

Figure 5.5 — Dialogue between user and client code

One of the requirements catered for in the design, but not implemented, was informing the user if a file could not be printed. Problems could occur at three phases:

- when the file is transferred into memory
- when the file is saved to disk
- when the file is printed or an attempt is made to print it.

At present, the client gets a reply (printed or can't print) at the first stage. Are the alternatives better?

In order to receive any message from the MAT, software has to be active within the client to receive the message. In the present environment, this would mean that the client would not be able to do anything else until the file had been printed. For this reason, a reasonable compromise would be to inform the client if anything goes wrong in the file transfer phase (as is done now), and also allow the client the option of waiting for a confirming message once the file had been printed, or failed to print. As the identity of the client is kept with the file's request block, this could be done without

major change to the design of the program. Care must be taken that suitable time-out values are chosen so that the client does not time-out before the MAT responds, yet does not wait excessively if the MAT dies.

Other work to improve reliability should be done, with a rigorous examination of the protocols involved.

The functionality of the MAT could be improved. Useful features would be to allow a user to query the print queue, and cancel print jobs. These features could be integrated into the present system easily. The print request block has a number of fields which are not used in the present implementation. At present, the MAT assumes that all requests are for print jobs. This could be changed by the MAT examining a field in the print request block. If the request was to cancel a job this could be done reasonably easily if that particular file was not printing. By using a global variable to indicate that a print job should be canceled, the print routines could all be changed so that they poll this variable to see whether the present print job should be canceled. For a request to examine the print queue, the names of the files or other information could be sent back to the client. This feature is available in other systems, for example the printer spooler on MacServe [Infosphere 1986].

Other issues are are:

- Protocol conversion routines should be written, and replace the stubs already there. With more protocols being supported, some files would undergo more that one protocol conversion: a file in format X to be printed in format Y, would be translated into PostScript before being saved on disk. In this case, thought would have to be given to where protocol conversion should take place. Doing the second protocol conversion on the *fly when printing the file* might be an efficient way to overlap processing and I/O time, especially as the printer is the bottleneck, not the MAT or the network.
- A limitation of printing one copy of a file was made. This is an unnecessary restriction. The print request block could easily carry the extra information required, and changing the print routines to print more than one copy would also be straightforward.

There are also major changes in design which could be considered. The

AppleTalk network allows the definition of a user-defined protocol. Nodes on the network (Macintoshes, and the Macbridge AppleTalk card) support this with the use of socket listeners. Socket listeners are low-level routines (written in assembler) which examine packets received on the network to see whether any packets need to be serviced by any process on that machine. If service is required, they are examined further, and the necessary routine is called to do this. All of this is done transparently to any process or program active on the machine at the time. While this still does not allow complete concurrency, such an approach would remedy the problems encountered with the Macbridge card.

If the MAT program were written in a language which properly supported concurrency, redesigning the MAT might lead to a cleaner design which was more maintainable and flexible.

Despite the problems experienced with the Macbridge card, the feasability of linking different networks to the MAT was shown. Although the routines which serve the different networks do manipulate common data structures, with sufficient forethought, this does not create problems. The design of the MAT allows integration of other networks into the system.

5.8 Summary

This chapter has examined the MAT as a printer server, and as a tool to examine resource sharing systems. Changes and extensions to the MAT were also proposed.

Particular problems encountered with the MAT were the limitations caused by the implementation environment, and the lack of proper synchronization and mutual exclusion primitives. Nevertheless, the use of interrupts has allowed a certain concurrency.

At present, the protocol conversion routines are limited, and more need to be written. They can be integrated into the present implementation easily. Although PostScript has not been tested in depth by the MAT, it is an acceptable method of page description, as far as its has been used.

The MAT is reasonably robust, although it is not completely reliable. For the type of work that the MAT is doing, the level of reliability is acceptable.

The performance of the MAT is good, and it meets its requirements in this respect. The memory requirements are small. Printing a file using the MAT takes less time than printing directly to a printer. This is true both for the time which the client machine is busy, and the actual time the document takes to print[5]. There is no other system which the MAT can be compared to directly. On the Macintosh, printing a file to a dot-matrix printer is faster using the MAT than MacServe because MacServe expects the client machine to do protocol generation. The Macintosh spoolers for the LaserWriter are relatively unreliable — a factor which militates against their use. The PC-net's printer spooling program (the software supplied with the NETBIOS card) has a performance similar to that of the MAT program. However, the server program has code about 60K in size, and has memory requirements of about 250K while running.

The impact of PostScript and other protocol conversion needs fuller assessment. Transport protocols on the AppleTalk are not suitable for the type of work the MAT is doing, and performance should improve with the use of appropriate protocols. The results of the MAT have substantiated other work in relation to factors influencing performance.

Overall, the MAT's design has proven good. The current implementation has some flaws, but these could be remedied without major change to the design. Performance is good, and the MAT — even in its present implementation — can perform a useful task. The MAT meets its requirements of usefulness. The versatility of the design would allow greater functionality, but, for major changes, a change in development environment is recommended.

[5]Printing the 11K file directly to the LaserWriter from an Apple Macintosh using MacWrite takes 155s for the Macintosh to complete and 200s for the LaserWriter to finish printing the document.

6. Conclusion

Resource sharing across networks and internetworks is feasible, and has many benefits. This dissertation has been an investigation of how resources can be shared and an examination of important factors affecting distributed systems' design and implementation.

This chapter concludes the dissertation by analysing:

- The implications of the ISO-OSI Reference Model on data communication and resource sharing,
- Factors affecting the performance and design of distributed systems
- The management of printer resources, and a model for printing services
- The needs and requirements of a printing service.

This contributes to an understanding of important issues in distributed systems and mechanisms for the provision of printing resources.

6.1. The ISO-OSI Reference Model

The ISO-OSI Reference Model has had a profound effect on the design of protocols for data communication. The principles of layering and abstraction are very useful in the design and implementation of protocols, while standardization increases the possibility of achieving communication. Implementing the full model may be costly as each of the different layers may require processing [Tanenbaum and van Renesse 1985], while enforcing one standard may also lead to inefficiency in some circumstances [Notkin et al. 1987].

Nevertheless, it is important to recognize the benefits of layering and standardization [Popescu-Zeletin 1984]. The experience of transport protocols with the MAT presented in chapter 5 shows this. Also, the model may be used for design, and not necessarily implementation [Watson and Mamrak 1987]. Another factor in this is the performance of operating systems and the support given by the system.

For internetworking (where the underlying physical media of the networks are different), techniques defined by the ISO-OSI Model can be used to promote the interconnection of different equipment. There have been a number of successful internetworks [Quaterman and Hoskins 1986].

However, at the higher levels of the model — where higher-level information is interchanged — it is not the physical difference between the underlying communication media which is the cause of the differences in the system. The focus moves from internetworking to heterogeneity [Cole 1987]. There are a number of different styles of heterogeneity [Notkin et al 1987]. There is a loose coupling typified by network operating systems and systems like the MAT. Here, a few services are shared by servers. The resources shared are still owned and managed by local systems. New systems can be integrated fairly cheaply.

On the other hand, there are systems like Locus and the V-system which are tightly coupled. A high degree of transparency imposes a homogeneous interface between applications and the system.

It is at this level that the ISO-OSI Reference Model is not complete. Standards for network and resource management and distributed systems generally are now under development [Aschenberger 1986; Hutton 1987; Roos 1987]. Individual protocols such as file transfer and virtual terminal protocols have been developed, and progress has been made with directory systems' standardization [Goodwin and McDonnel 1986]. However, it is the standardization of distributed systems as a whole which is necessary. As will be seen in the discussion on printing services, the definition of specific protocols is not sufficient for resource sharing.

Standardization of distributed systems is important for integrating existing systems. Successful systems like the V-system and Locus cannot be integrated easily.

The present state of the ISO-OSI Reference Model does not seem to provide a useful guide for designers of distributed systems. The substructure of the application layer [Bartoli 1983] viz. user element, specific application service element and common application service element, is not used by designers of distributed systems. A recent text book on the subject [Fortier 1986] only mentions the ISO-OSI model once. Work reported in the

literature only mentions the ISO-OSI model in connection with transport and lower functions.

6.2. Distributed Systems

In chapter 3, a general assessment of distributed systems was made. This section examines some of the issues in the light of the experience of the MAT.

Distributed systems have proved that adequate performance can be obtained, even in a network and internetwork environment. Experience of distributed systems — experience which the MAT confirms — is that the most important factors in the performance of distributed systems are:

- performance of local and remote computers
- high-level protocols
- transport protocols
- bandwidth

The performance of computers does not just depend on raw processor speed, but also an I/O speed, availability of memory and secondary memory, and the intelligent application of strategies such as buffering and caching. Performance of the operating system and device drivers has a great impact on other factors. For example, up to 80% of the time spent on inter-process communication can be taken up by the operating system and other system work [Watson and Mamrak 1987]. For this reason alone, operating systems need to be developed to adapt to a networking and internetworking environment.

Transport protocols affect the performance of systems. It is not just the performance of transport protocols which is important but their functionality. The problem which the MAT experienced with the AppleTalk Transaction Protocol is a more general problem. It has been argued that transaction protocols should hide implementation details of lower-level protocols from session and higher-layer clients. In particular, the size of network layer packets should be hidden; arbitrary limits on the size of transport-layer packets should not be made [ibid.]. Full error control and recovery may not be necessary as higher-level protocols will still have to cater for errors.

The functionality of transport protocols is also determined by what the operating system can provide. When a particular system attempts to have different transport connections at the same time, some sort of concurrency is necessary: "A transport protocol can provide no more service than the level of process management allows" [Cole 1987]. This was seen with the MAT. On the one hand, the network cards allowed different transaction connections to be open, on the other the system did not support the use of this properly. This could be particularly seen by the limitations which were imposed on the MAT when it serviced AppleTalk clients.

It is not the intention of this dissertation to say that lower-level protocols and bandwidth are unimportant. Clearly, they are important. Especially under high loads, protocols such as medium access protocols and bandwidth are important. As applications which need high bandwidth (including those using video signals) grow so will the need for high bandwidth. What this research has shown is that high bandwidth is only one part of the performance of a distributed system. If the other factors have poor performance, then increasing the bandwidth is not going to be a solution.

Another factor in the design and implementation of distributed systems is that system and language support are necessary. This was seen in the MAT and the results of others [Black 1985; Fry 1987; Morris et al. 1986]. Supporting concurrency within a system is especially important. The importance of this from a performance point of view has already been pointed out. The experience of the MAT and others also shows that the need exists for language support to promote simplicity and correctness in program design and implementation. One aspect which this dissertation has not examined at all is languages for specification of distributed systems. Examples of such languages are Estelle, Lotos and SDL [Saracco and Tilanus 1987; Specs Consortium and Bruijning 1987].

6.3. Printing Services

For the purposes of the MAT, PostScript proved a general and powerful printer protocol, although more assessment would be needed in a more general environment. However, even with the specific types of print jobs with which the MAT was dealing, a number of strategies for protocol translation exist.

In a more general system where there are a number of servers, and a greater number of printers on the network, greater printing management needs to take place. PostScript is good at describing in a device independent way what a document looks like. However, by itself, it does not provide information which manages the printing resources generally. The Adobe structuring convention [Adobe1985a;Adobe 1987] supplements PostScript by structuring a PostScript file in such a way that the resources which a print job needs are explicitly available to other programs such as printer spoolers[1]. A description of the Adobe structuring convention can be found in appendix C. Certain 'higher-level' information such as which fonts a document needs, what resolution the author of the document expects, what colour paper etc. either need to be known or assumed by the server which takes responsibility for the job.

The MAT accomplishes this by including some information in the request block. However for sophisticated documents, this type of strategy will not by itself be adequate, as in the general case, the meta-information may be quite substantial. The server executing a job should be able to extract the meta-information from the document quickly and flexibly — something which use of the structuring convention allows.

Once the server has this information, it should decide which printer the job should be assigned to: this may be explicitly stated by the user, or decided by the server on the basis of the resources needed by the job, and the resources available to the system. For example, a particular network may have two laser printers, one of which has fonts permanently defined that the other does not. A server would be able to send jobs which only need

[1]In a PostScript program some of the requirements of the resources which a print job needs are implicitly stated in the code (for example which fonts are needed). However, this information is not readily available. Furthermore, there are some requirements which the server needs to know which are *not* stated in the code.

the common fonts to either printer. Jobs which needed the fonts only available on the one printer would have to be sent to that printer. (Or, in the case of PostScript printers, if that printer happened to be very busy, it might be possible to embed the definition of the needed font in the print job, and send the job to the other printer).

This leads on to the question of the division of responsibility for the work that a print job needs. This discussion ties together some of the discussion of previous chapters.

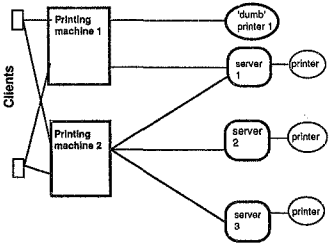
- At the end of section 3.2, the trade-off between a client and the server performing protocol translation was stated. The printer server performing this improves the performance of the client, reduces client complexity and allows greater flexibility. However, under heavy loads the server may become a bottle-neck.
- In section 5.5, the trade-off between the MAT performing protocol translation and the LaserWriter performing protocol translation was stated. Unless a full implementation of PostScript is available on the MAT, the LaserWriter will always provide a more flexible service. Generally, the approach of the LaserWriter performing the protocol translation would also save disk space and network traffic. However, at present, PostScript engines are relatively slow. The performance of the LaserWriter is poor relative to that of the MAT. The optimal solution would depend on the mix of jobs in the system.

The experience of the MAT leads one to believe that the spooler-device controller model [Janson et al. 1983] is not always appropriate. In the MAT's system, the MAT acted as spooler and device controller in the case of the dot-matrix printer, and in the case of the LaserWriter as a spooler only. The LaserWriter acted as device controller. However, it should be recognized that the LaserWriter is itself a server. The LaserWriter in the system was somewhat limited in memory and processor speed, but the Printer Access Protocols which it uses make the LaserWriter a server: it takes one job at a time and prints it.

As PostScript becomes more popular and memory and processor performance/cost ratios increase device controllers are likely to become more intelligent. Each device controller may, as in the case of the LaserWriter, become a printer server with which clients could

communicate directly. This increases the heterogeneity of the system.

A proposed solution is the *printing machine* model. This printing machine would act as a client of all the printer servers on the network and act as device controller for all non-intelligent printers. Clients would interact directly with the printing machine which would coordinate jobs among servers and printers. The relationship between clients of a printing service, the printing service and the printer servers is shown in Figure 6.1



Clients can access the printers through the printing machine. In this example, there are two printing machines (i.e. two different printer services). Printing machine 1 can give its clients service from server 1, and can also directly print a file to a printer attached to it (the MAT looks like printing machine 1). Printing machine 2 implements an independent printing service, and uses servers 1-3 to do this. In both cases the printing machine could be distributed.

Figure 6.1 — The Printing Machine

System wide management of printing resources would be performed by the printing machine. The burden of complexity would fall on the printing machine, although for performance or other reasons, either clients or the various servers may be encouraged to perform some of the protocol translation.

The advantage of the printing machine approach is that increased flexibility and performance can be provided to clients, while at the same time providing the clients a common interface to the printing service. One of the reasons for this would be improved utilization of printing equipment system-wide. Another is the use of buffering and caching. By taking cognizance of the fact that printers are becoming more intelligent, the printing machine would promote flexibility. Reliability would be promoted by the printing machine being able to switch jobs to spare printers should some printer fail. Printing services could grow dynamically. The printing machine could be a distributed machine — operating on a number of computers at the same time.

The printing machine/printer server concept is analogous to the filing machine/file server concept of the Tripos filing machine [Richardson and Needham 1983]. Another system where different filing systems can be implemented on top of the file servers on the network is the Amoeba File System [Mullender and Tanenbaum 1985].

Requirements for printing services

This discussion attempts to specify the requirements and needs for printer services. The printing machine concept is not assumed for any of these requirements, but it is submitted that because the printing machine concept takes into account the distributed nature of the printing service, it suits these requirements and needs better than the spooler-device controller concept.

- Printing systems need much of the same system and language support that distributed systems generally need. In particular, buffering and caching needs to be supported, as well as fast I/O.
- Printer protocols must be general, powerful and efficient. Device-independence in these protocols is extremely important.
- For system-wide management of printing resources, information about printing jobs needs to be readily available. Preferably, a higher-level protocol like the Adobe structuring convention should

be used.

- The semantics of error reporting and the limits of reliability in a system need to be clearly defined and understood by the users of the system. An example of the complexity of the problem is as follows. A user sends a job to the printing machine, and the printing machine sends the job to a printer server. Even once the server has accepted a job, something can go wrong. For reliability, the printing machine should keep a record of the job until it is informed by the server that it has completed[2], before informing a user that a job has printed.

- Reporting back to a user is also difficult as when a print job completes, the service provider cannot be sure that the same human user is still at the client machine. In a reliable environment, reporting back may be an unnecessary complication. Far better would be to allow a user to query the service provider to find the status of their job.

- In the case where ultra-high reliability is demanded, there may be no alternative but for a printer to be attached directly to a computer.

- Where security is demanded, the operation of the printer service must be strongly idempotent: an eavesdropper should not be able to copy the data being transferred across the network, and at a later stage be able to produce copy by sending this data to the printer. It is likely in such an environment that cost will not be a factor. But, providing security is still a general problem[3]. Users should be allowed privacy.

---

[2]Note that this complication is not introduced by the printing machine concept. A similar problem can be observed by printing to a dot-matrix printer with a buffer. At some stage before the job completes, the printer informs the computer that the job has safely printed. If the power were to fail immediately after this, the job would not finish printing. Some printer drivers quite happily 'print' a file and inform a user that the file has 'printed' even if there is no printer attached!

[3]Consider the MAT's environment. The LaserWriter is used to print exam papers. How long will it be before students use network monitors to get exam questions? This is technically possible.

6.4. Summary

This dissertation has examined how resources may be shared across heterogeneous networks. The first chapter introduced preliminary concepts, and justified the area of research. In the second chapter, a framework for data communication and resource sharing was described, as well as the impact of internetworking on protocols. The survey of file servers, printer servers and distributed systems in chapter 3 showed that the area of research is a fruitful one, but that there are many problems and areas which need to be explored further.

The design and implementation of the MAT — a printer server connecting two different networks — were described and assessed in chapters 4 and 5. Many of the general results of distributed systems applied to the MAT.

This chapter has discussed some of the concepts raised in the dissertation in the context of the MAT, other work and the theoretical framework. It has clarified important issues for distributed systems, and has proposed an extension to the model for printing services.

Appendix A. NETBIOS

The IBM Network Basic Input-Output System (NETBIOS) [IBM, 1984] is the interface between an IBM computer (PC, XT or AT), and a network adapter which is inserted into one of the computer's slots. The network adapter is the connection to a network—either a PC-net, or IBM token ring network.

In the case of the PC-network adapter, the adapter is a card which contains an Intel 80188 microprocessor, a communications controller, approximately 300K of ROM, 128K of RAM, and other controllers. The card is capable of direct memory access to the host computer's memory. The card's ROM performs the functions of protocols of the first five layers of the ISO-OSI Reference Model.

The NETBIOS commands are session layer protocol commands. A host computer performs NETBIOS commands using network control blocks (NCBs). An NCB is a block (logically, a record) in memory. An NCB can be described as

```
ncb_block = record
   command   : byte;
   retcode   : byte;{return code}
   lsn       : byte; {session number}
   num       : byte; {number of name}
   buffer    : double word; {pointer to memory}
   length    : word;{length of buffer}
   callname  : packed array[1..16] of byte; {address on network}
   name      : packed array [1..16] of byte;
   rto,sto   : byte; {receive and send time-outs}
   post      : double word; {pointer to a procedure}
   lana_num  : byte; {number of adapter}
   cmd_cplt  : byte;
   reserved  : packed array[1..14] of byte;
end;
```

The *command* field contains a number indicating the command which should be executed. The *retcode* field returns a number indicating whether the command succeeded, and if not, an error code. The *lsn* field is used when a session, or a virtual circuit, is set up between two communicating parties; it contains the local session number, a unique number given to each open session to identify the session.

A particular application or user can register a name on the card to identify itself. When this is done, its name is given a number. In any command where the application's name is needed, the *num* field is given the relevant name number.

The *buffer* field contains the address of the place in memory where the data to be transferred are placed, or the area in memory where data are to be transferred from. *length* contains the length of the buffer.

The name of the application or user being addressed on the network (usually one on another machine, although the network commands could theoretically be used by two users on the same machine to communicate with each other) is placed in the *callname* field. The name of the application issuing the command is placed in the *name* field. The *rto* and *sto* fields indicate the time-out values (in 500 ms intervals) for receive and send commands respectively. The *post* field is a pointer to a procedure. Its use will be explained later.

More than one adapter can be placed in a machine: the *lana_num* field specifies which adapter card should be used.

The *cmd_cplt* field indicates the status of a command being executed. This allows the monitoring of the status of commands executed asynchronously. The *reserve* field is not used by a user application.

A.1. Commands

A command is executed by setting up an NCB in memory with appropriate values, and placing the necessary data in a buffer. The address of the NCB is placed in the ES:BX register pair, and the the interrupt $5C_{16}$ is generated. Control is thereby passed to the adapter card.

Commands may be issued synchronously or asynchronously. If they are issued synchronously, the calling procedure is blocked until the command completes. The completion code is placed in *retcode*.

If the command is issued asynchronously, the computer may continue processing. A code is placed by the adapter card in the *retcode* field which gives the immediate return code. If this is good, it means that the

command is being processed. Once the command completes, the final return code is placed in the *retcode* field. The *cmd_cplt* field may be monitored to see whether the command has completed. If it contains FF_{16}, the command has not yet completed, if it contains any other value then it has completed. If the *post* field is given the address of some procedure, then when the command completes, the host computer is interrupted, and control is passed to this completion procedure. The procedure must perform a return from interrupt instruction when it finishes.

There are four categories of commands.

General

The general commands are *reset*, *cancel*, *adapter status*, and *unlink*.

The *cancel* command allows a previously issued command to be cancelled. The *reset* command resets a specified adapter card. This clears all name and other tables from the card, and allows the setting of certain parameters—for example there is a trade-off between the number of names and sessions which a card can support.

The *adapter status* command interrogates a specified adapter card for its status. Included in the information received is the identification of the card[1], results of its last self-check, traffic and error statistics, resources available on the card, the number of names registered in the local name table, and the local name table.

The *unlink* command is a specialized one used when a computer is linked via adapter cards to another computer at boot up time.

Name support

The name support commands allow the computer to be known by a name on the network. Names can be group names (this allows a group of computers to receive the same messages), or individual names. Besides the permanent name on the card, each card can have another 15 names.

The *add name* command allows a name to be added to one of the local

[1]Each card is given a unique name—the 16 byte name which can be used in the *name* or *callname* fields.

adapters. The *add group name* command allows a group name to be added to one of the local adapters. The same group name can be used on different adapters, other names must be unique. The *delete name* and *delete group name* commands delete names from a card.

Session support

The session support commands allow a logical connection or session to be set up between two communicating entities, identified by names. Starting a session also allows the communicating parties to set certain parameters for the session. Once a session is started, the parties can transmit and receive data. This data transfer is reliable—exactly-once transmission is supported.

A session is started by one entity issuing a *listen* command. It can specify who it wants to start a session with, or that it is prepared to start a session with anyone. The other entity must then issue a *call* command, specifying the first entity in the *callname* field. The various time out values are stated in the *listen* and *call* commands. If the session is started, each entity is given its session number in the *lsn* field[2].

Once the session is started, the *receive*, *send*, and *chain send* commands can be used to send and receive data. Each *send* and *receive* command can transfer up to 64K of data. The *receive any* command can be used by a computer to receive data from any other entity which it has a session open with.

The *hang up* command is used by both sides to close the session. The *session status* command can be used to find the status of all sessions active on an adapter.

Datagram support

Datagrams allow unacknowledged messages to be sent between two names. Messages are limited in size to 512 bytes, and this together with their unreliability makes the use of datagrams more tricky than sessions.

The *send datagram* and *receive datagram* commands are used to send and receive datagrams. 'Broadcast datagrams'—datagrams which can be read by all nodes on the network—can be sent and received by the

[2]The two numbers need not be the same, as they are only used locally.

send broadcast datagram and *receive broadcast datagram* commands.

A.2. Remote program Load

The PC-network allows a computer to boot from another computer at start-up time. To use this, some hardware settings need to be changed,and the adapter cards must be set.

The facility allows the introduction of diskless machines, as users will be able to worf off some central file server

Appendix B. AppleTalk protocols & MacBridge.

The purpose of this appendix is to give an outline of the technical details of some of the AppleTalk protocols, and the use of the Macbridge card. More detailed information can be found in [Apple undated; Appie 1985, Sidhu and Oppenheimer 1985; Tangent Technologies 1987].

B.1. AppleTalk protocols

The protocols implemented for the AppleTalk fall roughly within the ISO-OSI Reference model. On the Macintosh, various protocols can be used by application programmers directly. Programmers may also define their own protocols which use some of the lower level protocols. The description of the use of the AppleTalk protocols here uses *Lightspeed Pascal* functions which can be used by Macintosh programmers. All languages for the Apple Macintosh should support equivalent functions.

Link Access Protocol

The link access protocol (ALAP) is implemented at the physical and data-link layers. Its purposes include to provide access control, provide a node addressing mechanism and ensure packet integrity. ALAP uses collision-sense-multiple-access with collision avoidance as a medium-access protocol. Each device on the network is given a node number. Each ALAP packet contains 600 bytes of data, together with bytes which contain header and trailer information

The ALAP protocol can be manipulated with the *LAPWrite* and *LAPRead* procedures. There are also calls which allow the use of user-defined protocols—the ALAP procedures examine APLAP packets received to see which higher level protocols should handle the contents of the packet.

Datagram Delivery Protocol

The Datagram Delivery Protocol (DDP) is a network level-protocol. DDP sends packets between *sockets*—sockets are logical entities defined on nodes. AppleTalk addresses are uniquely determined by a node, socket pair. With each socket, a socket listener is defined. This is code which examines incoming DDPs, and performs necessary operations on packets destined for that socket. This is the way that higher-level protocols are implemented. The AppleTalk Transaction Protocol, for example, is a higher-level protocol which is defined by means of a socket listener.

Procedures to implement DDP include *DDPOpenSocket*, and *DDPCloseSocket* which allow clients of the DDP to open and close sockets. *DDPRead* and *DDPWrite* allow clients to read from and write to the network DDP packets.

AppleTalk Transaction Protocol

The AppleTalk Transaction Protocol (ATP) provides a reliable (at least once), loss free transport service. Optionally, ATP messages can be of the exactly-once variety. The DDP protocol is used by the ATP to do acknowledgements—the way this is done is not the concern of the ATP clients. Each transaction is given a unique transaction identifier.

ATP uses the request-response paradigm. One network entity sends a request to another which responds to the request. The responding end opens an AppleTalk socket using the *AtpOpenSocket* call, and then issues an *ATPGetRequest* call to wait for a request from some other node. The requesting node issues an *ATPSndRequest* specifying the address of the responding node. This request sends a data packet to the responder which can be interpreted by the responder. When this packet arrives at the responding end, the *ATPGetRequest* completes[1], and the responder examines the request. The *ATPSndRsp* command can be used to send a response to the requesting end. The maximum amount of data which can be transferred by one transaction is about 4K.

[1]As with other calls, this call may issued synchronously or asynchronously. If it is issued synchronously, the caller is blocked until the call completes. If it is issued asynchronously, the program must monitor the command status to see when it completes.

One of the parameters for the *ATPSndRsp* call is a pointer to an array of up to eight buffers. Each of these corresponds to a buffer used by the underlying DDP calls which are used to implement the ATP. See Figure B1.

There are other calls which allow the managing of transactions, and calls which are functionally identical to those described above, but use different memory management techniques.

Name Binding Protocol

The Name Binding Protocol (NBP) allows users of the AppleTalk protocols to associate a name with a node, socket pair. A name consists of three fields, an object field, a type field and a zone field. Users of the NBP are free to associate any strings with less than 32 characters in the object and type field. The object field is used to give a name to some object on the network, while the type field indicates what type of device the object is. For example, a LaserWriter would have type 'LaserWriter', and a user of MacServe would have type 'MacUser'. The zone field is used when multiple AppleTalks are connected to each other, A zone is an arbitrary subset of the AppleTalks. The symbol '*' means *this network*, and in the case where there is only one AppleTalk network, this can be used exclusively.

The *NBPRegister* procedure allows an NBP client to register an name on a socket, while the *NBPRemove* call deletes a name. The *NBPLookUp* command examines all the names on the network, and places all the names which match in some way a specified name into a buffer. The *NBPExtract* command, which is not strictly a network command, can extract useful information from this buffer.

Printer Access Protocol[2]

The Printer Access Protocol (PAP) is a protocol which is designed to allow users on the network to communicate with an intelligent printer such as the Apple LaserWriter. This printer acts as a printer server in the sense that it takes one job at a time from clients on the network, and prints them.

[2]The PAP protocol calls are not documented for the Apple Macintosh. They are defined in Sidhu and Oppenheimer 1985, and can be called from the Macbridge card.

There are PAP protocol calls for both the the server (the printer) and the clients. The *PAPRegName, PAPRemName, SLInit, GetNextJob* calls allow the server to register and remove its name, to perform initialization routines, and to get the next job.

The *PAPOpen* and *PAPClose* calls allows sessions to be opened between a server and its client, while the *PAPRead* and *PAPWrite* calls allows PAP packets to be read from and written to the network.

The documentation for the way in which the PAP protocols are used by the LaserWriter are poor. The *PSDump* program [Oppenheimer 1985] can be examined to see how it is used.

B.2. Macbridge

Whilst on the Macintosh the protocols can be manipulated using predefined procedures, users of the Macbridge adapter card which connects an IBM computer to the AppleTalk network must use a lower-level approach.[3]

The approach is much the same as using NETBIOS. An area of memory is set up with certain parameters. The DS:BX register pair is given the address of this area, and an interrupt (60_{16}) is issued, and the command is executed. The data structure is much more complicated than that for the NETBIOS commands because several protocols are supported. Each protocol uses a variant of a general record for the data that it needs. The data structure for ATP calls is as follows:

[3] A similar approach is also used on the Macintoshes when assembly language programs are used to manipulate the AppleTalk protocols.

```
ATPParams = record
     atd_command    :  integer;
     atd_status     :  integer;
     atd_compfun    :  memory address;
     atp_addr       :  AddrBlk;  {AppleTalk Network address}
     atp_socket     :  byte;     {local socket#}
     atp_fill       :  byte;     {filler}
     atp_buffptr    :  memory address; {request buffer ptr}
     atp_buffsize   :  integer;  {size of this buffer}
     atp_interval   :  byte;     {retry interval}
     atp_retry      :  byte;     {number of retrys}
     atp_flags      :  byte;     {control info}
     atp_seqbit     :  byte;     {sequence number}
     atp_tranid     :  byte;     {transaction id}
     atp_userbytes  :  packed array[1..4] of char;
     atp_bdsbuffs   :  byte;     {number of buffers for response}
     atp_bdsresps   :  byte;     {number of buffers filled}
     atp_bdsptr     :  memory address {pointer to buffers}
end;
```

Here, the use of the fields common to all protocols is summarized.

The *atd_command* field is a word long. A code representing a command is placed in it. The status of the command can be found in *atd_status*.

As with NETBIOS, commands can be issued synchronously or asynchronously. If the command is issued synchronously, the calling program is blocked. If the asynchronous option is used, the *atd_status* field can be checked to see when the command completes—it contains '1' until it does complete. For both the synchronous and asynchronous modes, the address of a procedure can be placed in *atd_compfun*. This procedure will be executed when the command completes. In this case the currently executing program will be interrupted, and control passed to an interrupt handler on the Macbridge. This handler will then set up a small stack next to the heap, and call the procedure referenced to by *atd_compfun*.

```
ATPParams = record
     atd_command    :  integer;
     atd_status     :  integer;
     atd_compfun    :  memory address;
     atp_addr       :  AddrBlk;  {AppleTalk Network address}
     atp_socket     :  byte;     {local socket#}
     atp_fill       :  byte;     {filler}
     atp_buffptr    :  memory address; {request buffer ptr}
     atp_buffsize   :  integer;  {size of this buffer}
     atp_interval   :  byte;     {retry interval}
     atp_retry      :  byte;     {number of retrys}
     atp_flags      :  byte;     {control info}
     atp_seqbit     :  byte;     {sequence number}
     atp_tranid     :  byte;     {transaction id}
     atp_userbytes  :  packed array[1..4] of char;
     atp_bdsbuffs   :  byte;     {number of buffers for response}
     atp_bdsresps   :  byte;     {number of buffers filled}
     atp_bdsptr     :  memory address {pointer to buffers}
end;
```

Here, the use of the fields common to all protocols is explained.

The *atd_command* field is a word long. A code representing a command is placed in it. The status of the command can be found in *atd_status*.

As with NETBIOS, commands can be issued synchronously or asynchronously. If the command is issued synchronously, the calling program is blocked. If the asynchronous option is used, the *atd_status* field can be checked to see when the command completes—it contains '1' until it does complete. For both the synchronous and asynchronous modes, the address of a procedure can be placed in *atd_compfun*. This procedure will be executed when the command completes. In this case the currently executing program will be interrupted, and control passed to an interrupt handler on the Macbridge. This handler will then set up a small stack next to the heap, and call the procedure referenced to by *atd_compfun*.

AppleTalk Transaction Protocol
Data structure for ATPSendRequest

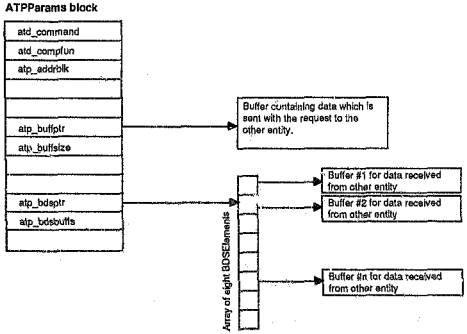


Figure B1

For the ATP calls, the other fields are used in the following way. *atp_addrblk* is the network address of the entity being communicated with. *atp_socket* is the local socket number being used for the communication. The *atp_buffptr* and *atp_buffsize* fields describe the buffer with the request information in it. The *atp_bdsptr* points to the response BDSElement array. (This is an array of up to eight records each of which has a pointer to a buffer for the data which will be received, and a field showing the size of the buffer). This is shown in figure B1 which describes the data structure for the *ATPSndRequestCall*. The *atp_flags* contains control information, including whether the transaction is at least once or exactly once, and whether the final packet of the transaction is being transmitted.

Appendix C. PostScript

This appendix introduces PostScript programming, and describes the Adobe convention for the structuring of PostScript files.

C.1. PostScript

Full details of PostScript can be found in Adobe 1985a and 1985b. Machanick presents a useful tutorial series [Machanick 1987a-h].

PostScript is a page description language. It has facilities for drawing text, images (for example, scanned photographs), and geometric figures (lines, arcs etc.) on a page (collectively called objects). In order to place an object on the page, the object must be defined, and then placed at an appropriate point.

A PostScript 'virtual' page is of infinite size, with a Cartesian coordinate system used to refer to particular points on the page. The origin of the page is typically at the bottom left hand point of the physical page which will be produced from the virtual page. The origin may be translated on the page, and the coordinate system may also be otherwise transformed and rotated. The units of the coordinate system are of point size (approximately 1/72 of an inch) when a program starts to run. Each pixel on the page can be of any colour and any intensity. Obviously, most PostScript supporting printers do not support all of these features. An Apple LaserWriter for example is a monochrome printer with a two-level intensity for each dot—on or off. A high resolution (300 dots per inch) allows a simulation of a grey scale, as PostScript's virtual resolution is 72^2 pixels per square inch.

The PostScript language itself is a stack based one, with post-fix notation being used. Here follows a brief introduction to some of the PostScript operators.

General

The following series of programs give an idea of how PostScript works The line numbers are placed here for notational convenience. They do not exist

in a PostScript program.

```
1 | 2 3 5 add sub
2 | 3 mul
3 | 3 4 5 6
4 | 5 2 roll
5 | pop
```

In line 1, the numbers 2, 3 and 5 are placed on top of the stack. The operator *add* takes two elements from the top of the stack (in this case, they would be 3 and 5). It then adds the numbers together, and places the result on top of the stack. After executing line 2, there would be two items on the stack—2 and 8, with 8 being on top. Line 2 pushes the number 3 on the stack (so, 3 is on top of the stack, with 2 and 8 underneath). The *mul* operator takes the top two elements, multiplies them together and pushes the result on the stack. After line 2, the stack (from bottom to top) has the following two elements: 2, 24. In line 3, the numbers 3, 4, 5 and 6 are pushed on top of the stack—the stack now has the following (from bottom to top): 2, 24, 3, 4, 5, 6.

The *roll* operator in line 4 takes two elements off the top of the stack, and performs a circular shift on the elements on the stack underneath. The second element it takes off the stack tells it how many elements must be moved, and the first element it takes off indicates how many positions they must be moved. So in line 4, 5 elements on the stack will be moved two positions up (a positive number indicates up, a negative one down). The stack would then look like (from bottom to top): 2, 5, 6, 24, 3, 4 (the underlined numbers are the ones which were moved).

In line 5, *pop* takes the top element (4) off the stack.

In the rest of this discussion, the *operator's arguments* refer to the elements that an operator takes off the stack. The first argument will be the argument which is lowest on the stack, the last argument will be the argument which is highest on the stack. The way in which operators work is as follows. All the arguments are popped off the stack. The operator then performs its functions, which may include pushing or popping elements onto or off the stack, and then pushes any results on the stack (there may be no results). The *currentpoint* refers to the place on the page where the

next object will be drawn.

PostScript allows the definition of procedures (which may make recursive calls), and has a number of powerful iteration and comparison operators. PostScript has a general set of arithmetical, trigonometric, and string handling functions.

Drawing figures

To start drawing on a page, the *newpath* operator is executed. This causes the current point to become undefined. The *setlinewidth* operator can be used to set the thickness of the lines to be drawn, while the *setgray*operator sets the greyness of the lines. The following PostScript program draws a black line from the point (50,50) to (70,70).

```
1.  newpath
2.  50 50 moveto
3.  70 70 lineto
4.  2 setlinewidth
5.  1 setgray
6.  stroke
7.  showpage
```

Line 1 initializes the page. In line 2 the *moveto* operator takes two arguments off the top of the stack (the 50s), and moves the current point to the the position indicated by these two numbers. In line 3, the *lineto* operator takes two arguments off the stop of the stack, and draws a line to that position indicated. The *stroke* operator in line 6, draws all lines etc. which have been defined since the last *stroke* operator. The thickness and darkness of the lines drawn depend on values set within the program. In this program, in line 4, the line width was set to two points, and in line 5, the darkness was set to most dark. If line 3 was replaced with

```
3.  20 20 rlineto
```

the same effect would be achieved. The *rlineto* procedure does a relative *lineto*, that is , it uses the two elements it takes off the top of the stack as an offset from the current position. *rmoveto* has the same relation to *moveto* that *rlineto* has to lineto.

PostScript also allows the drawing of curves. The *curveto* procedure takes 6 arguments—three pairs of points (x_1, y_1), (x_2, y_2), (x_3, y_3).

PostScript then adds a Bézier cube between the current point and the point (x_3,y_3), using the points (x_1, y_1) and (x_2, y_2) as the control points.

When an enclosed object is drawn, its inside can be filled with a colour (or on a monochrome printer, a shade) using the *fill* operator.

Text

PostScript does not have just one stack, it has four. The stack referred to so far is called the *operand* stack—it is the stack which most operators use for operands, and results of operations. The *dictionary* stack is used by the PostScript to keep track of the definition of objects, including procedures. Font definitions are kept in the dictionary stack. To print a string in 12 point Times Roman at position (100, 100) on the page, the following program fragment can be used:

```
1 | /Times-Roman findfont
2 | 12 scalefont
3 | setfont
4 | 100 100 [illegible]veto
5 | (This [illegible]tring) show
```

In line 1, the dictionary stack is searched using the *findfont* operator for the definition of the Times-Roman font. In the next line, the 12 point size is chosen, and in line 3, it is set as the current font. Line 4 moves the current point to the point (100, 100), and in line 5, the string "This is the string" is placed on the page by the *show* operator.

PostScript allows the definition of fonts within programs—so a document need not use only the fonts directly supported by the particular printer.

There are a number of other operators which allow the manipulation of text, including *kshow* (which supports, among other things, kerning), *widthshow* (which can be used to perform calculations on text). Text and other graphics can be intermixed on the page.

Summary

This very brief outline of PostScript shows some its features. How the other stacks can be manipulated and used is beyond the scope of this appendix, as is how bit images can be used and manipulated. How the coordinate system can be used has also been neglected.

Together with PostScript's general purpose operators, the graphics operators provide a powerful framework for page description.

For completeness, a list of most PostScript operators follows.

Operand stack manipulation

pop	exch	dup	copy	index
roll	clear	count	mark	cleartomark
counttomark				

Arithmetic operators

add	div	idiv	mod	mul
sub	abs	neg	ceiling	floor
round	truncate	sqrt	atan	cos
sin	exp	ln	log	rand
srand	rrand			

Array operators

array	[	]	length	get
put	getinterval	putinterval	aload	astore
copy	forall			

Dictionary operators

dict	length	maxlength	begin	end
def	load	store	get	put
known	where	copy	forall	errordict
systemdict	userdict	currentdict	countdictstack	dictstack

String operators

string	length	get	put	getinterval
putinterval	copy	forall	anchorsearch	search
token				

Relational, boolean and bitwise operators

eq	ne	ge	gt	le
lt	and	not	or	xor
true	false	bitshift		

Control operators

exec	if	ifelse	for	repeat
loop	exit	stop	stopped	countexecstack
quit	start			

Graphics operators

gsave	grestore	grestoreall	initgraphics	setlinewidth
currentlinewidth	setlinecap	currentlinecap	setlinejoin	currentlinejoin
setmiterlimit	currentmiterlimit	setdash	currentdash	setflat
currentflat	setgray	currentgray	sethsbcolour	currenthsbcolor
setrbgcolour	currentrgbcolour	setscreen	settransfer	currenttransfer

Coordinate system and matrix operators

matrix	initmatrix	identmatrix	defaultmatrix	currentmatrix
setmatrix	translate	scale	rotate	concat
concatmatrix	transform	dtransform	itransform	idtransform
invertmatrix				

Path construction operators

newpath	currentpoint	moveto	rmoveto	lineto
rlineto	arc	arcn	arcto	curveto
rcurveto	closepath	flattenpath	reverspath	strokepath
charpath	clippath	pathbox	pathforall	initclip
eoclip	clip			

Painting

erasepage	fill	eofill	stroke	image
imagemask				

Character and font operators

definefont	findfont	scalefont	makefont	setfont
currentfont	show	ashow	widthshow	awidthshow
kshow	stringwidth	FontDirectory	StandardEncoding	

Omitted from this list are type attribute and conversion operators, file operators, device setup and output operators, font cache operators, as well as some miscellaneous operators.

C.2. Adobe structuring convention

PostScript was designed by Adobe Systems Inc, which implements all PostScript engines. Adobe has set up a convention [Adobe 1985a; Adobe 1987] for the structuring of PostScript files. Files which conform to this convention are called *conforming*.

The purpose of the convention is to allow system-wide management of resources. By structuring the file in a certain way, other programs—such as document managers and printer servers—are able to meet user requests flexibly and efficiently. Information about the PostScript program and the resources it needs are easily accessible. For example, if the fonts which are to be used are known before a job is printed, a printer server would be better able to choose a printer, and if the choice of the printer was fixed the server would be able to prepare the resources needed, and so be able to use the printer more efficiently. If each of the pages was independent of the

others, then the pages could be reordered by the server. In some cases user requests (such as colour or quality of paper) may need human intervention, and this type of information may also be needed.

The information in this appendix is intended to give an indication of how the convention works, and is not a substitute for the convention specification [Adobe 1987]. The convention works by organizing PostScript programs in a certain way, and using comments, which are easily recognizable to a document manager[1].

Ordinary documents are divided into two parts, a prologue which contains only definitions, and a script which contains only executable code.

The following comments can be embedded into PostScript programs.

The first line of each PostScript program should start with `%!PS-Adobe-X.X`, where the `X.X` is the current version number of the convention.

Other comments are

`%%Title:` This is followed by ASCII text with the title of the document.

`%%Creator:` Followed by the name of the creator

`%%CreationDate:` Followed by text with the date the document was created

`%%For:` Followed by text naming who the document is for.

`%%Routing:` Followed by text explaining where the document should be sent.

`%%Pages:` Followed by a number indicating how many pages there are in the document, and optionally followed by a number indicating in which order the pages should be printed. If no page order is specified, then pages may be ordered in any order. This raises the possibility of pages being printed in parallel—different pages of the same document being printed on different printers at the same time.

[1]This is *post-processing* of a file

`%%Requirements:` Followed by a list of requirements, which include whether the document should be stapled or punched, and whether it should be printed on one side of the page or both.

`%%DocumentFonts:` Followed by a list of all the fonts which will be used by the document. There are also comments which state which fonts the document defines for itself, and which fonts must be provided by the system—these fonts must either be built into the printer, or defined by the printer server when the file is downloaded to the printer.

`%%DocumentNeededFiles:` Followed by a list of files which must be inserted into the document. This allows other PostScript documents to be embedded within the document.

`%%DocumentPaperSizes:` Followed by a list of paper sizes or types which the document needs.

`%%EndProlog` This comment ends the header comments in a file, and marks the beginning of the script.

Within the body of the script, several comments can appear. One of the important decisions which a PostScript generator must make is whether the pages in the text will be independent of each other. If they are to be, then within the processing for each page, no system parameters must be changed: each page can depend on the definitions in the prologue, but may not depend on any processing done on other pages.

If pages are independent of each other, then each page should be preceded by some comments. Some are:

`%%Page:` This is followed by a label which may be printed on the page, and the logical page number which the page is.

`%%PageFonts:` Followed by a list of fonts which will be used on that page. Using this allows the server to determine whether the page can be printed on a certain printer. In a complex document this may also be useful, as if many fonts are being used, this allows the server to determine which fonts will not be used in the processing for the page.

There are many other comments and rules which can be used for

and the logical page number which the page is.

`%%PageFonts:` Followed by a list of fonts which will be used on that page. Using this allows the server to determine whether the page can be printed on a certain printer. In a complex document this may also be useful, as if many fonts are being used, this allows the server to determine which fonts will not be used in the processing for the page.

There are many other comments and rules which can be used for structuring files. Adobe recommend that a comment never be used if the information may be wrong, and secondly that as many comments as possible should be used.

Appendix D. PostScript protocol conversion

This appendix describes the three protocol conversion strategies mentioned in section 5.3.

Notation and assumptions

Let: P be the prologue of the file[1]
D be the data to be printed
D_i be the ith line of the data, where there are n lines in total

The length of the file is Δ=length(D), or $\sum_{i=1}^{n} \text{length}(D_i)$

Define the space ratio of a protocol conversion strategy S given its output file F_S to be

$$\sigma(S) = \frac{\text{length}(F_S)}{\Delta}$$

Assume that there are, on average, 66 characters to a line, and 60 lines to a page.

Analysis

The first strategy—the brute force strategy, B—is after setting up the prologue, to take each line, and enclose it in brackets, with the *show* operator after it to print the line. This is followed by a command to move to the next line. When a page becomes full, the *showpage* instruction is issued to print the current page. An extract from an output file would then look something like this:

[1]The prologue of a file contains both comments, and definition of procedures, constants etc.

```
(D_I) show
30 900 moveto
(D_I+1) show
30 888 moveto
(D_I+2) show
30 866 moveto
```

Here, the Pascal program keeps track of where the current point being drawn on the page is. This strategy implies that there is an overhead of 22 characters for every line. The length of the file would be

$$\Delta + 22n^{\S} + \text{length}(P) + 9\Delta \text{ div } (60*66)^{\ddagger},$$

and the space ratio of this strategy would be

$$1 + \frac{22n + \text{length}(P) + 9\Delta \text{ div } (60*66)}{\Delta}.$$

Substituting $66n$ for Δ, gives us

$$\sigma(B) = 1.33 + \text{length}(P)/\Delta \text{ (rounding to two decimal places)}$$

The next strategy—call it C—offloads some of the work to the LaserWriter. One of the main problems with *B* was that there is a big fixed overhead for each line, for the *show* and *moveto* procedures. *C* defines a Postscript procedure to do this. It is given the name *gl*—short, cryptic names do have their place somewhere. Now, the LaserWriter has the responsibility of keeping track of the currentpoint, and calculate when to move to a new line where necessary. An extract from the output file would look something like:-

```
(D_I) gl
(D_I+1) gl
(D_I+2) gl
```

The overhead per line is now four characters, although the converter now must imbed a procedure with 52 characters in the prologue The length of the file is now equal to

$$\Delta + \text{length}(P) + 4n^{\S} + 9\Delta/(60*66)^{\ddagger}, \text{ and the space ratio}$$

$$\sigma(C) = \frac{\Delta + \text{length}(P) + 4\Delta/66 + 0.0025\Delta}{\Delta}$$

$$= 1.07 + \text{length}(P)/\Delta$$

Comparing the space ratios for *B* and *C*, the following is obtained:

§ Overhead for printing a line and moving to a new line
‡ Overhead for going to new page

$$\frac{\sigma(B) - \sigma(C)}{\sigma(B)} = \frac{0.26 - 52/\Delta}{\sigma(B)}$$

This shows that for any file which has more than 250 characters, method *C* will be more efficient in terms of disk space used, and amount of network traffic. For files larger than 3K, the saving will be between 18 and 20%. These figures are also conservative figures; they relate to pages which are full of text. Where there is more white space, the savings will be greater.

A third strategy would be to offload all the work to the LaserWriter. The prologue would be a Postscript program, which would use the data file as input. The MAT's converter would not manipulate the data file at all. This would all be done by the LaserWriter. The space ratio depends on how sophisticated the algorithm is. There is a fixed overhead which is the same for all files (the length of the program), and there is no other overhead depending on the length of line or page etc. An example program which take a file of characters and produces Postscript output for them can be found in [Adobe 1985b, pp178-181]. This program is 770 characters long. Calling this strategy *P* gives a space ratio of

$$\sigma(P) = \frac{\Delta + 770}{\Delta} = 1 + \frac{770}{\Delta}$$

As far as space is concerned, this only betters the efficiency of *B* for files which are larger than 12K, although for files on infinite length, the space ratio tends to 1 (no overhead at all!).

References

1. ABRAMS M.D. 1985. Observations on Operating a Local-area network. *IEEE Computer* 18(5), May, pp51-66.
2. ADOBE SYSTEMS INC. 1985a. *PostScript Language Reference Manual*. Addison-Wesley, Reading, Massachusetts.
3. ADOBE SYSTEMS INC. 1985b. *PostScript Language Tutorial and Cookbook*. Addison-Wesley, Reading, Massachusetts.
4. ADOBE SYSTEMS INC. 1987. *Adobe Systems Document Structuring Conventions Version 2.0*. Adobe Systems.
5. AGGARWAL, S., SABNANI, K., and GOPINATH, B. 1985. A new File Transfer Protocol. *AT&T Technical Journal*. 64(10), December, pp2387-2411.
6. APPLE COMPUTER INC. (undated). *Inside AppleTalk*. Apple Computer, Cupertino, California.
7. APPLE COMPUTER INC. 1984. *LaserWriter*. Apple Computer, Cupertino, Calfornia.
8. APPLE COMPUTER INC. 1985. *Inside Macintosh: Volume II*. Addison-Wesley, Reading, Massachusetts.
9. ASCHENBERGER, J.R. 1986. Open Systems Interconnection. *IBM Systems Journal* 25(3/4), pp369-379.
10. BARTOLI, P.D. 1983. The Application Layer of the Reference Model of Open Systems Interconnection. *Proceedings of the IEEE* 71(12), December, pp1404-1407.
11. BERGLUND, E.J. 1986. V-System. *IEEE Micro* 6(4), August, pp35-52.
12. BIRRELL, A., LEVIN, A., NEEDHAM, R., and SCHROEDER, M.D. 1982. Grapevine : An Exercise in Distributed Computing. *Communications of the ACM* 25(4), April, pp260-274.
13. BLACK, A.P. 1985. Supporting Distributed Applications: Experience with Eden. In *Proceedings of the 10th Symposium on Operating Systems Principles*, ACM, December, pp181-193. Published as *Operating Systems Review* 19(5).
14. BROWNBRIDGE, D.R., MARSHALL, L.F., and RANDELL, B. 1982. The Newcastle Connection. *Software — Practice and Experience* 12(12), December, pp1147-1162.
15. BOGGS, D.R., SHOCH, J.F., TAFT, E.A. and METCALFE, R.M. 1980. Pup: An Internetwork Architecture. *IEEE Transactions on Communications* COM-28(4), April, pp612-624.

16. BSSRS 1985. *TechnoCop*, FreeAssociation Books, London.

17. CALLON, R. 1983. Internetwork Protocol. *Proceedings of the IEEE* 71(12), December, pp1388-1393.

18. CARLSON, D.E. 1982. Bit-Oriented Data Link Control. In *Computer Network Architectures and Protocols*, P.E. Green (Ed.). Plenum Press, New York, pp111-143.

19. CERF, V.G., and CAIN, E. 1983. The DoD Internet Architecture Model. *Computer Networks* 7(5), October, pp307-318.

20. CHERITON, D.R, and ZWAENEPOEL, W. 1985. Distributed Process Groups in the V Kernel. *ACM Transactions on Computer Systems* 3(2), May, pp77-107.

21. COLE, R. 1987. A Method for Interconnecting Heterogeneous Computer Networks. *Software — Practice and Experience* 17(6), June, pp387-397.

22. CONRAD, J.W. Character-Oriented Link Control. In *Computer Network Architectures and Protocols*, P.E. Green (Ed.). Plenum Press, New York, 1982. pp87-110.

23. DATA COMMUNICATIONS 1986. User problems cloud broadband's future. *Data Communications* 15(9), August, pp47-50.

24. DAY, J.D. 1980. Terminal Protocols. *IEEE Transactions on Communications* [illegible], April 1980, pp585-593.

25. DAY, J.D. and ZIMMERMAN, H. 1983. The OSI Reference Model. *Proceedings of the IEEE* 71(12), December, pp1334-1340.

26. DELLAR, C.N.R. 1983. A File Server for a Network of Low Cost Personal Microcomputers. *Software -- Practice and Experience*. 12, pp1051-1068.

27. DION, J. 1984. The Cambridge File Server. *ACM Operating Systems Review* 14(4), October, pp26-35.

28. [illegible]ONS, W.F. and CHANDLER, A.S. 1983. OSI Session Layer : Services and Protocols. *Proceedings of the IEEE* 71(12), December, pp1379-1460.

29. FORTIER, P.J. 1986. *Design of Distributed Operating Systems : Concepts and Technology*. Intertext/McGraw-Hill, New York.

30. FRY, M.R. 1987. The Transport Layer in a Microcomputer Network. *The Australian Computer Journal* 19(2), May, pp56-62.

31. GIEN, M., and ZIMMERMANN, H. 1979. Design Principles for Network Interconnection. *Proceedings of the 6th Data Communications Symposium*, pp [illegible]-119.

32. GOODWIN, M., and MCDONNELL, [illegible]. Access Control for Network Directory Systems. In *SIGCOMM '86 Symposium: Communication Architectures and Protocols*, ACM August, pp 282-289. Published as *Computer Communications Review* 16(3).

33. HIRSCHEIM, R.A. 1986. The Effect of A Priori Views on the Social Implications of Computing. *ACM Computing Surveys*.18(2), June. pp165-195.

34. HOLLIS, L.L. 1983. OSI Presentation Layer Activities. *Proceedings of the IEEE* 71(12), December, pp1401 1403.

35. HUTTON, J. 1987. Standardization Update. *Computer Networks and ISDN Systems* 13(3), pp171-174.

36. IBM 1984. *Technical Reference Manual: PC Network*. International Business Machines Corporation, Florida.

37. INFOSPHERE 1986. *MacServe: Information Sharing Without the Worry*. Infosphere, Portland, Oregon.

38. JACKSON, W.A., ROGERS, P.C., HEARN, R.J., and MATTIACE J.C. 1986. Performance and Availability in a Network File Server. *IEEE Micro* 6(4), August, pp18-34.

39. JANSON, P., SVOBODOVA, L., and MAEHLE, E. 1983. Filing and Printing Services on a Local-area Network. Research report RZ-1220, IBM Zurich Laboratory, Zurich.

40. KUROSE, J., SCHWARTZ, M., and YEMINI, Y. 1984. Multiple Access Protocols and Time Constrained Communication. *ACM Computing Surveys* 16(1), March, pp43-70.

41. LAM, S.S. 1986. Protocol Conversion—Correctness Problems. In *SIGCOMM '86 Symposium: Communication Architectures & Protocols*, ACM, August, pp19-29. Published as *Computer Communication Review* 16(3))

42. LAMPSON, B.W. 1983. Hints for Computer System Design. In *Proceedings of the 9th ACM Symposium on Operating Systems Principles*, ACM Order No. 534 830, October, pp33-48. Published as *Operating Systems Review* 17(5).

43. LANTZ, K.A. and NOWICKI, W.I. 1984. Structured Graphics for Distributed Systems. *ACM Transactions on Graphics* 3(1), January, pp23-51

44. LANTZ, K.A., NOWICKI, W.I., and THEIMER, M.M. 1984. Factors Affecting tthe Performance of Distributed Systems. *Computer Communications Review* 14(2), pp116-123.

45. LANTZ, K.A., NOWICKI, W.I., and THEIMER, M.M. 1985. An Empirical Study of Distributed Application Performance. *IEEE Transactions on Software Engineering* SE-11(10), October, pp1162-1174.

46. LAZOWSKI, E.D, ZAHORJAN, J., CHERITON, D.R., and ZWAENEPOEL, W. 1986. File Access Performance of Diskless Workstations. *ACM Transactions on Computer Systems* 14(3), August, pp238-268.

47. LEAH, T. 1987. Apple and the Africans. *AppleUser* 7(7), July, pp34-35.

48. LEWAN, D., and LONG, H.G. 1983. The OSI File Service. *Proceedings of the IEEE* 71(12), December, pp1414-1419

49. LININGTON, P.F. 1983. Fundamentals of the Layer Service Definitions and Protocol Specifications. *Proceedings of the IEEE* 71(12), December, pp1341-1345.

50. LININGTON, P.F. 1984. The Virtual Filestore Concept. *Computer Networks* 8(1), pp13-16.

51. LOWE, H. 1983. OSI Virtual Terminal Service. *Proceedings of the IEEE* 71(12), December, pp1408-1413.

52. MACHANICK, P. 1987a. Programming in PostScript. *MacNews* 1(1), January, pp5-8, p29.

53. MACHANICK, P. 1987b. Programming in PostScript Part2. *MacNews* 1(2), February, pp6-10, p30.

54. MACHANICK, P. 1987c. Programming in PostScript Part 3. *MacNews* 1(3), March, pp25-29.

55. MACHANICK, P. 1987d. Programming in PostScript Part 4. *MacNews* 1(4), April, pp23-28.

56. MACHANICK, P. 1987e. Programming in PostScript Part 5. *MacNews* 1(5), May, pp13-17.

57. MACHANICK, P. 1987f. Programming in PostScript Part 6. *MacNews* 1(6), June, pp18-22.

58. MACHANICK, P. 1987g. Programming in PostScript Part 7. *MacNews* 1(7), July, pp3-6.

59. MACHANICK, P. 1987h. Programming in PostScript Part 8. *MacNews* 1(8), August, pp19-21.

60. MANOIM, I. 1986. "The case for the 'alternative' press." *Weekly Mail* 2(22), June 6-12 , p10.

61. MAYNE, A.J. 1986. *Linked Local Area Networks*. (2nd ed) Wiley, New York.

62. MITCHELL, J.G. and DION, J. A. 1982. Comparison of Two Network-Based File Servers. *Communications of the ACM* 25(4), April, pp233-245.

63. MORRIS, J.H., SATYANARAYANAN, M., CONNER, M.H., HOWARD, J.H., ROSENTHAL, D.S.H., and SMITH, F.D. 1986 Andrew : A Distributed Personal Computing Environment. *Communications of the ACM* 29(3), March, pp184-201.

64. MUIR, S., HUTCHISON, D., and SHEPHERD, D. 1985.Arca : A Local Network File Server. *The Computer Journal* 28(3), July, pp243-249.

65. MULLENDER, S. and TANENBAUM, A.S. 1985. A Distributed File Service Based on Optimistic Concurrency Control. In *Proceedings of the 10th Symposium on Operating Principles*, ACM, December, pp51-62. Published as *Operating Systems Review* 19(5).

66. NEEDHAM, R.M., and HERBERT, A.J. 1982. *The Cambridge Distributed Computing System*. Addison-Wesley, London.

67. NOTKIN, D., HUTCHINSON, N, SANISLO, J., AND SCHWARTZ, M. 1987. Hetrogeneous Computing Environments: Report on the ACM SIGOPS Workshop on Accommodating Heterogeneity. *Communications of the ACM* 30(2), February, pp132-140.

68. PETERSON, J. and SILBERSCHATZ, A. 1985. *Operating Systems Concepts* 2nd edition. Addison-Wesley, Reading, Massachusetts.

69. POPESCU-ZELETIN, R. 1984. Some critical considerations in the ISO/OSI RM from a network implementation point of view. *Computer Communication Review* 14(2), pp188-194.

70. POSTEL, J.B. 1980. Internetwork Protocol Approaches. *IEEE Transactions on Communications*. COM-28(4), April, pp604-611.

71. QUATERMAN, J.S. and HOSKINS, J.C. 1986. Notable Computer Networks. *Communications of the ACM* 29(10), October, pp932-971.

72. RICHARDSON, M.F., and NEEDHAM, R.M. 1983. The Tripos filing machine, a front end to a File Server. In *Proceedings of the 9th ACM Symposium on Operating Systems Principles*, ACM Order No. 534 830, October, pp120-128. Published as *Operating Systems Review* 17(5).

73. ROOS, J. 1987. *Current Standards in LAN Management*. Local Area Networks in South Africa Conference, Sandton Sun Hotel, March 24-25.

74. ROSENBERG, R. 1987. Privacy in the Computer Age. *The CPSR Newsletter* 5(1), Winter, p1ff.

75. SARACCO, R., and TILANUS, P.A.J. 1987. CCITT SDL: Overview of the Language and its Applications. *Computer Networks and ISDN Systems* 13(2), pp65-74.

76. SHELTZER, A.B., and POPEK, G.J. 1986. Internet Locus: Extending Transparency to an Internet Environment. *IEEE Transactions on Software Engineering* SE-12(11), November, pp1067-1075.

77. SIDHU, G.S. and OPPENHEIMER, A.B. 1985. AppleTalk Printer Access Protocol. In *Inside LaserWriter*, Apple Computer Inc.

78. SPECS CONSORTIUM and BRUIJNING, J. 1987. Evaluation and Integration of Sepcification Languages. *Computer Networks and ISDN Systems* 13(2), pp75-89.

79. SPROULL, R.F., and COHEN, D. 1978. High-level Protocols. *Proceedings of the IEEE* 66(11), November, pp1371-1386.

80. SRINIVASAN, B. and ANANDA, A.L. 1986. A Simple File Server for A Terminal Support Network Environment. *Computer Networks and ISDN Systems* 11, pp383-389

81. STALLINGS, W. 1984. Local Networks. *ACM Computing Surveys* 16(1), March, pp3-41.

82. SVOBODOVA, L. 1984. File Servers for Network-Based *Distributed* Systems. *ACM Computing Surveys* 16(4), December, pp353-398.

83. TANENBAUM, A.S. 1981. *Computer Networks*. Prentice-Hall, New Jersey.

84. TANENBAUM, A.S., and VAN RENESSE, R. 1985. Distributed Operating Systems. *ACM Computing Surveys* 17(4), December, pp419-470.

85. TANGENT TECHNOLOGIES. 1987. *PC Macbridge: PC AppleTalk Driver Version 3.00 Software Interface Reference Manual*. Tangent Technologies, Ltd., Norcross, Georgia.

86. THEIMER, M.M., LANTZ, K., and CHERITON, D.K. 1985. Preemptable Remote Execution Facilities for the V-system. In *Proceedings of the 10th Symposium on Operating Principles*, ACM, December, pp2-12. Published as *Operating Systems Review* 19(5).

87. TRIPATHI, S.K., YENNUN HUANG, and JAJODIA, S. 1987. Local Area Networks: Soft ware and Related Issues. *IEEE Transactions on Software Engineering* SE-13(8), August, pp872-879.

88. VOYDOCK, V.L., AND KENT, S.T. 1985. Security in High-level Network Protocols. *IEEE Communications Magazine* 23(7), July, pp12-24.

89. WALKER, B.J. and POPEK, G. (undated). *The Locus Distributed File System*.

90. WALKER, B.J., POPEK, G., ENGLISH, R., KLINE, C., and THIEL, G. 1983. The Locus Distributed Operating System. In *Proceedings of the 9th ACM Symposium on Operating Systems Principles*, ACM Order No. 534 830, October , pp49-70. Published as *Operating Systems Review* 17(5).

91. WATSON, R.W. and MAMRAK, S.A. 1987. Gaining Efficiency in Transport Services by Appropriate Design and Implementation Choices. *ACM Transactions on Computer Systems* 5(2), May, pp97-120.

92. WEINSTEIN, M.J., PAGE, T.W., LIVEZEY, B.K., and POPEK, G.J. 1985. Transactions and Synchronization in a Distributed Operating System. In *Proceedings of the 10th Symposium on Operating Principles*, ACM, December, pp115-126. Published as *Operating Systems Review* 19(5).

93. ZIMMERMAN, H. 1982. A Standard Layer Model. In *Computer Network Architectures and Protocols*, P.E. Green (Ed.). Plenum Press, New York.