The Design of Protocols for High Performance in a Networked Computing Environment

by

Gary D. Law

Doctor of Philosophy

Department of Computer Science

University of Edinburgh

1989

Abstract

Technological advances in both local area networks and computer processor design have led to multiple computer installations being composed of a much wider range of network devices than previously possible. High bandwidth computer networks may now interconnect large numbers of devices that have different processor architectures and instruction sets, as well as various levels of performance. This thesis is concerned with the merits of such networks and addresses the problem of how the many different types of computers may be integrated to form a unified system.

A review of a number of approaches towards the formation of multiple computer systems includes campus computer networks, configurations of mainframes and examples of distributed computer systems. This study provides an insight into the fundamental principles of this field. The key features of the systems considered in the study are grouped together in a description of a general network structure. Subsequently, the network devices in this structure are classified into three groups, according to their rôles and communication requirements.

The three-way classification of devices leads to the development of a Triadic Network Model to describe the interactions within and between the three groups. The model's specification of network communication provides the basis for protocols that are well suited to the needs of this computing environment. The thesis covers the principles of the protocols and the details of their implementation in an experimental system. The software tools developed to support the implementation are also described.

Acknowledgements

The author acknowledges the financial support of the SERC for the period of his research studentship at Manchester and Edinburgh universities. The author gratefully recognises the assistance of Spider Systems Limited, in enabling him to have time in which to complete the production of this thesis. The author wishes to express his appreciation to his supervisor, Professor Roland Ibbett, for his support and encouragement throughout the course of this work. Further gratitude is due to Professor Ibbett, in his capacity as the Head of the Computer Science Department, for permitting access to the resources needed to undertake this work.

The contact and discussions with colleagues at both Manchester and Edinburgh universities have proven to be of great benefit. In particular, the help of Dr. Nigel Topham and Tim Hopkins should be singled out in this respect. Without the strength and support of family and friends, this thesis would never have been completed. The author wishes to thank the *'crew'* in Manchester, and Richard, John and Kamran, for those memorable *'seshes'*. Special thanks are due to Carl and Andrew for their help and friendship throughout that painful transition, and beyond :

"you gave me something that I won't forget too soon".

The author's deepest gratitude is reserved for three very special people : his wife Lesley, for her patience throughout the many long hours, and his parents, for their never ending support. This work is dedicated to them, for :

"there is a light that never goes out".

Preface

The author graduated with B.Sc. (hons.) in Computer Engineering at the University of Manchester in July 1984. In October 1984, he joined the MU6-V research team in the Department of Computer Science at the University of Manchester. In October 1985, he transferred to the University of Edinburgh, accompanying his supervisor, Professor R.N.Ibbett. The major part of the research described in this thesis has been undertaken in the Department of Computer Science at the University of Edinburgh. Since January 1988, the author has been employed by Spider Systems Limited, where he is currently engaged as a Project Leader in the Network Management Business Centre.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Research Overview

This thesis is concerned with the integration of computers, with differing characteristics, to form a single system. The level of cooperation that may be achieved within multiple computer systems has been the source of significant research effort in recent years, and many *distributed computing systems* have been produced. The work described here is not an attempt to design another distributed computer system, although it is considered that this research could lead to the development of such a system. Rather, this thesis investigates the possibilities afforded by the interconnection of the wide variety of computers available today, and how best to unify a system composed of a number of such machines.

1.1.1 Outline of Thesis

This chapter continues with an outline of the basis for the research described in this thesis, and an example is given to illustrate the direction of this research. The subsequent chapter reviews various approaches towards forming multiple computer

systems, including campus computer networks, configurations of mainframes and distributed computing systems.

The following chapter opens with a description of a network structure considered to be representative of configurations that will be prevalent in the near future. The chapter proceeds with the definition of a triadic network model to represent the rôles of nodes in a distributed system and to determine the nature of the communication that will exist in this environment. The concepts developed in this model form the foundation for the work detailed in the succeeding chapters.

Chapter 4 explains the principles of a protocol set derived from the model for use in a heterogeneous multiple computer system. The fifth chapter is concerned with the practicalities of an implementation of the network protocols. Finally, the concluding chapter indicates the current state of the work and presents some pointers for future research that may be conducted using the results of this thesis.

1.1.2 Background

There is a significant amount of attention being paid towards distributed computing systems. Before outlining the direction of the research work in this thesis, it is worth considering some of the changes that have influenced the focus of research into distributed computing.

Decentralization

The trend towards decentralization of resources is apparent throughout the history of computing. The early computers were large machines dedicated to the

processing of batch jobs entered via punched cards and providing results on directly attached lineprinters. All tasks associated with the computational task occurred at a single location - the execution of the program, mass storage of data, input and output of information. The first stage of decentralization came with the use of Remote Job Entry (RJE) stations, allowing input and output at sites distant from the central mainframe. These RJE stations were somewhat restricted in their capabilities, but they did serve to offload much of the input/output-limited processing associated with each job. However, most of the resources still had to be situated close to the central computer.

The introduction of timesharing systems significantly broadened the accessibility of the computer for the users, allowing the input and output of information to be distributed to where it was neeeded. The initial effect was that existing packages were made easier to use, but soon a wealth of new software appeared, written to exploit the advantages of timeshared systems. The ability to allow a number of users to share on-line access to large quantities of data opened the door to many new applications.

Perhaps the next prominent step towards the distribution of processing came with the arrival of multi-computer installations, where communication between the machines was achieved through the use of special purpose, vendor dependent interfaces. The fact that these interfaces were specific to the machines of individual manufacturers is significant because it restricted the flexibility of interconnection that could be achieved with each computer. The sharing of processing between the different computers was, in general, effected by the user or application soft-

ware, and was limited to large programs with no interprocessor communication requirements.

The advent of local area networks greatly expanded the range of configuration strategies possible in multi-computer systems. This led to much greater distribution of processing and control in networked systems, but still tended to result in the computers themselves being centrally sited. The development of personal computers has now allowed processing to move from the traditional central computing centre to the locations where the personnel need it most. This more localized approach has meant that it is now viable for the whole of a department's computing resource to be distributed wherever it is needed.

Technological Influences

Continued developments in the design and production of very large scale integrated (VLSI) devices has meant that much greater computing power is provided by the processors of small computers. The functionality of VLSI devices has also increased, enabling software that was formally restricted to minicomputers to become accessible to personal computer users.

Advances in storage device technology mean that a much greater density of data can be stored by these small machines. This has resulted in the porting of "memory hungry" applications software from much larger machines, and has allowed useful extensions to be made to existing packages for small computers.

The enhanced accessibility of powerful computing resources has been accompanied by the development of both software and hardware to provide a more versatile and amenable interface between human and computer. This has in turn resulted

in these personal computers being capable of supporting sophisticated applications, previously restricted to mainframe users, with greatly enhanced interaction capabilities.

'Proliferation' of Workstations

As the facilities of small computers have continued to expand, so the costs of hardware have fallen. The effect of this is that it is possible to purchase a desk-top machine for less that £5000 with the processing power of a VAX 11/780. These machines are being called *personal workstations*, which aptly describes their combination of computing power and flexible user interface.

The amalgamation of personal workstations and high bandwidth local area networks may be the first opportunity to satisfy the proponents of decentralization. With the power of yesterday's room-sized mainframes in today's desk-top microcomputers, it may appear that the need for centralized computing resources has disappeared. However, the case for retaining certain forms of shareable processing resource is still very strong. The merits arise from the fact that, with technological advances and reduction in hardware costs, it has become practical to develop special purpose processors, dedicated to forms of processing that are only occasionally required by any one user, and to attach these machines to a network to permit their use to be shared amongst the users of the system.

1.1.3 Direction of Research

The ultimate goal of this research is a multiple computer system composed of greatly dissimilar machines, all interacting in a well organised manner to cooperate

in the provision of a powerful and versatile computing service to the users of the system. It is anticipated that this target system will comprise both small scale general purpose computers and specialised high performance processors, with all nodes interconnected by a local area network. The smaller machines correspond to *personal workstations*, with processing power suitable for most of the user's needs. The more powerful computers supplement the workstations by providing specific computational functions, using architectures tailored to those algorithms, and thereby supplying high levels of efficiency for the appropriate functions.

The work that is the subject of this thesis is concerned with the design of communication protocols to support directly the interactions that predominate in the environment outlined above. The objective is to produce network protocols oriented towards the target system for an optimum response.

The following section describes a computer system that serves to illustrate the target system outlined here. The philosophy of the system is described and an example of a special purpose processor is given. The requirements of a network connection of this processor are considered, together with some of the detail of a high bandwidth local area network that ideally satisfies many of the needs of a distributed system.

1.2 Case Study

1.2.1 The MU6 System

Philosophy

The philosophy behind the MU6 system was formed at the end of the seventies by a group of academics in the Computer Science department at Manchester University. The overall structure of the system, illustrated in figure 1–1, was conceived as having three *ranks* of computer, corresponding to the differing levels of processing power and the contrasting rôles of the machines involved.

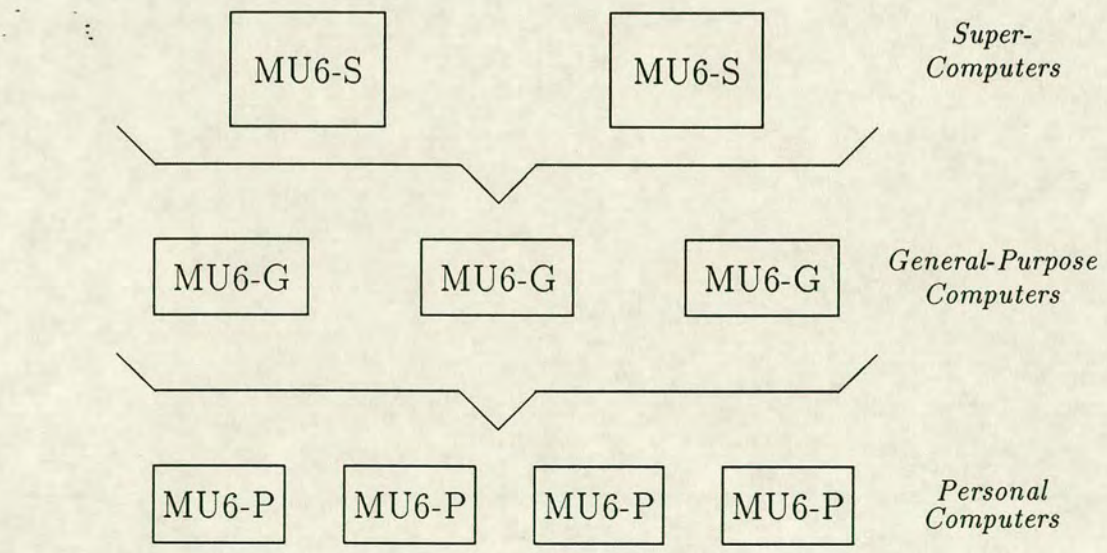

Figure 1–1: The MU6 System

The lowest of the ranks was occupied by a large number of MU6-**P** machines. These were to be the **personal** computers, with screen, keyboard and some degree of local processing power to enable interactive editing, and functions with

similar requirements, to be provided. The MU6-P's would make use of MU6-**G**, **general-purpose**, computers for any tasks requiring greater computing power. In addition to flexible general-purpose time-shared processing capabilities, the MU6-G machines would provide access to peripherals and mass storage, as well as acting as front-end processors for the higher level machines.

The uppermost rank in this hierarchy of computers would be the MU6-**S**, **super computers**. Unlike the MU6-P and MU6-G machines, MU6-S computers would, most probably, have dissimilar architectures. Their rôle in the MU6 system is to provide high levels of performance in those applications for which they were designed.

Each rank in the MU6 system reflects the degree of common utilization, the quantity and cost of the computers at that level. So, the MU6-P machines would be inexpensive, generally useful and very accessible because of their provision in large quantities. In contrast, the MU6-S computers would be much more costly, with limited areas of applicability and consequently would be present in small numbers only.

The MU6 system, as described here, was never fully realised, but the following section describes the developments that were made in the partial implementation of the system.

Implementation

The first computer in the MU6 system to be designed and subsequently constructed was the MU6-G computer [29,71]. MU6-G had an architecture and order code designed specifically to be appropriate to the compilation and execution

of high-level languages and operating systems. The 32-bit word size and 64-bit floating-point arithmetic unit made it useful for scientific computations. It had a performance of about 2 MIPS that was readily predictable due to the machine's simple architecture. A three stage pipeline was used, but ECL 10k series technology was required to produce the required throughput.

The prototype MU6-G was primarily constructed using AUGAT wirewrap boards, mounted eight to a panel on a four sided column. It was air-cooled by fans at the top of each side of the column. A PDP-11/34 acted as a front-end processor for MU6-G to control access to peripherals [92].

Conventional terminal lines were the means of access to MU6-G by other computers, including the MU6-P machines, MU6-V (q.v.) and VAXs that were used for operating system development. The MU6-P machines were 68000 based microcomputers with graphic capabilities and floppy discs for removable storage. They ran the same operating system as MU6-G, namely the Manchester University System Software (MUSS) [31,74,79].

The file system of MUSS incorporated the ability to open a directory that was in fact resident on a remote machine. This provided the means by which files could be transferred to and from MU6-G, and represented the limited degree of cooperation that was achieved between MU6-Ps and MU6-G.

There were no MU6-S processors provided as a service in the MU6 system, but a prototype vector processor was constructed and this communicated with MU6-G via a terminal line for the downloading of programs and the return of results - both operations being under the direct control of the operator. This machine, MU6-V, is described in the following section.

1.2.2 MU6-V

MU6-V is a parallel vector processing system capable of achieving a high performance by enabling a number of vector processors to cooperate in the solving of a single problem[42,103,104].

Architecture

A primary objective of the MU6-V design was that the set of vector processors should have a wide applicability to many different algorithms, rather than being restricted to a much smaller range of functions due to its topology. For this reason, the structure is formed from a linear array of processors capable of both scalar and vector operations, each with local memory and all interconnected by a common communication medium, figure 1–2. Communication between processors occurs when the value of a global variable is to be updated, and is achieved through the use of a global broadcast by the processor effecting the change. The processor places a value/identifier pair on the common highway, and all other processors holding a local copy of this variable accept this new value and update their copy.

Addressing Mechanism

The identifiers used for the variables are virtual addresses, with the name of a vector being analogous to the segment number in a conventional virtual storage system. So, within each processing unit there exists a virtual-to-real address translation mechanism.

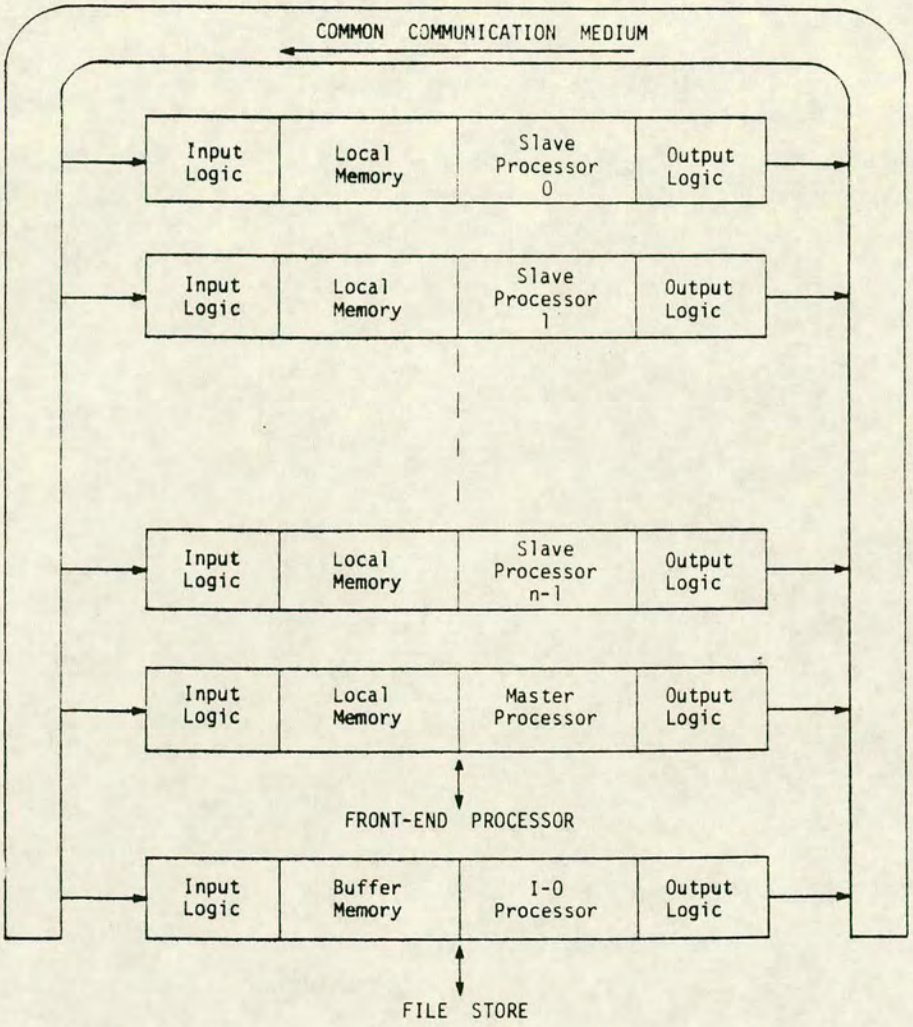

Figure 1–2: MU6-V System Organization

The vector names are used to index into a table of descriptors, each containing an origin address, size and type information. The latter describes the form of the vector : dense or sparse, directly or indirectly accessed. Hence, the use of generic instructions is possible since the hardware may determine the means of accessing the vector from the type information.

Synchronization

To ensure an orderly flow of data between producers and consumers, a single synchronization bit is provided for every word of vector memory to indicate the validity of the vector element. *Writing* to an element will *set* the bit, whilst *reading* the element may *reset* the synchronization bit, depending on the instruction. When the bit is set, the local unit may read or write to that element, but other vector units are unable to overwrite the data. If the bit is reset then the local unit may not read data from the element, but all units are able to write data into the element.

Dedicated Processor Units

From the system organisation of MU6-V, figure 1–2, it can be seen that, in addition to the many "slave" processor units, there are two special units : the master processor and the input/output processor unit. The former performs initialisation before initiating parallel sub-processes and may act as the "harness" process required by many programs. The master process might test for termination or convergence of sub-processes and initiate further sub-processes.

The rôle of the input/output processor unit is of special interest because it determines the manner in which MU6-V is used in a multiple computer system.

Rôle of MU6-V in the MU6 System

MU6-V was intended for use in the MU6 system as one of the MU6-S super computers. In this capacity, MU6-G would act as its *front-end*, performing scheduling

and similar organisational tasks on behalf of prospective MU6-V users, leaving MU6-V to concentrate purely on vector calculations.

In the MU6 system, MU6-V may be considered as a special type of resource providing a processing service to be shared amongst the users of the system. This "MU6-V service" allows any user to perform vector calculations, with a high level of performance, by supplying the necessary code and source data and receiving, in return, the results of the computation. The provision of this "MU6-V service" within the system brings a number of points to mind.

As has been stated already, MU6-V does not have an architecture that is suitable for running a conventional operating system. Instead, MU6-V has an input/output processor that communicates closely with another machine, its front-end, which in turn is responsible for high level functions. Hence, potential users of the MU6-V service are unable to interact directly with MU6-V, but must do so via the front-end machine.

Use of the MU6-V service requires the transfer of large quantities of data, but the question arises as to where this data should normally reside. The user's machine will require access to the data for editing the program, producing the source data and analysing the results. MU6-V needs the code and the source data in order to execute the program, and has to then store the results. It is also possible that either the source or the result data could be required by machines other than those already mentioned.

The review of distributed systems in the next chapter will examine storage considerations further, but for the present the configuration in figure 1–3 is sufficient for a look at the input/output processor unit in the following section. This

diagram shows the user's machine, MU6-V and its front-end to be separate from a shared fileserver. The fileserver may well be part of one of the other three machines, but referring to it as a separate machine more fully illustrates the function of the input/output processor.

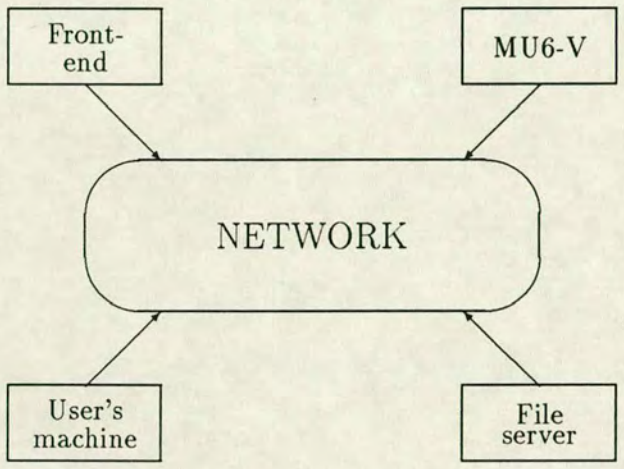

Figure 1–3: Machines Involved in Use of MU6-V

Requirements of the Network Connection

The architecture of MU6-V is unsuitable for the functions that are usually associated with the operating system. So within the MU6 system, MU6-V is treated as a *processing resource*, with its primary operating system requirement being the ability to communicate with its front-end. The communication between MU6-V and its front-end would be concerned with the initiation, monitoring and control, and termination of programs. All of this inter-machine communication would be performed by the input/output processor of MU6-V.

The I/O-processor's responsibilities extend beyond the simple exchange of short messages to govern the execution of programs by MU6-V. The processor must also effect the transfer of large amounts of code and data between MU6-V and the fileserver. The duties of the I/O-processor may be better explained by considering its actions in the execution of a typical program.

1. The front-end sends a directive to instruct the I/O-processor that a job is awaiting execution.

2. The front-end sends a job description that includes file identifiers for use in accessing the program, source data and for storage of the results.

3. The I/O-processor uses these file identifiers to request the code and data needed from the fileserver for execution of the program.

4. The fileserver transfers first the code and then the data to MU6-V.

5. The I/O-processor loads each section of code into the corresponding processor unit, via the common communication medium, and then does likewise with the source data. The relationship between the processor units of MU6-V and the blocks of code and data may be determined from the initial job description supplied by the front-end.

6. Once all of this code and data transfer is complete, the input/output processor initiates the master process, which, in turn, will start up the subprocesses.

7. As new values for variables are transmitted over the common highway by the slave processor units, the I/O-processor accepts the values of those variables it knows to be result vector elements. It forms these elements into whole vectors and transfers them to the fileserver using the file identifier already supplied.

8. On termination of the program, I/O-processor ensures that all result vectors have been transferred to the results file and then informs the front-end of the conclusion of the job.

The input/output processor unit may also be required to assist the front-end in the debugging of programs under development and analysis in the event of abnormal termination due to an error condition.

It is apparent that the I/O processor plays an important rôle in the operation of MU6-V, but the requirements of the network connection should not be overlooked. Since the main MU6-V processor units are dedicated to the application software, the I/O processor has to satisfy the network protocols at all levels, from the physical nature of the connection to the higher level application-oriented conventions.

The next section describes a high bandwidth local area network that is ideally suited to the needs of distributed systems. This is because of the relatively good quality of service provided by its low level protocols and also due to the inherent network intelligence present in the network.

1.2.3 Centrenet

The development of **Centrenet** [22,36,41], in the Department of Computer Science at the University of Manchester, began at the start of the 1980s. Its design objectives were :

> "to satisfy the requirements of both closely knit multi-computer systems and communities of users spread across large campus areas ... who wish to transfer files between their own machines and the central site, to gain access to a variety of systems and to share a variety of hardware and software resources."[41]

To achieve these aims, Centrenet has a high bandwidth and possesses a form of "network intelligence". This intelligence is provided by a collection of processors, each of which is directly attached to a switching node.

Structure

Centrenet is formed from high speed switching nodes, designated **Starpoints**, connected by links that carry packets from source to destination, according to a 4-bit address. The address fields of a Centrenet packet, figure 1–4, have 16 bits, so the overall structure of the network is that of a four-level singly-linked tree. Figure 1–5 shows the hierarchical nature of Centrenet and indicates how the routing is acheived by decoding *nibbles* of the destination address field.

To understand the routing mechanism, consider a starpoint at level 3 in the hierarchy, such as node '3a' in figure 1–5. If the upper 12 bits of the address match

DESTINATION ADDRESS	SOURCE ADDRESS	DATA	CONTROL
16	16	32	8

Figure 1–4: Centrenet Packet

those of the node, then the lower 4 bits indicate one of the ports on that starpoint. However, if the uppermost portion of the address is different, then the packet is directed through an **uplink** to the node at the higher level. At this point, bits 4 to 7 determine the **downlink** [1] to be used, assuming the upper byte of the address corresponds to the node.

The simple nature of this routing mechanism means that the starpoints may be implemented by high speed logic, rather than needing processors like in the ARPA network [98]. On the other hand, Centrenet does not possess a means of providing alternative routes, unlike the ARPA network.

A characteristic of the Centrenet topology is that the total available bandwidth is distributed throughout the network. So, ports communicating across one Starpoint will not detract from the bandwidth available for ports of another Starpoint.

Starpoints

The relatively small size of a Centrenet packet (72 bits) makes it possible for parallel interconnections between ports in a single Starpoint. The implementation

[1] A downlink appears to the Starpoint as an ordinary port.

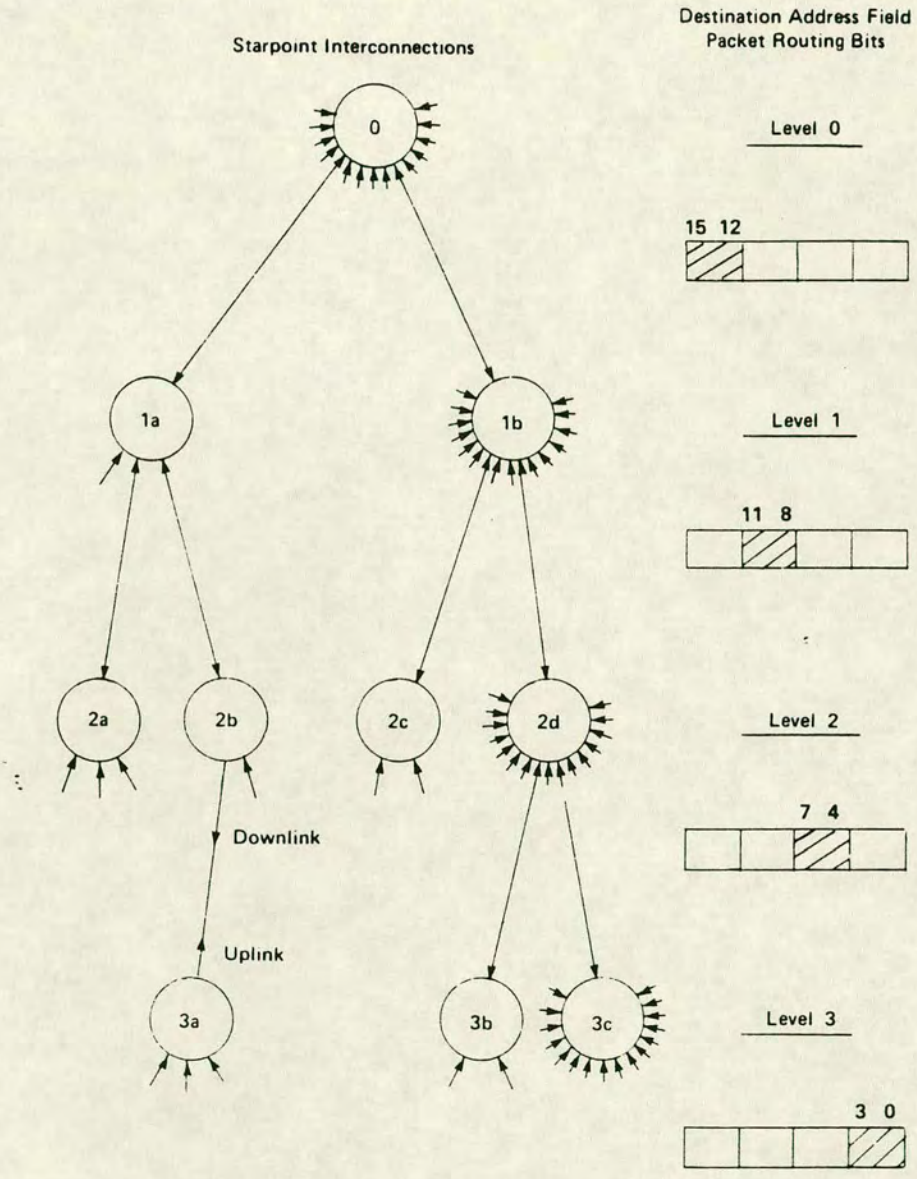

Figure 1–5: The Centrenet Hierarchical Structure

of the Starpoints is based on an LSTTL backplane interconnection bus that has a throughput in excess of 160 Mbps of user data. Each Starpoint has a capacity for 16 **Port Cards**, which may connect to devices or downlinks, and one **Uplink**

Card[2]. This results in a maximum capacity for the network of 65,536 devices. The devices communicate via 72-bit input and output buffers that are selectively polled by the Starpoint controller card, figure 1–6.

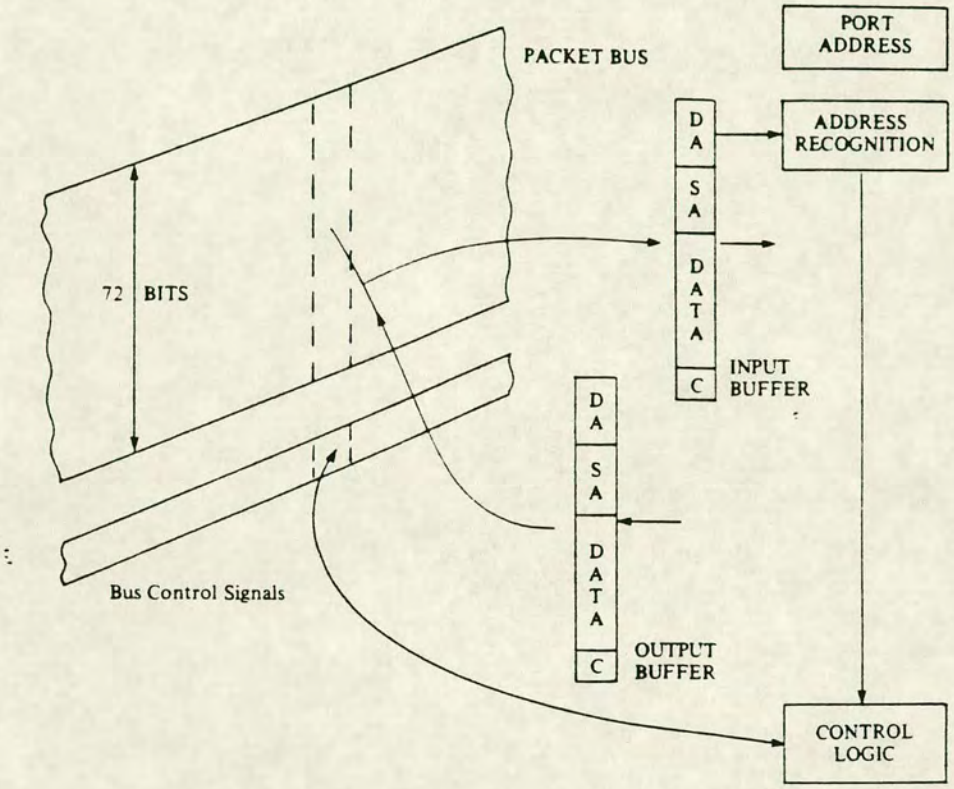

Figure 1–6: Starpoint Bus Interface

To illustrate the mechanisms involved in the transfer of a packet from one port to another on the same Starpoint, consider the following sequence :

1. The source of the data loads the packet into the Output Buffer Register.

2. When the card is polled the packet is transmitted onto the Starpoint bus by enabling the Output Register.

---

[2]The *root* does not have an uplink.

3. All other cards in the Starpoint inspect the new packet and check the destination address against their own address. In the case of the port cards, they will test for equivalence in the range of bits under consideration (see section 1.2.3). However, an uplink card will look for a mis-match in the upper portion of the address field. It does not inspect the 4 least significant bits of the portion of the address under consideration.

4. The port which determines that the packet is directed towards it will latch the incoming packet into its Input Buffer Register, ready for acceptance by the attached device.

5. The receiver returns an acknowledgement to the transmitter to indicate that the packet has been accepted.

If the transmitter were to receive no acknowledgement within the timeout period, it would re-transmit the packet on each subsequent poll, up to the re-try limit. This positive-only acknowledgement scheme allows a simplified protocol to be used, but without unnecessary data loss. The re-try limit ensures that the failure of a port does not cause the network to become blocked.

If a transmitter reaches its re-try limit, it notifies its attached device of the problem and activates a fault report signal on the bus when it is next polled. This signal will inform the local **Network Intelligence Module** of the fault.

Network Intelligence

Every Starpoint may contain one **Network Intelligence Module** (NIM) but, where a NIM is not connected, a bus controller board will provide its low level

functions. These functions are fundamental to the operation of the Starpoint, and include the provision of the polling and clock signals for the ports. Additionally, the NIM may collect statistics about the behaviour of the Starpoint and could adapt the polling sequence to give certain ports a greater proportion of the bandwidth available in the Starpoint.

Within each Starpoint, the NIM is responsible for initializing the ports and attached devices on power-up, receiving fault reports and performing error recovery when necessary. Initially it was considered that the NIM should have control over terminal connections, but this power has been devolved to the **Terminal Multiplexer**[82].

The NIM has an equally important rôle to play in the higher level operation of the network. The NIM may perform virtual circuit connections and disconnections on behalf of other nodes, and other high level functions, such as name serving. Hence it would be possible for nodes to communicate using *virtual addresses* or *names* rather than needing to know physical addresses.

Furthermore, the use of *generic names* would be possible, so that a node could send to the NIM a request for a virtual circuit connection to a certain *type* of device (such as, say, a printer) rather than specifying a particular physical address. The NIM would then search its internal name tables for the specified service and endeavour to establish a virtual circuit with a device providing that service. Finally, the NIM would return an acknowledgement to the original node, indicating the physical address to be used.

The full potential of the NIM has not been exploited because its original implementation, based on a Z80 [36], had insufficient processing power. The recent

development of a 68000 based NIM [96] should enable an extensive investigation of the versatility of the NIM and its support for *distributed network services* [35].

Links

The Starpoints are interconnected by single links that may be of a variety of types. The current implementations use either 2 co-axial cables, one for bi-directional data and the other for clocks, or 2 optical fibres, one for each direction. The essentially asynchronous nature of Centrenet, together with the distributed bandwidth, means that it is possible to use different technologies for the constituent parts of the network. Hence, a collection of Starpoints, some connected by co-axial cables and others by optical fibres, may form a single network.

Taking things even further, it would be possible to produce a microwave link to allow Starpoints, separated by kilometres, to interact. Also these Starpoints may be implemented using different technologies. The versatility of Centrenet means that it may be readily adapted to suit the needs of a particular installation, and subsequently enhanced as the configuration expands.

Network Protocols

The Centrenet protocols have been designed to accommodate a variety of different types of communication and achieve this, in part, by allowing a number of packet types, distinguished by the bits in the control field. The packet types fall into two categories :

Structured Protocols : These provide end-to-end communication of words and characters between computers, terminals or each other[106].

Raw Mode Protocols : These are for digitized sound [37] or vision [38], for which the network latency and throughput are more important than the quality of the connection; the occasional loss or re-ordering of packets is much less critical for these applications than for computer data transfer.

There are currently two structured protocols :

1. The **Terminal Protocol**, used by the Terminal Multiplexer [82], enables the transmission of 8-bit data items across the network, with end-to-end acknowledgement. It has two elements :
   - the **Connection Protocol** for virtual circuit management
   - the **Character Protocol** for the data transmission.
2. The **Burst Protocol** allows a block of between 1 and 64K bytes to be transferred across the network, with an acknowledgement returned at the end of the transfer.

Of all these protocols, the Burst Protocol [24] is most appropriate to this discussion because most of the inter-machine communication is likely to use this for performance reasons. The form of the protocol is illustrated in figure 1–7.

The **Burst Header** is a single packet that gives the number of bytes in the burst. This number excludes any *padding* bytes and the checksum. The **Burst Header Ack** returns the *accepted length* for the data, which will usually be the

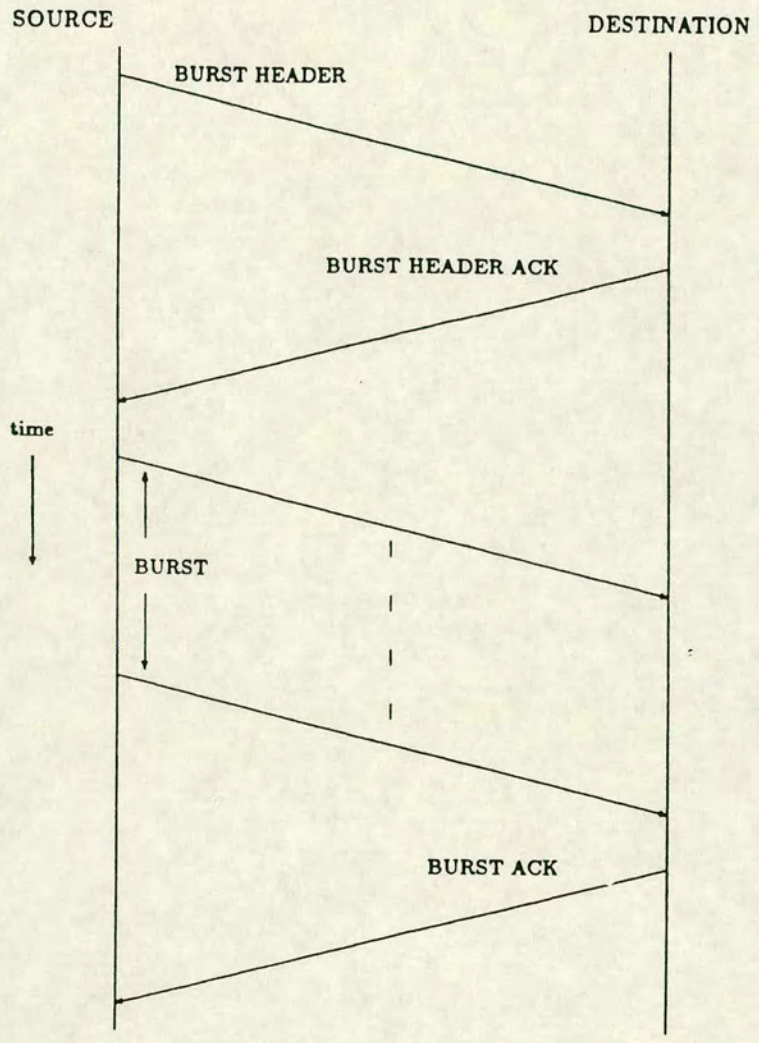

Figure 1–7: The Burst Protocol

same as that requested in the **Burst Header** but may be less if the receiver has insufficient capacity for the entire burst. The data transmission proceeds in the same manner, but no more than the *accepted length* is sent.

The **Burst Data** comprises the data to be transferred, together with any zero-filled bytes necessary to bring the total number of bytes to $4n + 2$. The final packet contains the last two bytes and a 16-bit checksum. To conclude, the receiver returns a **Burst Ack** to the transmitter.

Applications

The development of a gateway between Centrenet and Ethernet [34] was a necessary step towards enabling the integration of a wide selection of resources into a single network. Centrenet may now be used to bridge disparate Ethernets, and enable Ethernet based devices to access resources connected directly to Centrenet.

The unification of the two networks permits the benefits of each to be exploited : there is a tremendous quantity of devices with Ethernet interfaces, and Centrenet has a high bandwidth together with inherent "network intelligence" in the form of the NIMs. Consequently, it would be possible for Ethernet based networks of workstations to utilize Centrenet for interconnection and access to high performance devices providing specialized services.

On another path, the combination of digitized speech and images could allow the development of

> "an Integrated Information Presentation system, which could provide generalized inter-personal and person-computer comunications." [37]

These applications rely on the high distributed bandwidth provided by Centrenet, and benefit from the NIM support for decentralized network services.

1.3 Summary

This chapter has concentrated on the aims of the work described in this thesis. The case study presented here is an indication of the target system for this research and will be used to illustrate the reasons for certain design decisions in later chapters.

However, the MU6 system itself is not a dominant factor in the direction of this work, but more of an inspiration to strive for even better. It should become clear in chapter 3 that the computing environment that is the motivation for this thesis has very different characteristics from that of the MU6 system.

In the following chapter, a study is made of different approaches towards forming multiple computer systems. This study is in part retrospective, considering some of the more notable systems of the recent past, takes an overview of contemporary distributed computing systems and suggests some paths that are being followed in current research.

Chapter 2

Multiple Computer Systems

2.1 Introduction

This chapter considers a number of approaches to forming multiple computer systems. The systems selected for inclusion in this study provide a varied view of some of the interconnection strategies that are in common usage. These systems share certain characteristics with others, but they also have striking disimilarities. Whilst the systems described here do not represent an exhaustive collection of all research efforts in this area, they do provide a useful insight into most of the principles and techniques that are fundamental to the study of multiple computer systems.

2.1.1 Classification

It would be useful to categorize the various systems mentioned in this chapter, in order to clarify the differences and similarities between them. However, most of the systems fall into more than one group, so impairing any such classification.

All of the systems are composed of more than one computer. Some are *homogeneous*, meaning all of the computers share the same architectural structure, others are *heterogeneous*, in that the systems are composed of computers of more than one type. All of the systems could be described as *distributed*, because the processing does not all occur at the same place. However, the degree of distribution of processing and control is very variable, and for some of these systems, such as the IBM PC/VM Bond described in section 2.2.3, any claims to being a "distributed processing system" are rather tenuous.

Philip Enslow Jr. considers the description of systems as "distributed data processing system" in [28]. His list of claims made by some such systems is typical of many of the systems in this chapter, but he goes on to give a "research and development" definition of distributed data processing systems.

1. A multiplicity of general purpose resource components, but homogeneity is not necessary.

2. A physical distribution of these components, interacting through a communications network.

3. A high level operating system to unify control of these components, although individual processors may have different local operating systems.

4. System transparency, so that services may be requested by name rather than location.

5. "Cooperative autonomy" characterizing the operation and interaction of the resources.

Transparency is a key requirement, and the LOCUS system described in section 2.3.2 is a prime example of what can be achieved in this respect. Indeed, for the LOCUS system, "network transparency" was the primary objective, with "Unix compatibility" falling a close second. A number of other systems have been based on, or incorporated, a Unix-like system; the Andrew system, section 2.5.3, is another example.

A common approach towards providing a unifying high level operating system is to require all nodes to execute the same system software. The Apollo DOMAIN, section 2.6.1, has a collection of nodes that all possess the same standard set of services. Other systems have adopted a service-based approach, where there are nodes that are each dedicated to the provision of a specific task, such as the Cambridge Distributed Computing System, section 2.3.1.

An alternative scheme involves the nodes in the system providing a minimal framework on which a variety of different operating systems may function. The distributed V Kernel, section 2.4.1, provides a "communications infrastructure" for the modules of an operating system. The WFS filesystem, section 2.4.2, supports a rudimentary set of file operations, but may be used by a number of different operating systems. Amoeba, section 2.4.3, constitutes on object oriented approach to achieving similar objectives.

Most of the systems described here are based around a communications network, variously involving Ethernet, the Cambridge Ring, token passing rings and higher performance networks. However, the MISS network, section 2.2.2, uses conventional terminal lines and the IBM PC/VM Bond uses a co-axial cable.

2.1.2 System Models

Andrew Tanenbaum describes three categories of system models into which, he claims, most distributed systems fall. [99]

1. The **minicomputer** model is composed of a collection of identical minicomputers. The user is connected to one of these machines and is able to communicate with the others. The LOCUS system is implemented in this fashion, and the mainframe networks discussed in section 2.2.1 are an extension of this model.

2. The **workstation** model involves cooperation between a number of workstations, each one associated with a single user. The Dartmouth College Personal Computer Project, section 2.5.3, has these characteristics and the implementation of the V kernel incorporated a number of SUN workstations.

3. The **processor pool** model is based on a collection of processors that are temporarily allocated to users for computation. The Cambridge Distributed System has a pool of processors that are allocated statically for the duration of a terminal session. Amoeba uses groups of dynamically allocated processors for specific computational tasks.

2.1.3 Objectives of Study

A mammoth amount of research effort has been devoted to the investigation of distributed systems and a wealth of literature exists to describe these efforts. Any survey of work in this area is, by necessity, selective in content, and this study is

no exception. The primary objectives of this review are to provide a useful insight into some of the most fundamental techniques and to highlight certain features that are of particular interest with respect to the research work described in this thesis.

The study commences with a look at networked computer systems that fail to meet the criteria for classification as "distributed data processing systems", but which nevertheless prove to be of interest in this area. Subsequently, two "classic" distributed systems are described in some detail. These two systems have provided the impetus for many other distributed systems.

Alternative approaches towards providing distributed systems are then considered. These systems introduce extra flexibility in the services supported because they only provide a "streamlined" set of functions. The campus computer networks discussed in the next section form a stark contrast to the other systems in this chapter because of their scale. Many of the decisions made during the development of these systems are relevant in a wider context.

The final system to be included in this study is a well developed commercial product. The incorporation of a supercomputer into this system provides additional interest.

2.2 Networked Computer Systems

This section looks at multiple computer systems that are not considered to be "distributed data processing systems" because either their control is centralized

or the interconnection is achieved by means other than a general purpose communications network. First, the connection of mainframes in local area networks is discussed, with particular regard being given to the UMRCC configuration, which is based around a CDC LCN. This is followed by the MISS network, which developed an approach for sharing the processing capabilities of computers of different processing power for support of laboratory environments. Finally, more recent approaches towards interlinking microcomputers and mainframes are described.

2.2.1 Interconnection of Large Computer Systems

For many years now, large computer installations have moved away from the use of a single large mainframe towards having multiple processors at a single site. This is for a number of reasons :

- critical applications may require a *fall-back* system
- the processing requirements may exceed the capacity of even the most powerful mainframes
- utilization of specialized equipment and computers may be desired, e.g. for graphics, special computation needs or communications.

In the past, the interprocessor connections were implemented according to demand, resulting in expensive and inflexible solutions. The advent of high bandwidth local computer networks such as **HYPERchannel** [107] and Control Data's **Loosely Coupled Network** (LCN) [18,47] permitted a more generalized interconnection strategy to be adopted.

There is a fair degree of similarity between the HYPERchannel and LCN. Both systems are based on multiple common data highways (**trunks**), implemented using coaxial cable with data transmission rates of about 50 Mbit/sec. Processors and other devices are attached to the trunks via microprocessor controlled interfaces, called **Network Adaptors** in HYPERchannel and **Network Access Devices** (NADs) in LCN. Each interface may be connected to up to 4 trunks, and the number of such interfaces per trunk ranges from 32 for the LCN to 64 for the HYPERchannel. Both systems have a range of many thousands of feet.

A Large Installation

To simplify this discussion, this section will concentrate on one of the systems, the LCN, and in particular one installation, that of the University of Manchester's Regional Computer Centre (UMRCC). In the period 1983-1984, the UMRCC saw the departure of its ICL 1900 series mainframes, and the arrival of a CDC CYBER 205, CYBER 176, Amdahl 470/V7A and a MASSTOR M860 mass storage device. From their introduction into service, these devices were capable of communicating with each other, and the existing CDC CYBER 170-730 and 7600 joint system, using the Control Data LCN. The configuration was approximately as shown in figure 2–1.

Each of the mainframes runs a Control Data package called the **Remote Host Facility** (RHF) in order to utilize the LCN and communicate with the other mainframes. The CYBER 170-730 was primarily for local interactive use, but the Amdahl 470/V7A performed the rôle of *front-end* for the more powerful *back-end* mainframes (the two 7600s and the CYBER 205).

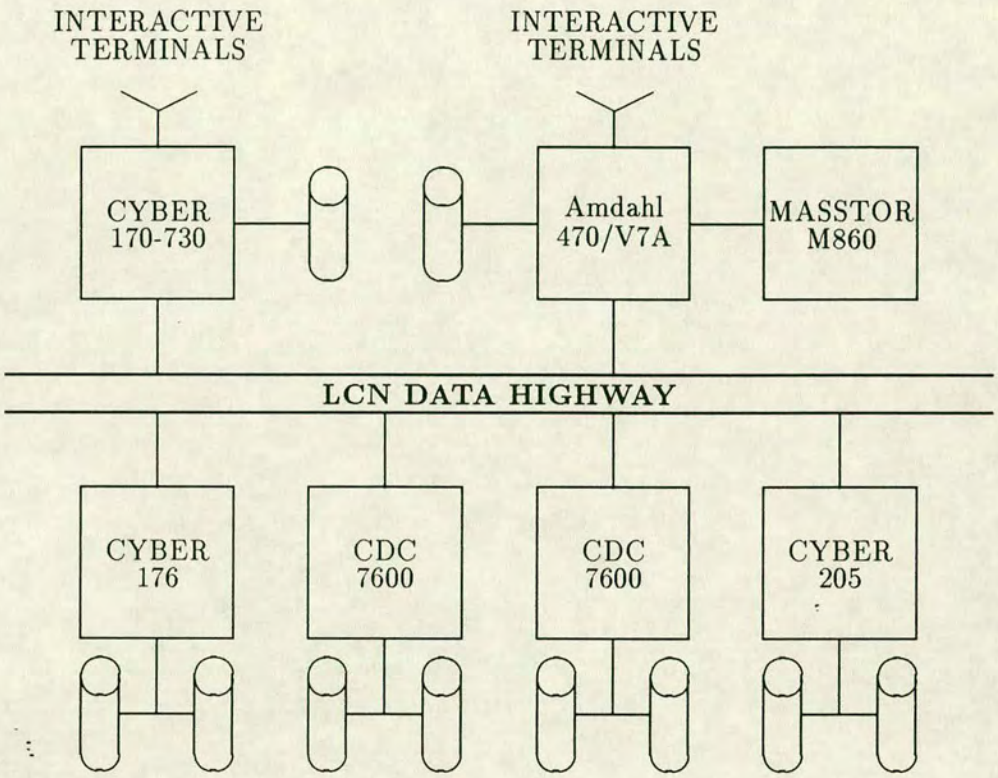

Figure 2–1: UMRCC System Configuration (1984)

Control Data provide implementations of the RHF software for different operating systems that use the ISO/OSI 7 layer reference model as the basis for the LCN structure.

The lower 4 levels are provided by the LCN and the upper 3 levels have specific implementations for each of the supported operating systems.

Level 5 - Host Driver Interface : this is the NAD interface, and where necessary performs conversion to 8-bit ASCII. [1]

Level 6 - Host-to-Host : establishes connections between hosts.

Level 7 - Application-to-Application : allows block or file transfers to occur between two applications. UMRCC uses the following applications :

1. The **Permanent File Transfer Facility** (PTF) manages the transfer of permanent files from one mainframe to another, using a single command - **MFLINK**[40].
2. The **Queue File Transfer Facility** (QTF) enables job files to be transferred from the *input queue* of the local host to that of the remote host, and for the output files to be re-directed back to the *output queue* of the local host. The user does not explicitly communicate with the QTF, but the first line of the job file will indicate a remote site identifier, and the system will automatically re-route the job.

The network capability provided by the LCN and the RHF software is very crude, and only really intended for large batch jobs and data sets. The rôle of the application packages and the nature of the commands are very familiar to mainframe users and they provide an effective means for transferring programs and data to different mainframes when the need arises.

Consider the sequence of events when a job is submitted on the Amdahl 470 for processing on the CYBER 205 :

---

[1] 8-bit ASCII is the format for all LCN transmissions.

1. As with normal batch jobs, a file is formed with the necessary commands to control execution of the program.

2. When the job is submitted, the Amdahl 470 will perform preliminary scheduling, and will eventually transfer the control file to the input queue of the CYBER 205 using QTF.

3. The CYBER 205 performs further scheduling of its input queue and when the job begins execution, authentication and accounting procedures are initiated.

4. If files on the Amdahl are needed, they are fetched using PTF.

5. On completion of the job, the output is re-routed to the output queue of the Amdahl 470, and from there to the user's terminal or a lineprinter.

The CYBER 205 behaves like a conventional batch processing mainframe - with the added bonus of the RHF networking software. Despite the fact that the processor has a specialized architecture designed for computations involving large vectors, the mainframe is required to perform accounting, scheduling and other functions that seriously reduce its efficiency. The front-end acts as little more than a Remote Job Entry (RJE) station, with the exception of providing mass storage and performing initial job verification checks.

If this operational model were to be followed in an environment composed of workstations and high performance processors, it is clear that these specialized machines would be inefficiently used. A more appropriate model may be to shift the majority of the housekeeping chores to the more general purpose machines wherever possible.

2.2.2 Hierarchical Networking

The **Minicomputer Interfacing Support System** (MISS) [8] was a minicomputer network implemented by the Institute for Computer Research at the University of Chicago. The rôle of MISS was to provide support for the needs of experimental laboratories using low power minicomputers for real time monitoring and control. This was achieved by a hierarchical network of minicomputers running applications that interfaced to remote centralized facilities.

MISS Hierarchy

The MISS hierarchy comprised three levels. The lowest level consisted of a variety of minicomputers located at a number of application sites, under the control of their users/owners. These minicomputers spent most of their time operating in a *standalone* mode, and hence the interface to MISS was minimal.

The intermediary level was a computer dedicated to the support of the lower level minicomputers by providing shared access to a number of services. The facilities provided by this level included extended I/O services, mass storage, access to specialised support services and connection to central computing facilities. The uppermost level in the hierarchy was the campus Computation Centre, providing general purpose batch and interactive computing.

Figure 2–2 illustrates the three level hierarchy of MISS. The lowest level minicomputers, small PDP-11s and PDP-8s, are connected to the intermediate level by 9600 baud communication lines; the intermediate level, a PDP-11/45, is in turn connected to the upper level, an IBM 370/168, by a 50 kbaud line.

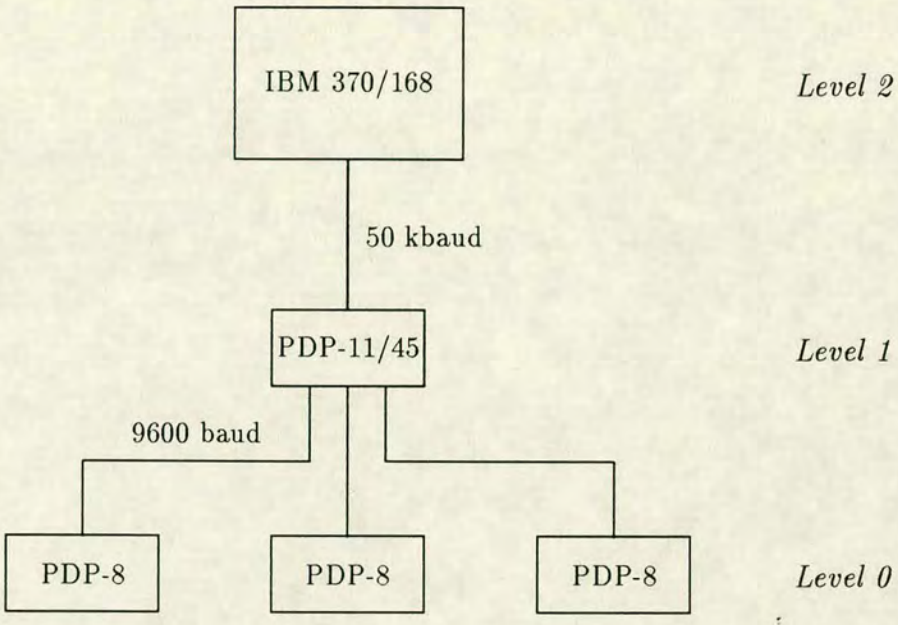

Figure 2–2: Three Level Hierarchy of MISS

Remarks

The key feature of the MISS network is the division of responsibility amongst the three levels of machine. The lowest level is concerned with small scale computation but needs to have a good responsiveness for real time monitoring and control. The second level exists to support the operation of the small machines and as such is quite heavily burdened. The uppermost level is the central computation resource, and provides the main processing resources for the entire network.

Whilst the level 0 machines are capable of operating independently of the remainder of the system, their processing power is severely limited, thus requiring access to the level 1 machine for even the simplest tasks. The connections between the machines have a low bandwidth and do not really form a general purpose communications network; if the level 1 machine crashes then all of the level 0

machines are stranded. Furthermore, the topology of the network is extremely evident to the user. Indeed, the user needs to be aware of the structure of the system in order to make any use of the system.

In conclusion, it is clear that the MISS network is remote to the central theme of this review. However, the layered division of responsibility within the system is more generally applicable. In fact, there are features of the MISS network that appear in the IBM PC/VM Bond, described in the next section.

2.2.3 Micro-Mainframe Links

The MISS system, described in the previous section, enabled the processing requirements of a scientific community to be distributed to a large number of simple minicomputers. These small computers were capable of satisfying the more localized computational needs, such as real time monitoring and control of experiments, but the more intensive processing was performed on the central mainframe. In the commercial sector, the ever increasing presence of personal computer systems, such as the IBM PC and its compatibles, has led to the adoption of some of the facets of the MISS system.

Traditionally, the company's mainframe was responsible for holding and manipulating all of the corporate databases. The ever increasing requirements of users meant that the mainframe was stretched to the limit of its capabilities. The arrival of microcomputers would appear to provide the solution to this demand for computing power, but there remain a few difficulties.

The greatest problem is the fact that the corporate databases are centralized and unified, but a multitude of departments will be continually updating and inspecting this information. Present microcomputers are incapable of storing data of this volume, and the intricacies of maintaining a consistent database, spread amongst a very large number of machines, are immense.

Probably a more important obstacle to the replacement of the company's mainframe by personal computers is the company's own data processing department. The accumulation, management and dissemination of the corporate data is controlled by the DP department and it is unlikely to accept the devolution of its power too readily [69]. More pertinently, the practices of data management are well established, and so all changes must be undertaken gradually and with caution.

For these reasons, and others, companies are opting to retain the mainframe for holding databases, but use the microcomputers to retrieve sections of this data and analyse it off-line. However, there remains some opposition to the idea of allowing the personal computers to update the databases.

There are four approaches to connecting microcomputers to mainframes [93] :

1. The microcomputer can simply emulate a terminal, usually a DEC VT-100 or IBM 3101.

2. In addition to terminal emulation, the discs of the personal computer may be used to *capture* the information from the mainframe and thereby enable local analysis.

3. The mainframe may provide access to some of its files as if they were large virtual discs, directly connected to the microcomputer. This allows the normal operating environment of the small computer to be used, whilst permitting access to the remote data.

4. Close cooperation between applications software on the mainframe and the microcomputer can result in the personal computer user being able to access and manipulate data on both machines in the same manner.

The first two techniques are in common use and are well understood, so they are not considered further. This section will consider a particulár implementation of the third strategy, the IBM PC/VM Bond, and will briefly survey some new attempts to provide the fourth solution.

PC/VM Bond

The **PC/VM Bond** [57] was developed jointly by the IBM Endicott Laboratory and the Thomas J. Watson Research Centre. It is formed from two separate software packages :

PC Bond runs on a mainframe running the IBM **Virtual Machine/System Product** (the VM operating system[26]).

VM Bond is executed by an IBM Personal Computer (PC), with the 3278/79 Emulation Adaptor Card attached.

The two programs cooperate to allow the IBM PC user to :

1. Use a *virtual disc* from PC DOS, that is in reality a CMS (Conversational Monitor System) file on the mainframe. The virtual disc behaves as a fixed disc on the microcomputer.

2. Send messages to users of PC/VM Bond or VM on the same mainframe, or another VM host connected by a network.

3. Emulate a 3278 model 2 terminal, accessed by a *hot-key* sequence. Users are able to assign 3278 function to keys on the IBM PC keyboard, according to their personal preference.

4. Issue a subset of VM commands from the PC DOS and receive the results back whilst in a program or at command level. It is possible for a user to issue a command to VM, such as a compilation request, and work on PC DOS while the VM system processes the request.

5. Write programs in the REXX/PC language on PC DOS, and thereby develop applications that exploit both VM and PC DOS. The REXX/PC language (REXX stands for Restructured Extended Executor) is a language that is common to both systems and is intended as a form of job control interpreter. The language has features that provide a means for a single REXX/PC program to call routines on the VM host and process the results locally under PC DOS.

In addition to these features, there are utilities provided that convert files from CMS format to common microcomputer formats, such as Lotus, Dif and Sylk. Curiously, one application of the PC/VM Bond has been to control and

monitor laboratory instruments, returning the data to a mainframe for detailed analysis - just as with the MISS system, 10 years earlier.

Although the PC/VM Bond has satisifed the need for a means of enabling cooperation between microcomputers and mainframes, there are still problems. The file size is limited to 32 Mbytes, which is too small for many mainframe applications. There is also a performance constraint in that the continued need for a significant amount of processing by the mainframe will impair the response time.

Further Integration

There are two paths towards achieving even greater levels of cooperation between microcomputers and mainframes, permitting the merits of both types of system to be exploited; mainframes can manage large quantities of data, whilst microcomputers provide a better user interface. Both of these paths are under active development.

The first approach lies in the production of applications packages that distribute their functions between machines on the personal computer and on the mainframe. Fundamental to this approach is the use of a suitable interface between the two halves of the application. The **Structured Query Language** (SQL), developed by IBM, is one such interface that is emerging as a standard. SQL allows random access to data, with a resolution to single fields of a record, and there are a number of products that exploit its features.

The alternative approach involves much closer integration of the operating systems of both the mainframe and the microcomputer. For some time now, DEC

have been able to claim some degree of success in achieving this objective. The DEC family spans from the Microvax 2000 to the 8600 range of machines, all of which run a version of the VMS operating system, and are capable of running the same program.

The recent series of IBM product announcements are indicative of a strong move in the same direction. The key to this IBM strategy would appear to be the **Systems Application Architecture** (SAA). SAA is an assembly of standards, encompassing communication protocols, user and application interfaces, of which SQL is one element. Much of SAA has yet to be fully fleshed-out, but the definition of SAA took a step further with the announcement of OS/2 [94], the operating system for the Personal System/2 (PS/2) range of microcomputers.

In addition to the PS/2 microcomputers, other IBM machines that are expected to be covered by SAA are the System 3X minicomputers and the System 370 mainframes. So, although the concept of SAA is in its infancy, the closer integration of IBM computer systems looks inevitable.

2.3 Distributed Systems

This section concentrates on two well known implementations of distributed systems. The Cambridge system and LOCUS are described in some detail because of the importance of the concepts that they introduced. Both systems acted as test beds for new ideas that have since become fundamental to the subject of distributed computing. Furthermore, these ideas and methods are of particular relevance to the work described in subsequent chapters.

2.3.1 The Cambridge Distributed Computing System

The **Cambridge Distributed Computing System** was developed by the Computing Laboratory at the University of Cambridge towards the end of the 1970's. Research work on the system followed the development of the Cambridge Digital Communication Ring and, although the present system uses the Ring, the choice of local area network does not dominate the structure of the system. In figure 2–3 an example Ring with six stations is shown [11].

In [76] the motivation behind the Cambridge Distributed System is explained :

> "with the advent of inexpensive microcomputers and the ready availability of minicomputers we were stimulated to consider the use of interconnected machines constituting a coherent system rather than just a collection".

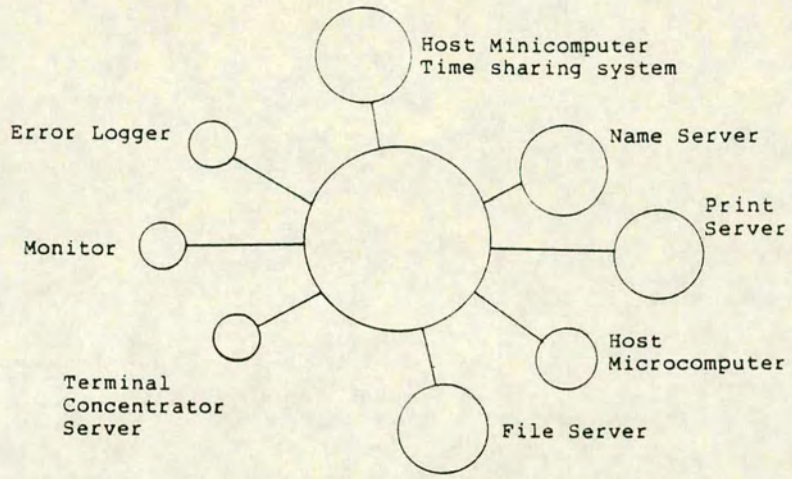

Figure 2–3: An Example Ring

In the Cambridge system, the central mainframe is replaced by a collection of smaller computers that fall into two categories : the small servers and the processor bank.

Small Servers

The small servers are dedicated machines, providing useful basic services [12] such as a file service, name service and time service. These machines are fairly simple in design, and thereby inexpensive and reliable. In the system described in [76], the small servers were implemented using Z80-based microcomputers, constructed on $7^1/_2$ inch square circuit boards using about 40 chips for a total cost of less than £100.

For the small servers, the distribution of services is along functional lines, with each server providing a single service. This has the benefit of providing a

much more predictable level of performance because of the independence between services. This also results in greater resilience to system failures.

Although the small servers are assigned to specific tasks, their functional dedication is not fixed at hardware level. Each server has a copy of a program called SBOOT in ROM. When a small server is activated, the SBOOT program sends a message across the Ring to a special server called the **Boot Server**. The Boot Server uses a reverse look-up facility provided by the Name Server in order to determine the identity of the new server, then arranges for the appropriate code to be transferred to the server. The SBOOT program then initiates execution of this code.

The Boot Server introduces flexibility into the allocation of services to servers and is therefore required to have an even greater degree of resilience to system or Ring failures. In the Cambridge Distributed System, two approaches are taken to ensure the robustness of this "remote bootstrapping" technique.

- Since the Boot Server is a stand alone machine, if it becomes unable to access other services, its internal tables may be manually manipulated.

- If the Boot Server becomes inaccessible to the servers, the SBOOT program enters a different mode in which any machine can instigate a loading operation.[2]

---

[2]The SBOOT program alternates between its two modes, so if the Boot Server recovers then the usual automatic mechanisms come into effect.

The **Name server** plays an important rôle in the operation of the Cambridge Distributed system. To use any service, the client must first of all send the name of the service to the Name Server in order to obtain the location of the server. The Name Server matches all incoming names with its internal tables to determine the relevant address.

Processor Bank

The small servers provide the administrative functions in the Cambridge Distributed System, reminiscent of the operating systems of traditional mainframes, but the bulk of the computing power in the system resides in the **Processor Bank**. The minicomputers that constitute the Processor Bank are not permanently assigned to particular users, but are allocated for short periods of time to individuals on demand. They may be considered to be *processing servers*, providing computing power for the users of the system.

Once allocated, a processing server will be loaded with the code specified by the user and, for the duration of the session, the machine will act as if it were the user's own personal computer. The user has complete freedom over the use of the machine and may execute code that accesses network services.

The only means that the system has of exerting control over the use of the service is through the Processor/Ring interface. The interface is concerned with the loading of code, the initiation and termination of execution, debugging operations and access to other network services. This is in common with the requirements of the MU6-V I/O-processor's requirements, described in section 1.2.2.

To meet these needs, the Ring interface has to have some degree of intelligence and be able to manipulate the data in an efficient manner. The interface described in [76] is microprocessor controlled with a DMA capability for high speed data transfer, figure 2–4. The design of the interface is such that it may be readily adapted for use with different machines.

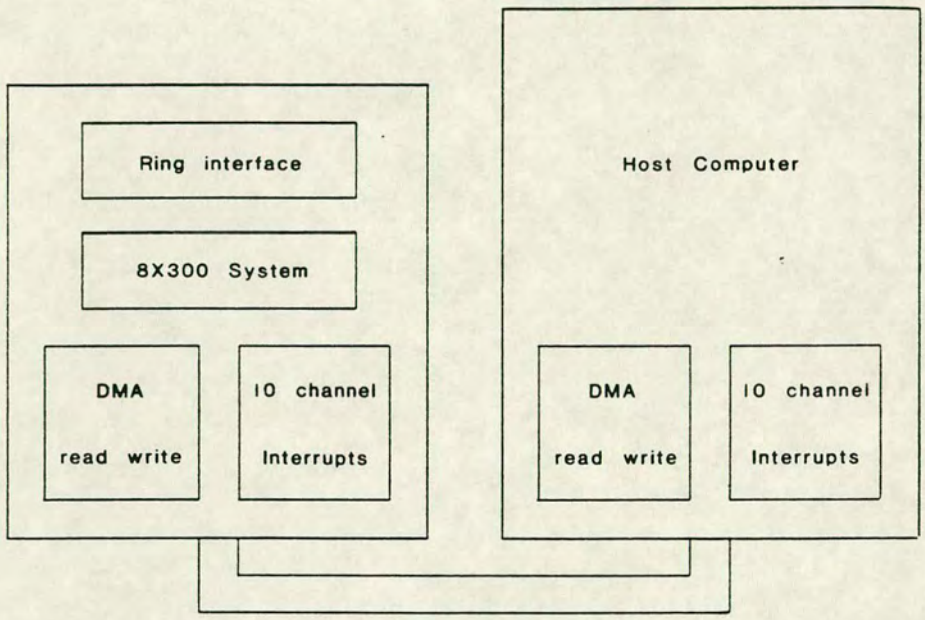

Figure 2–4: Processor/Ring Interface

File Storage

In the Cambridge Distributed System there is a central **File Server** and the processing servers do not have local discs. This means that users may access their files from whichever processing server they are allocated, and allows useful programs and data to be shared by the users.

There are a number of different operating systems used in the system, with varying requirements of the filing system, and some of the servers require use of permanent storage facilities, but without the overheads imposed by filing system constraints. To satisfy these needs, a **universal** file server was designed that provides standard functions and uses a basic linear naming system that may be shaped by higher levels of software as needed.

Each file is uniquely identified by its PUID (Permanent Unique Identifier) which includes a random element that effectively makes access impossible without permission from the higher level system software.

Access to the System

Every user of the system has a personal computer that is only capable of performing simple tasks, the most important of which is to connect to one of the servers in the processing bank. The user's command to connect to a processing server includes details of the abilities required of the machine. On the basis of these needs, and the availability of suitable processing servers, the **Resource Manager** will allocate a server for the stated period, and then reconnect the user's terminal to the newly assigned machine. Subsequently, the Resource Manager is only utilized when the assigned period is concluded, the user presses BREAK at his terminal or the processing server indicates that the session has finished.

Operation

Reviewing the operation of the Cambridge Distributed System, the authors of [76] observed that the use of screen-based software imposed a substantial load

on the processing server. Their proposed solution to this involved increasing the intelligence and processing power of both the user's personal computer and the processor/Ring interface. Following this, use of a screen-based communication protocol would be possible, with significant improvements in performance.

The system is in daily use by a substantial user community, and has expanded to incorporate three rings with about 90 attached devices. There are now a number of such configurations in use in universities throughout the country [11].

2.3.2 Network Transparency

The LOCUS operating system was developed at UCLA with the primary objective of providing a high level of **network transparency** [108] so that the network of computers appeared to the users as a single machine. Consequently, services would be accessible in the same manner regardless of whether they were provided locally or remotely. This approach provides enormous benefits for both the user and the programmer; the former is presented with a simpler interface to the system, whilst the latter is rewarded by reductions in the time and cost of developing and maintaining software.

The LOCUS maxims are "very extensive transparency" and "Unix compatibility", so that an application written for a single Unix computer will run, unmodified, on the LOCUS distributed system. This extends to a heterogeneous LOCUS system, which means that if a user requests a service that only exists on a machine of a different type then LOCUS ensures that the program executes on an appropriate computer.

Supporting this broad transparency introduces the need for a high reliability and availability. To answer this need, LOCUS uses a filesystem that provides automatic replication with **name transparency** so that an object's name is independent of its location, and a full implementation of **nested transactions.** Additionally, the partitioning and merging of subnets is supported.

The concepts that form the basis for the LOCUS operating system are fundamental in the field of distributed computing. Consequently, it is worth examining these techniques to gain an insight into, what Gerald Popek calls, "the unpleasant truth of distributed systems" [81], and more especially the approaches taken to overcome these unpleasantries.

LOCUS Distributed Filesystem Overview

The most critical feature of the LOCUS system architecture is the filesystem, particularly the distributed naming catalog, because it is utilized by most of the system. The performance of an operating system is frequently dependent on the efficiency of its file system . This is especially true for Unix because file activity is so predominant, and consequently the file system plays a critical rôle.

The LOCUS file system appears to both users and applications software as a Unix-like single tree-structured naming hierarchy. The LOCUS tree structure encompasses the names of all directories and files of all the machines in the network, but the names are *location transparent* so that data and programs can migrate to different sites and yet still be accessed in the same manner.

As with Unix, LOCUS is procedure based. Hence, processes make use of services supplied by the system through system calls which cause a *trap* to the

kernel. The process need not be aware of whether the service is locally or remotely provided because the LOCUS kernel will determine the location of the service and, if necessary, send a message to the appropriate remote site. This is a form of **remote procedure call** [77] and the sequence of events is illustrated in figure 2–5.

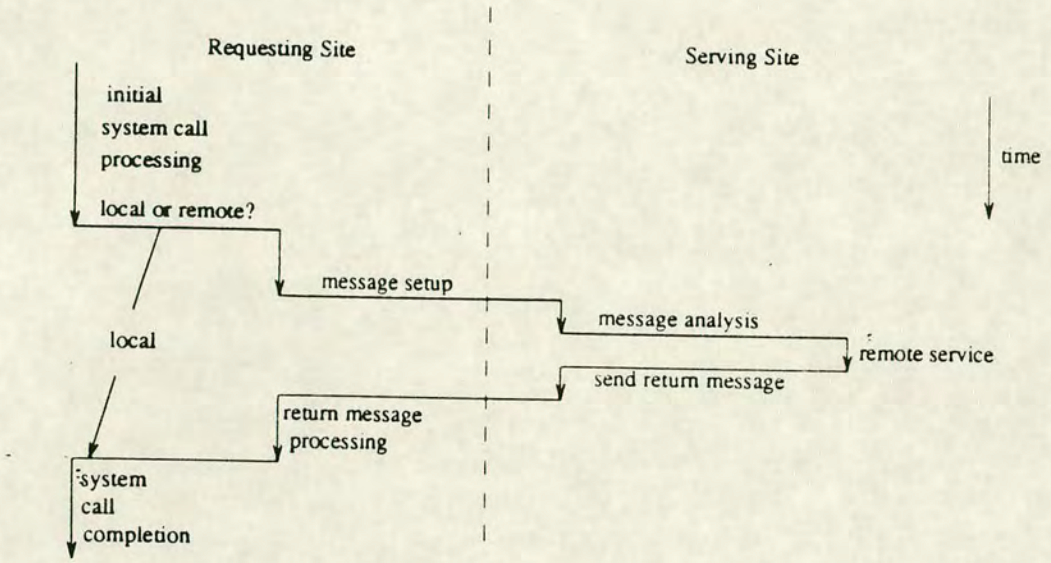

Figure 2–5: Processing a System Call

The LOCUS kernel detects, at an early stage, whether or not the system call is to a foreign site, and consequently is able to optimize its operation for local services. This optimization is successful in that the time to service a local call in LOCUS is comparable to that of system calls in a single machine implementation of Unix. This is considered further in section 2.3.2.

Remote Services

It is important to note the technique taken in LOCUS to support remote services, since it supplements a shortcoming of the Unix system on which LOCUS is based.

The approach that has been taken traditionally is to pass the service call up to an application level process. This incurs enormous overheads in operating system processing because of requiring the following steps :

- message transfer to the application level process
- scheduling of the process
- implementing the appropriate kernel call
- passing the results back to the process
- returning a message to the requesting site.

Where an application process is to be executed, this approach is appropriate because the mechanisms listed above are needed for the protection of the program. However, in the case of a LOCUS system call, these overheads are needless and in fact would dominate the implementation of the service.

An alternative would be to satisfy the service request entirely within the interrupt routine that initially received the message. This should be discounted immediately because interrupt processing should be reserved for time critical events; processing of the service call would require an unsatisfactorily long period of time to be spent at interrupt level. The LOCUS designers also point out the additional safeguards needed for the integrity of data structures that the service may need to access.

The technique introduced in LOCUS involves the creation of *lightweight processes*, which are designated **server processes**. Their code and stack are resident

in the operating system nucleus, global variables are part of the nucleus reserved storage area and they are able to call internal subroutines directly. Consequently these server processes are compact and yet very efficient.

Although the LOCUS server processes are perhaps insufficiently generalized to satisfy the advocates of lightweight processes, they were an ideal solution to the requirements of the LOCUS designers, providing structural simplicity together with enhanced performance.

Synchronization

LOCUS filesystem operations involve up to three logical functions, that are in turn represented by three logical sites :

1. **Using site** (US) : the site requesting file access, and the destination for pages of the file.

2. **Storage site** (SS) : the location of the requested file, and the site selected to supply pages of the file to the using site.

3. **Current Synchronization site** (CSS) : this site enforces a global access synchronization policy for the requested file's group and selects a suitable storage site for each request.

These three independent rôles mean that any individual site may operate in one of eight different modes. The general case, where all the logical functions are located on different physical sites, is illustrated in figure 2–6 which shows the **open** protocol. The CSS plays an important part in this protocol. It enforces

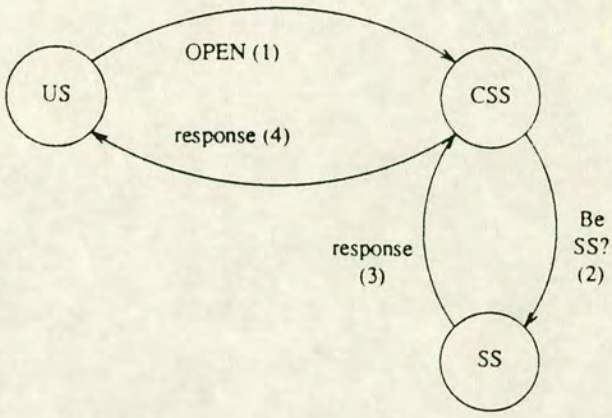

Figure 2–6: The Open Protocol

synchronization controls, determines the storage site for the operation and supplies information about the latest file version for all prospective storage sites.

File Replication

In the LOCUS distributed system, replication of certain critical files is essential in order that the system will continue to operate even when some machines are inaccessible. There are, of course, additional benefits for the user :

- The presence of multiple instances of a file results in increased availability.
- There may be performance improvements through using a nearby copy of a file rather than one that is more distant.

In a tree structured naming system, the desirability of replication increases in relation to the level of utilization, and correspondingly in accordance with the level within the tree. This is especially true for directories, illustrated by the fact that

the root and upper level directories experience a much greater degree of "look-up" than the lowest level sub-directory.

The LOCUS filesystem operates file replication by distinguishing between **logical filegroups** and **physical containers**. For every logical filegroup there will be a number of physical containers at a multiplicity of sites. There is no requirement for every physical container to hold versions of all the files in a filegroup, so any container will possess only a partial subset of files.[3]

Propagation of the latest copy of a file to all its storage sites occurs when a **file commit** operation is performed. This technique of atomically committing changes is commonplace within database systems, but is equally valuable in general purpose applications. When a commit is issued, the current storage site will replace old copies of a file's pages on the disc by amended versions and then notify all of the file's other storage sites of the commit operation.

Context Dependent Files

There are certain instances when the implementation of a command needs to vary according to the nature of the host processor issuing the command. This could be for any of a number of "hardware heterogeneity problems", such as differing number representations, byte ordering and hardware differences. In LOCUS, files that are in some way "context sensitive" are treated specially through the use of the following technique :

---

[3]The exception to this is the *primary* physical container, which holds a copy of all files in the corresponding filegroup.

- The global name of the file is made to refer to a **hidden directory**, which in turn holds the different versions of the file, named according to their context.

- Whenever the special file is accessed, the context for the current process determines which of the files in the hidden directory is to be used.

By appending an '@' to the name of a hidden directory it is possible to override this mechanism, when necessary.

Interprocess Communication and Remote Tasking

A **pipe** is a Unix mechanism that enables one-way communication between processes. The communicating processes use the pipe as if it were a normal file, whereas it is implemented as a finite length buffer. Hence, the reading process is suspended if it attempts to read when there is no data in the buffer, and the writing process will be blocked if it fills the buffer. An attempt to write to a pipe that has no reader, or read from an empty pipe with no writer, results in an error condition that is notified to the process.

A pipe may be established in one of two ways :

1. A process may issue the **pipe** system call, and then spawn child processes which will share the same file descriptor table and thereby be able to access the same pipe.[4] This is called an **unnamed** pipe.

---

[4]The processes will be divided into those that close the pipe's read descriptor but maintain the write descriptor and so act as *writers*, and those that do the opposite and act as *readers*.

2. A **named** pipe has a name in the filesystem and is accessed in the same way as a conventional file, i.e. using an **open** call. This allows unrelated processes to communicate.

LOCUS supports the operation of both named and unnamed pipes. The mechanism to achieve this involves three sites : the current **reading site**, the current **writing site**, and the **storage site**. The storage site uses tokens to synchronize the operation of the reading and the writing sites.

The LOCUS implementation of pipes allows the network to remain totally transparent to the user. Similarly, the Unix mechanisms for the creation of processes have the same interface in the LOCUS system. Two system calls are involved in the spawning of new processes :

1. **fork** creates a new process, executing the same program as the calling program at the point in the code where the fork occurred. The child process inherits the same data as the parent and has a copy of the same file descriptor table.

2. **exec** causes the code and data of the process to be replaced by those specified by the parameters.

Frequently, a fork operation is followed immediately by an exec, so in LOCUS the **run** system call is introduced to optimize this function. Also, the **migrate** system call allows a process to move to another site for execution in the middle of a program. Table 2–1 summarizes the effects and relationship of the four system calls.

	Old Image	New Image
Move old process	*migrate*	*exec*
Create new child	*fork*	*run*

Table 2–1: Remote Tasking System Calls

Fork and migrate both require the existing memory image to be copied for the new process, hence these system calls may only be used with machines of similar architecture. Exec and run will also need to copy the process and data structures, as well as the command line and environment, but execute new programs so these system calls may be used with dissimilar machines.

Overcoming Difficulties

The benefits of file replication have already been explained, but it would be foolish to disregard the many difficulties that may arise from its use. Primarily, the existence of multiple copies of a file only creates problems when *partitions* appear in the structure of the system and the possibility arises of the individual copies of a file being updated separately.

When partitions are merged, the system needs to perform consistency checks on all copies of files. Where one copy of a file has been altered but the other copies are unchanged, the amended version is propagated through to the other storage sites. However, when multiple copies of a file have been updated separately, attempting to reconcile these versions will result in a *merge conflict.* Such a conflict might only

be resolved through the intervention of higher level software, such as a database manager, or direct manipulation by a user.

LOCUS incorporates consistency checking and has special purpose protocols for dealing with partitions and mergers. Low level decisions of a trivial nature are made by this software, but more difficult conflicts require the intervention of a user, and simple tools are provided to assist the user in such endeavours.

There are protocols provided in LOCUS to enable *dynamic reconfiguration* of the system to occur transparently to the user. These protocols use a high-level synchronization strategy and are largely independent of the LOCUS system architecture.

Performance

Performance measurements made using LOCUS indicated that local system calls are on a par with those of a single site Unix, and that remote accesses are insufficiently distinct from the local case to warrant consideration by the user. The authors of [108] admit to being surprised at this level of *"performance transparency"*, but make the following comment about network transparency in general :

> "experience with transparency has been very positive; giving it up, once having had it, would be nearly unthinkable"

The LOCUS designers achieved their objective of providing a high level of network transparency, but at what cost :

- LOCUS is a third larger than Unix, and is significantly more complex

- file replication poses great difficulties when recovery is needed

- dynamic system reconfiguration introduces immense problems in the cooperation of processes executing in separate environments.

The use of application specific protocols is essential in the LOCUS system in order that the performance degradation for remote system calls is minimized. For similar reasons, the compatibility and structural benefits of the ISO 7 layer reference model have to be foregone in the interests of efficiency.

In LOCUS, a single distributed operating system is provided for the users of the network of machines. They are presented with a familiar interface to the system because the LOCUS designers viewed Unix compatibility as an important benefit. Indeed it is, because of the wealth of software that already exists for Unix. The network transparency of LOCUS should mean that these applications are usable without modification. Retaining Unix compatibility required the LOCUS designers to rewrite the kernel completely, adding embellishments to support the network mechanisms.

2.4 Support for Distributed Systems

The previous section concentrated on two notable systems that provide *complete* operating systems; that is, it is relatively difficult to amend their basic form. This section turns to three examples of a *minimal* approach to supporting a distributed operating system. The **V kernel** and **Amoeba** systems comprise a concise kernel, present on all machines in the system, that contains the fundamental communication mechanisms and little else. The **WFS** file system includes a small set of file operations, but exerts no influence on the design of the higher level system functions.

2.4.1 Communications Infrastructure

When compared to the LOCUS system discussed in the previous section, the **Distributed V Kernel** represents a strikingly different approach to providing a distributed operating system. The LOCUS system encompasses all of the features of a single machine operating system, together with complex software to enable reliable distributed operation. In contrast, the V kernel only provides the mechanisms for communication between the elements of an operating system. The higher level functions are supported by servers outside of the kernel. These functions include the file system, resource management and protection services.

The distributed V kernel was implemented in the Computer Systems Laboratory at Stanford University on a group of SUN workstations, interconnected by 3Mb or 10Mb Ethernet. The workstations have no local discs, but rely on network

file servers for all secondary storage. However, the V kernel does not possess specialized protocols for file access, like those of LOCUS, but uses a general purpose set of communication primitives. These primitives do not incorporate provision for *streams*, to deal with network latency, but rely on a synchronous *request-response* model of message passing and data transfer.

These simple and general purpose primitives permit flexibility in the support of different types of network communication, but would appear to result in a significant performance degradation compared with the application specific protocols of LOCUS. In [20] the performance of the V kernel is considered, and the claim is made

> "the V message facility can be used to access network files at comparable cost to any well-tuned specialized file access protocol."

V Kernel Communication Primitives

The basis of V kernel operation is a number of small processes communicating by short, fixed-length (32 byte) messages. Each *request* message has a corresponding *response* message, and there may be an associated data transfer operation when larger quantities of data are involved. The basic communication sequence, designated a **message exchange**, has the following form :

1. The **client** process issues a **Send**(message,pid) to a **server** process and then blocks, awaiting a reply. The **pid** is a 32 bit globally unique process identifier.

2. The server process executes a **Receive** to accept the message, and may perform a number of **MoveTo** or **MoveFrom** operations to transfer data.

3. Finally, the server process returns a **Reply** to the client.

It is possible to append segments to messages, and use the primitives **ReceiveWithSegment** and **ReplyWithSegment**, to improve page-level file access.

The V kernel communication primitives are basic but fast and may be implemented efficiently because the buffers required by small, fixed sized messages may be statically allocated, thereby reducing queuing and buffering problems. The designers claim that the distinction between the exchange of small messages and the transfer of large amounts of data complies with observed usage.

As with LOCUS, remote services are implemented within the kernel, rather than through a process-level server, to avoid unnecessary overheads. Also, the LOCUS policy of not using layered protocols, because of the associated performance penalty, is followed in the V kernel. The synchronous nature of the message exchange enables reliable communication to be achieved using unreliable datagrams, without the need for explicit acknowledgements.

The implementation of pipes in the V kernel differs from the approach taken in LOCUS, as described in section 2.3.2. The V kernel supports pipes through the provision of a **pipe server process** [111] and uses the general purpose communication primitives. This contrasts with the kernel support and dedicated protocols of LOCUS.

A pipe is created by executing a **Send** to the pipe server, specifying the pids for the reader and writer processes, and setting the buffer size. The reader and

writer processes then have to execute **Send** messages to the pipe server in order to complete the creation of the pipe. Throughout the operation of the pipe, all of the processes involved communicate using the standard V kernel message exchange - no special protocol is needed.

In analysing the behaviour of V kernel pipes in [111], Willy Zwaenepoel observes that performance falls by 50 percent when the processes (reader, writer and pipe server) are on separate machines, compared to the case of the pipe server being local to either the reader or writer. Willy Zwaenepoel deduces that it is desirable for every machine to have a pipe server.

The use of a pipe server rather than dedicated kernel support is shown to reduce performance by 8 percent for remote pipes and 25 percent for local pipes. For the V kernel, where pipes are not a primary means of interprocessor communication, this overhead is considered to be acceptable.

V Kernel Performance

The performance of the V kernel is discussed in some detail in [20], and the authors emphasise that caution must be observed in drawing conclusions from their results. These results indicate that the message sequence **Send-Receive-Reply** takes three times longer in the remote case than for a local operation. However, the difference between the two instances is only around 2 msec, which frequently is insignificant when compared to the time needed to process the request in the server.

The more interesting results are concerned with the performance in file access. For page-level accesses, remote operations incur a delay of 4.2 msec, but for a disc

latency of 20 msec the remote access is around 18 percent more than for a local access. For sequential file access, with a disc latency of 15-20 msec, the V kernel is within 15 percent of the disc latency and the time for each page read is on a par with that of LOCUS.

The designers of the V kernel argue that streaming is unnecessary in a local area network because of the low latency, and using synchronous interprocess communication permits a large degree of concurrency, reducing the effects of network latency. However, a file server that provides *read-ahead* and *write-behind* can reduce the effective disc latency for sequential data, and this will increase the proportion of the delay in accessing the disc that is attributable to network latency. In these circumstances the benefits of streaming protocols outweigh any merits of the V kernel primitives.

The V kernel may be more desirable than, say, LOCUS in applications where its small size and the generality of the communication mechanisms are more important than the presence of higher level services, and their associated protection and synchronization needs. Additionally, the V kernel permits a significant versatility in the provision of services by the system for the user.

2.4.2 The WFS File System

In the previous section it was noted that the distributed V kernel provides the infrastructure for communication between the elements of an operating system, but does not impose any restrictions on the nature of those elements. With a similar objective, the WFS file system [97] supports a wide variety of applications by providing a concise set of commands for use in a distributed system. WFS

does not endeavour to provide a high level of functionality like that of LOCUS. It is closer in strategy to that of the Cambridge file system, section 2.3.1, in that it provides a set of rudimentary file operations and expects the client system to *enhance* these functions to achieve the desired standard of service.

It is the client system's responsibility to implement stream I/O and the directory system. WFS operations are concerned with page-level requests, and hence the behaviour of the file server is like that of a remote virtual disc. The commands are atomic and WFS will only return an acknowledgement once an operation is complete, minimising the risk of inconsistency at the file and page level and thereby improving reliability.

If the client does not receive a response within a reasonable period, it should re-transmit the request. WFS write operations are designed such that they may be repeated and still have the same effect. However, there is no protection against the re-ordering of requests. The client system may provide this by using sequence numbers in the private fields stored with each page. This use of atomic commands together with connectionless protocols obviates the need for WFS to maintain transitory state information between requests.

Every file has a file identifier (FID), which is a 32 bit unsigned integer. There is also a set of **file properties** for every FID, again with fields that are private to the client. WFS uses fields to store information about the state of the file (free, allocated, deleted, expunged), and sets a **dirty** bit on every operation that changes a file. This allows a client system to back-up those files that have the dirty bit set and then reset the bit.

The performance of WFS is restricted by its dependence on atomic page-level

operations. Where an application requires access to a contiguous set of pages, there is no mechanism provided by WFS to allow this to be optimized by, for example, streaming. The authors of [97] suggest that by spawning a process within WFS to issue the appropriate sequence of commands, performance would be significantly improved but the integrity of the virtual disc could be maintained.

WFS was developed to provide shared file access in the multiple workstation environment of **Woodstock**, but a number of other applications have been implemented.

2.4.3 Object Oriented Model

Amoeba is a capability-based, object-oriented distributed operating system [73, 99]. The system has been developed at Vrije Universiteit, Amsterdam, on a collection of Motorola 68010 Multibus computers interconnected by a 10 Mbps Pronet token-passing ring. Amoeba is destined for use with, what is described in [73] as, "a fifth-generation computer system architecture". This architectural model is based on a single circuit board, comprising processor and memory, that is used in varying quantities by a three-tier range of machines :

1. Personal computers, with high quality displays and a small number of these processor-memory modules.

2. Departmental computers containing hundreds of modules.

3. Large mainframes constructed from thousands of modules.

The essence of this model is that the same program should be capable of executing on any of these machines, exploiting the processing potential provided in each case. The only difference between the machines is in the *quantity* of processor-memory modules, not in the *type* of these modules. This model has much in common with the objectives behind the development of the transputer.

Traditional *process-based* approaches towards distributed operating systems do not map readily onto this architectural model because of their

> "emphasis on processes, and by inference, processors".[73]

Hence, Amoeba is concerned with *abstract objects*, each with associated *capabilities* which control access to those objects and determine the *abstract operation* that may be performed on them. An object may be implemented on one processor-memory module, but it is inherently location independent. Users execute functions on objects by issuing a request to the system, specifying the identity of the object and the operation to be performed, and in return receive a reply. The user has no need to know the location of an object, giving the system the flexibility of moving the object to different processors according to demand, and even varying the number of processors involved in the implementation of the object.

The architecture of the Amoeba prototype, figure 2–7, has four components :

- workstations for interactive tasks
- a pool of processors that may be allocated on demand
- specialized servers, dedicated to specific functions

- gateways to different sites.

The Amoeba processor pool differs from the Cambridge Procesor Bank, section 2.3.1, in that the processors may be dynamically allocated in groups to perform particular operations, such as compilations, but are then returned to the pool. Whereas the processors in the Cambridge Processor Bank are assigned, individually, to users for the duration of the terminal session.

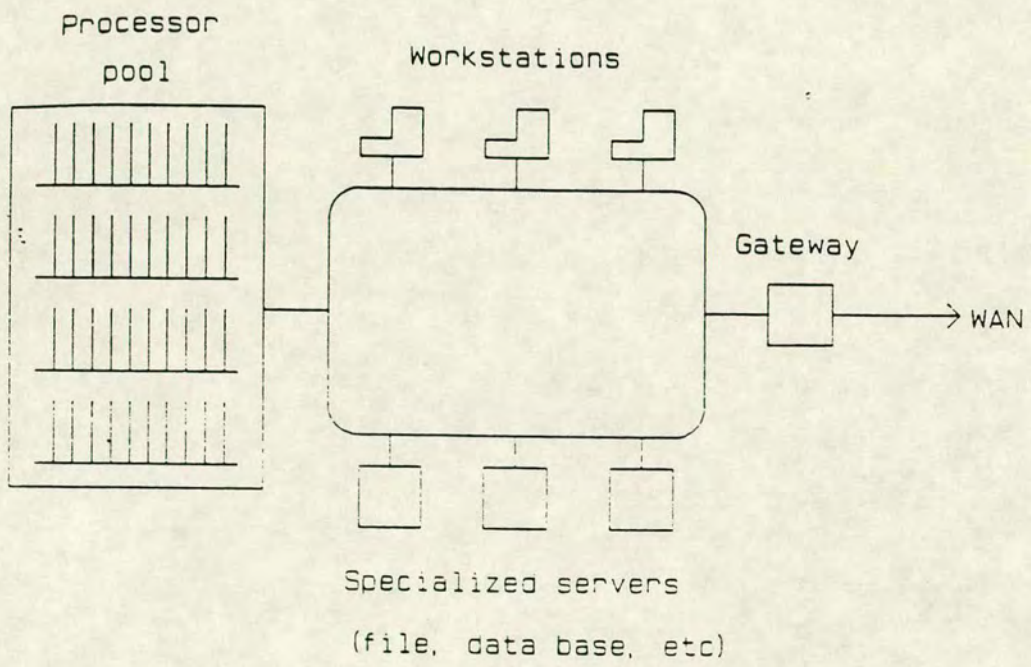

Figure 2–7: The Architecture of the Prototype Amoeba System

All of the machines in the Amoeba system execute the same kernel. This provides communication mechanisms but leaves the operating system functions to be supplied by user-level processes, some of which run on the specialized servers. As with the V kernel, section 2.4.1, users are able to replace system services

by their own, if desired. In turn, this makes development of new services very straightforward.

Communication Protocols

Amoeba is based on a *request-reply* style of communication with a simplified layering structure, as opposed to the ISO OSI *connection-oriented* protocols. The basic mode of operation is :

- the *client* executes *trans* to send a message of up to 32 kbytes and blocks, awaiting the result.

- the *server* uses *getreq* and *putrep* to receive the request and return the results.

This communication sequence is similar to V kernel message exchange, except that the size of the messages in Amoeba can be much larger. If a client does not receive a reply within the timeout period, it should retry; there are multiple instances of server processes and the network has a high reliability so a retry should only be necessary when the system is heavily loaded.

The Amoeba communication protocol is *blocking*, but there is provision for the transmission of *out-of-band* messages in the form of the **exception** message. An **exception** message will be transmitted to the server if the user presses BREAK on the terminal, and will cause the server process to terminate normally with a status of **request terminated**.

Capabilities

In Amoeba, as in other object-based systems, every object has an associated *capability*, for which each position in the bit-field indicates an operation that the holder of the capability may or may not perform. Every object is managed by a user-level service that interprets the capabilities for its objects. An Amoeba capability has four fields, figure 2–8.

Service Port : the address of the service that manages the object.

Object number : an identifier, internal to the service, that enables it to distinguish between its objects.

Rights field : a bit-field, where each '1' indicates a permitted operation.

Check field : a random number, used to authenticate the capability.

SERVICE	OBJECT	RIGHTS	CHECK

Figure 2–8: An Amoeba Capability

When an object is created, the server selects an identifier and a new random number, then encrypts the **rights** field (initially all '1's) with this random number and a known constant to form the last two fields of the capability. It is possible for the holder of the capability to give it to another user with the same access rights, but impossible to grant reduced or enhanced rights without returning the capability to the server and obtaining a new one. The use of the random number

results in a limited degree of protection for the capability, but in [73] a more elaborate scheme is described that involves *one-way functions.*

The Amoeba distributed operating system seems to exploit the potential of the object-oriented model, and exhibits a significant degree of versatility. It would appear that object-oriented systems may operate in a distributed environment quite successfully. However, it remains to be seen whether or not they are much better than process based systems when used in the "fifth-generation computer system architecture" that is the target of the Amoeba system.

2.5 Campus Computer Networks

2.5.1 Introduction

An expanding number of universities are becoming involved in the construction of campus-wide networks of computers. In such installations, the number of machines is very large, running into 100s or 1000s, and the computers themselves are supplied by a number of different manufacturers. It is clear that the development of a campus computer network is an onerous task, for a number of reasons :

- The variety of types of computer may result in problems due to incompatibilities between these machines.

- The wide area of a campus provides technical difficulties in the forming of a single unifying network.

- A primary objective of campus computer networks is to broaden the accessibility of the machines and encourage a much more substantial user base to exploit the potential of the system. However, this also means that the software provided by the system needs to be expanded to satisfy the needs of the new users.

- The integration of computing resources from different departments into a single system introduces additional organizational requirements. The upkeep of the system is of special concern, particularly because of the various

computers having a multiplicity of owners, users and most especially maintenance contracts.

The efforts of research groups to overcome these obstacles are made more worthwhile by the widely held regard that a campus computer network can greatly improve the operation of a university through the enhanced communication and information processing abilities that it provides.

The preceding sections of this chapter have been concerned with systems that have been developed for operation within a single department. It could be argued that both MISS and the Cambridge Distributed system incorporated a means of gaining access to the campus central computing facility. However, these systems did not provide for the distribution of the campus computing resources amongst the participating departments.

In contrast to the strategy of enabling the user of a small computer to gain access to a conventional mainframe for temporary large scale processing requirements, this section is concerned with a more decentralized approach to satisfying users' computational needs. Primarily, this section will concentrate on the **Andrew** distributed personal computing environment that has been developed at Carnegie-Mellon University. The approaches taken by the SPICE project, Dartmouth College and Brown University will also be discussed in comparison with the Andrew system.

2.5.2 Motivation

The possibilities afforded to universities by the formation of campus-wide computer systems are tremendous. In [105], Doug Van Houweling describes the university as the "quintessential information processing organization", but he observes that "the demands of rapid expansion in knowledge are placing immense pressures on student time". Van Houweling considers the potential for relieving this pressure :

> "The advent of communications networks and networked computing is on the verge of making audio and pictorial information more easily manipulable ... information gathering (through access to library resources and computer data banks), information processing (via text editors and database editors), and communication (through electronic mail and bulletin boards) can reduce the time spent in routine tasks." [105]

Van Houweling's student-oriented perspective of campus computer networks is similar to that taken by Brown University in their **Scholar's Workstation Project**. Martin Michel describes the goal as being to

> "help scholars in their daily knowledge work." [65]

This inclination is sympathetic to the practice at Brown University, where the emphasis is on undergraduate teaching. Dartmouth College and Carnegie-Mellon University envisage a broader purpose for campus computing, described by William Arms as

"to improve the quality of academic teaching and research." [7]

In [70], the potential effects of **Andrew** on university education are described as being concentrated in four main areas :

1. Computer-aided instruction

2. Creation and use of new tools

3. Communication

4. Information access.

Whilst the introduction of campus computer networks can extend the facilities provided in these four areas, the authors of [70] stress that there is an increase in functionality rather than any replacement of existing methods. Taking communication as a clear example :

> "We do not expect computer mediated communication to supplant the more traditional methods, but it will broaden and deepen the community's ability to communicate." [70]

2.5.3 Andrew

Background

In October 1982, the **Information Technology Centre** (ITC) was established in a collaborative effort by IBM and Carnegie-Mellon University. The task of

the ITC was the design and development of computing resources to satisfy the anticipated requirements of the university. The resulting system, **Andrew** [6,70], is an ideal example of what can be achieved in a campus computer network. This is in part attributable to the considerable computing expertise that was involved in the creation of Andrew, but is also the result of the experience gained from other systems. Noteworthy systems that are acknowledged as being instrumental in guiding the course of Andrew's development are the MIT Athena Project, the Xerox Alto System and the SPICE Project.

The **Xerox Alto System** [101] demonstrated how technological advancements in the design of personal computers could be exploited to improve drastically the transfer of information between user and computer. Central to these developments was the integration of a pixel-addressable display with a mouse. The combination of these two elements allowed operations to be effected and represented by the manipulation of graphical items (*icons*) using the mouse.

The ideas introduced by Xerox have been widely adopted, to the extent that **Window, Icon and Mouse Packages** (WIMPs) are commonplace. IBM has adopted a form of this interface in the operating system for its new personal computers, the PS/2s, and thereby endorsed the acceptability of WIMPs for commercial usage.

The ITC saw this form of human-computer interface as a means of achieving a uniform application-level interface to a number of software packages. A high degree of uniformity in the presentation of different software was considered essential in order to encourage the large new user community to exercise unfamiliar tools. The user interface of Andrew is considered further in section 2.5.3.

The **Scientific Personal Integrated Computing Environment** (SPICE) project [9] was developed in the Computer Science Department of Carnegie-Mellon University by researchers who later joined the ITC. It is therefore quite reasonable to expect some of the ideas behind SPICE to have been incorporated in the Andrew system.

Technical Features of the Personal Computers

Perhaps the most noticeable common feature between SPICE and Andrew was the hardware specifications of the personal computers. For SPICE the requirements were :

1. A micro-programmable processor capable of
    - executing 1 million macro instructions per second (1 MIPS).
    - providing for the efficient execution of Pascal and LISP programs, with sufficient control store memory for more than one resident instruction set.
2. Virtual memory with
    - 1 megabyte of real memory.
    - 100 megabytes of local disc storage.
3. A powerful user interface, supported by
    - high-resolution bit-mapped display with a million pixels.
    - keyboard.

- pointing device.
- provision for audio capabilities.

This hardware specification was drawn up in the belief that, by the mid-1980s, such a system would cost about $10,000 and would subsequently be affordable to students.

The SPICE project adopted the Three Rivers Corporation Perq for development, but endeavoured to minimize hardware dependencies because the Perq did not meet the price/performance needs. For the Andrew project, the SUN workstation was chosen for the development machine since it possessed most of the more important requirements.

The choice of a personal computer that provides a flexible user interface with a price/performance ratio appropriate for purchase by students is a characteristic of campus computer networks. The computers selected by Brown University and Dartmouth College differ from SPICE and Andrew because both these projects predate the work at Carnegie-Mellon and they required a shorter period for introduction of the new machines.

The Dartmouth Personal Computer Project specified a *preferred* personal computer, to be supplied to faculty members and purchased by a large number of students. The **Student Package** comprised an Apple Macintosh 128k computer, with MacWrite, MacPaint, TrueBASIC and DarTerminal software supplied for a total cost of $1,285. This is comparable to the average annual expenditure by students on books. Brown University also chose the Apple Macintosh for their **Scholar's Workstation Project**, but additionally IBM PC/RTs have been used.

Structure of System

At Dartmouth College, the personal computers were expected to be used in two ways :

1. **As a computer :** for

    - word processing
    - graphics
    - programming in BASIC and Pascal
    - course materials

2. **As a terminal :** for

    - electronic mail and shared data
    - library catalogue
    - time shared computers.

The overall structure of the system was based on the **Kiewit Network** designed by Stanley Dunten, figure 2–9. The baseband network was not fully developed because the emphasis remained on the provision of medium speed services provided by time shared computers. There were no dedicated fileservers, but files could be transferred between machines and the Apple Macintoshes have floppy discs. The simplistic nature of the Dartmouth system is in stark contrast to the more advanced nature of the Andrew system.

The Andrew system is based on a computing model that combines personal computing with timesharing, figure 2–10. At the centre is **VICE** (the "Vast, Inte-

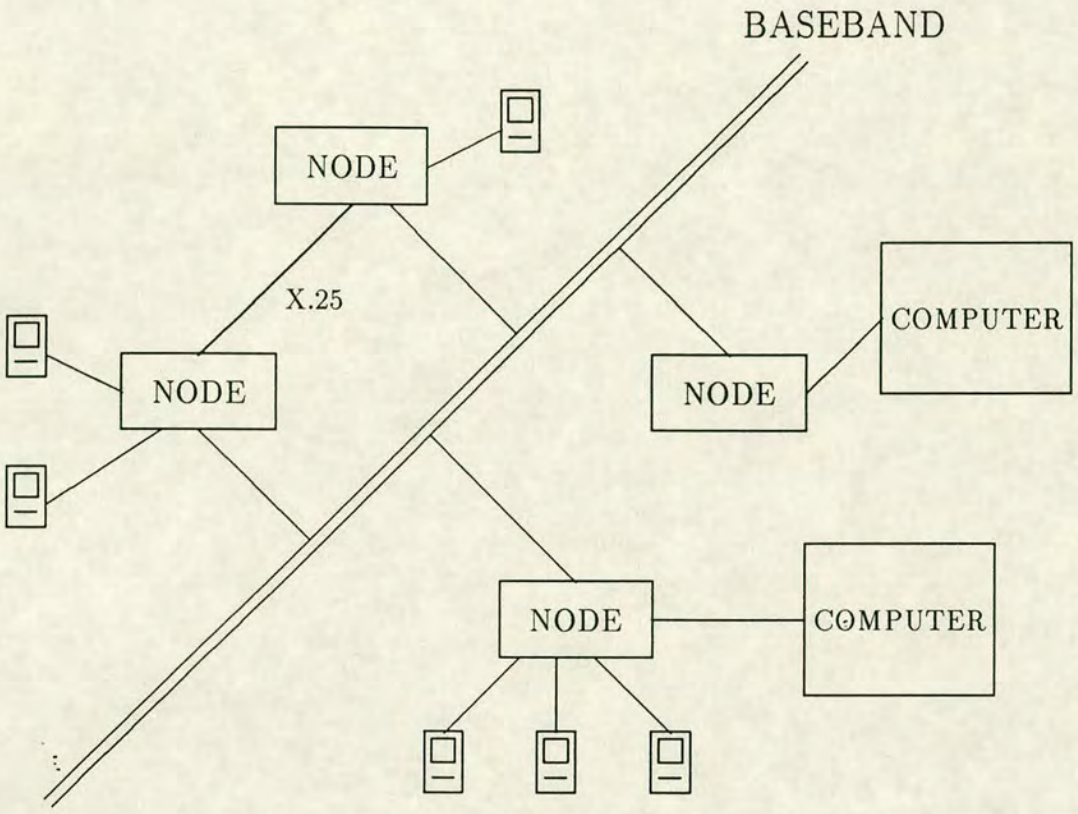

Figure 2–9: The Kiewit Network

grated Communication Environment"), which forms the backbone of the system, providing both communication and processing resources. VICE provides a uniform shared name space for files, so that users may access files in the same fashion, independent of the location of the workstation.

The **VIRTUE** workstations connect to VICE to provide users with a powerful human-computer interface and supply computational resources. VIRTUE stands for "Virtue is reached through Unix and EMACS". Specifically, it is the Berkeley version of Unix that forms the basis for the system. The main reason for this is that in academic circles Unix 4.2 BSD has become a *de facto* standard. Consequently, not only were most of the developers familiar with it, but also it has been

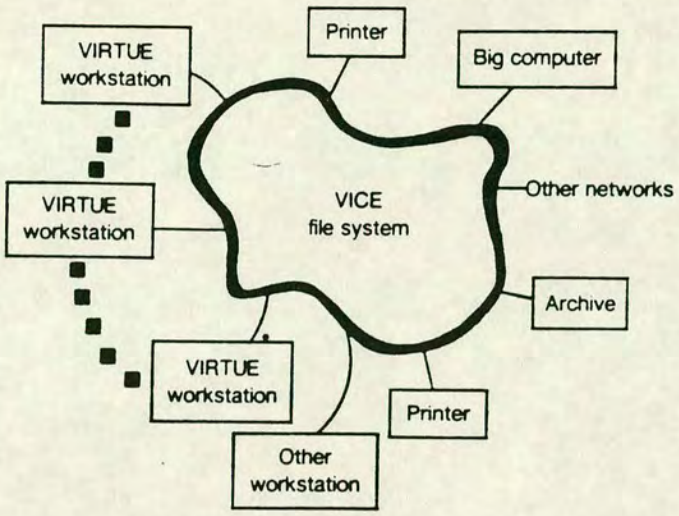

Figure 2–10: VIRTUE and VICE

implemented on a multitude of machines. This allows software developed using Unix 4.2 BSD to be readily portable to many other machines.

As noted by the authors of [70], the designers of Andrew "distanced" themselves from the "mainstream personal computing world", by opting for powerful Unix-based workstations instead of IBM PCs and Apple Macintoshes, as chosen by Brown University and Dartmouth College. The benefits of virtual memory and a good programming environment were considered preferable to the advantages of being able to use the commercial software available for IBM PCs and Apple Macintoshes. However, the anticipated convergence of scientific workstations and commercial personal computers should eventually result in the availability of commercial software on Andrew.

The integration of the various components that collectively form Andrew is illustrated in figure 2–11. The partitioning of VIRTUE and VICE allows new workstations to be added to the network with a minimum of effort to allow cooperation

with VICE, and also without disrupting the existing parts of the system. Additionally, the VIRTUE-VICE interface is the *boundary of trustworthiness.* No user programs reside within VICE, and hence it forms a secure environment. However, workstations are owned by individuals who are capable of modifying the hardware or software at a whim.

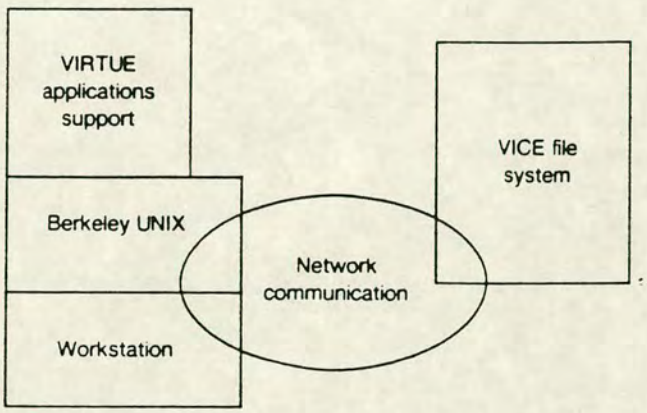

Figure 2–11: Components of Andrew

Network Communication

The basc topology of Andrew, figure 2–12, shows how VICE is formed from a collection of clusters, each with a local server, and all of which are interconnected by a backbone network. The *internet* comprises token rings within buildings, with fibre optic links to 2 central routers. The *bridges* act as routers and traffic filters, interconnecting Ethernets and ProNet token rings, but assuming TCP/IP protocols. The TCP/IP protocols were chosen partly because of the association with the ARPA network, but mainly due to the existing implementations of these protocols in Unix 4.2 BSD.

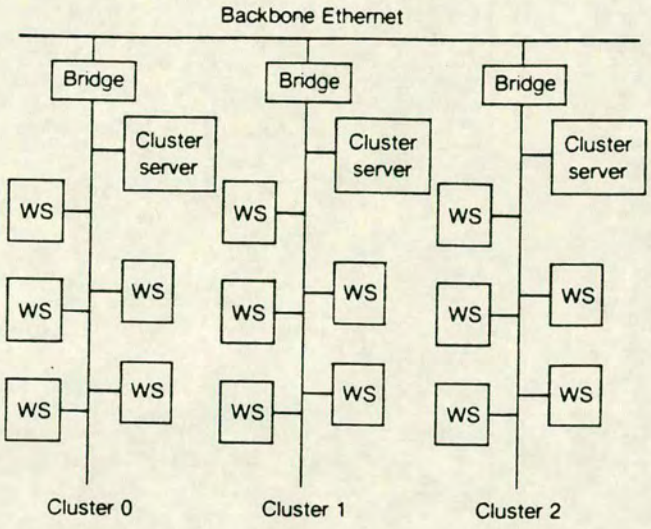

Figure 2–12: Topology of Andrew

At a higher level, the VICE-VIRTUE interface is implemented using remote procedure calls for data and control transfer. There are inbuilt authentication capabilities and the option to use encryption for secure communication.

Dartmouth College concentrated on the provision of medium speed services, with personal computers communicating via a local switching node. These nodes can support up to 56 nodes, connected by AppleTalk, RS-232 or X.25. The protocols used over the network are

- asynchronous ASCII PAD
- TcFace for Dartmouth timesharing
- X.25, levels 2 and 3
- AppleTalk.

Brown University placed a greater emphasis on the development of a campus wide broadband network to interconnect all of the personal computers. Additionally Syteck Localnet/20 was used for terminal-host connections, as well as AppleTalk, IBM PC-net and 10 Mbps Ethernet.

File System

Application programs on VIRTUE workstations view the file name space as partitioned into *local* and *shared* areas. However, the vast majority of files accessed by users are in the shared area, thus providing the uniformity of access mentioned in section 2.5.3. The shared name space is reached by *mounting* onto the node **/cmu**. Subsequently, all accesses to files in this directory are referred to the appropriate fileserver.

Commonly accessed system files are translated to remote accesses through the use of symbolic links. The choice of symbolic link may vary according to the machine being used. For example, **/usr/local/bin** will be a symbolic link

on a SUN to **/cmu/unix/sun/usr/local/bin**

on a VAX to **/cmu/unix/vax/usr/local/bin**

This reduces the local disc space needed, simplifies the release of new software and provides support for heterogeneous workstations. This use of symbolic links is similar in effect to the LOCUS hidden directories, section 2.3.2.

When an attempt is made to access a shared file, the request is passed to a local process, **Venus**, that represents the file system and manages the temporary

storage of files and communication with remote file servers. If the file is not held locally, a copy of the entire file is transferred from the file server and then held in a local cache. When the file is subsequently closed, if it has been changed, a copy is returned to the file server. In this way, write-through caching of files is achieved.

Entire file caching results in improved performance, due to fewer interactions with the file server, and simplifies the operation of Venus. However, this scheme is only workable when the workstations have sufficient local disc storage to be capable of caching all of the user's working set of files, and is undesirable when workstations do not have local discs. Files of up to 5 megabytes have been used in this way, but [70] claims that the majority of files are quite small, adding the comment that large databases cannot be accommodated with the present system.

Workstations that have no local discs use a "PC-server", which is a process running on another Unix Machine. The protocol used for communication between a disc-less workstation and a server is called **SNAP**. Primarily, however, Andrew is intended for use by workstations with local discs.

The hierarchical name space is partitioned into sub-trees, each of which is managed by a single server, designated its **custodian**. The custodian for a sub-tree is responsible for servicing all requests for files within it, but frequently accessed sub-trees may have read-only replicas at other servers, created by system managers using *cloning*. It is assumed that these read-only replicas and the custodian for a sub-tree change very rarely.

User Interface

The ITC had two primary objectives when creating the workstation software :

1. To provide tools that allow application developers to exploit the graphical capabilities of the workstations.

2. To encourage consistent application-user interfaces, in order to reduce the degree of application-specific knowledge needed by a user for extensive use of the system.

The second of the ITC objectives is common to one of the important aspects of IBM's Systems Application Architecture (SAA), section 2.2.3. The recent announcement of the availability of the OS/2 Presentation Manager is viewed as an important element in developments of the personal computer in the immediate future. It is understandable that the ITC placed so great an emphasis on the rôle of the user interface of the workstation software.

The structure of the software on a VIRTUE workstation is illustrated in figure 2–13. It can be seen that the **window manager** has a very important function in the operation of all applications. It manages the display by dividing it into independent rectangular areas, each of which is attached to a process. The window manager receives input from the keyboard and the mouse, and directs this input to the process associated with the window displaying the mouse cursor.

As illustrated in figure 2–13, above the window manager is the **base editor tool kit**. The tool kit contains a collection of data types, each of which has a

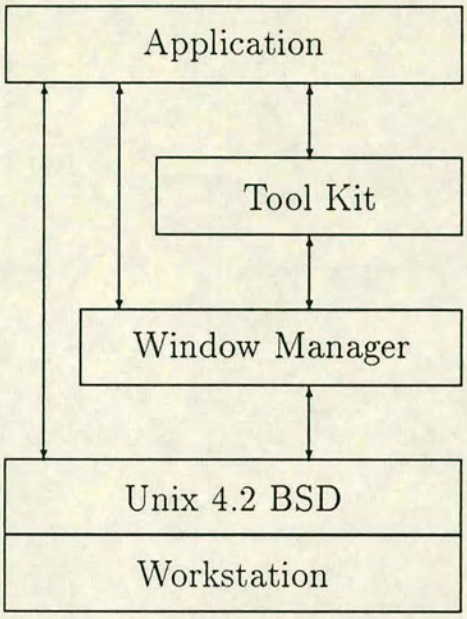

Figure 2–13: VIRTUE

programming interface and a well-defined **user-interface**. Use of the tool kit by an application provides the following advantages :

- it has a higher level interface, thereby reducing the complexity of controlling the display

- applications using the tool kit are more likely to have consistent user interfaces

- the performance and behaviour of software may be enhanced more easily by manipulating individual data types than by attempting to upgrade a number of application programs.

Applications that have been developed using these facilities include a text editor with dynamic formatting, drawing editors, mail and bulletin board browsing programs and the Tutor programming language.

Experience

Carnegie-Mellon University have concluded that the introduction of small personal computers has been very successful. The user community is, in general, happy with the system, but complains about the unreliability and low performance. The authors of [70] consider these problems to be typical of prototype systems and envisage improvements in the long term. There were noticeable difficulties in the integration of the new system with existing computing usage, but the implementation of a "federal" network, involving 17 organizations, has been very beneficial.

At Dartmouth College, a more extensive study was made of the effects of the introduction of the campus computer network. The academic impact was marginal :

- most of the course usage was by courses that had already used the existing computing resources
- lecture notes improved
- the average length of papers increased by 50 per cent.

The use of the general purpose timesharing facility was drastically reduced, but the load on the VAXs increased. This led to the conclusion that the Apple Mac-

intoshes were ideal for small tasks, but not for more computationally demanding jobs.

2.6 An Integrated Distributed System

2.6.1 The Apollo DOMAIN

The designers of the **Apollo DOMAIN** regard it as "the third phase in the history of computer architecture", with the previous two phases being batch processing in the 1960s and timesharing in the 1970s. Their objective was to provide the performance and responsiveness of dedicated minicomputers with the advantages of resource and data sharing provided by traditional mainframes. This target was to be reached by fully distributing both access and processing, table 2–2, using an integrated network of powerful personal workstations and server computers. The nature of the Apollo systems is reflected in their name - DOMAIN stands for **Distributed Operating Multi-Access Interactive Network.**

	ACCESS	PROCESSING
Batch	centralized	centralized
Timesharing	distributed	centralized
Apollo DOMAIN	distributed	distributed

Table 2–2: Comparison of Access and Processing Capabilitiies

"A DOMAIN system is intended to provide a substrate on which to build and execute complex professional, engineering and scientific applications." [60]

In [60], the elements of this substrate are listed :

- A powerful processor with a large virtual memory to enable mainframe applications to be supported by each user's node.

- A high resolution display which, in conjunction with the powerful processing capabilities of the node, could provide a user interface to "maximize user productivity".

- A high bandwidth local area network for communication and resource sharing.

These features of the Apollo DOMAIN system are common with the requirements of the SPICE and Andrew projects at Carnegie-Mellon, section 2.5.3. However, the Apollo system was intended for more sophisticated installations and its price range reflects this fact; the price of Apollo DOMAIN workstations is beyond that specified for the campus networks discussed in the previous section.

Architecture

Before concentrating on some of the more interesting features or the Apollo DOMAIN system, it is useful to consider the operational model that forms the basis for the system architecture. The authors of [60] describe the DOMAIN as being

based on an "integrated model" as opposed to a "server model". The Cambridge Distributed Computing System is taken as an example of the latter, where there are nodes that are dedicated to the provision of specific services. In contrast, the DOMAIN architecture is formed from nodes that each have a full complement of standard software. This software may be configured such that the node then behaves as a dedicated server. In the same manner, many of the nodes that are directly accessed by users are capable of supplying all of the services that are needed for the majority of tasks.

The DOMAIN nodes do not depend on the presence of the network for their operation, but are capable of autonomous usage. Utilization of the network is encouraged by standard mechanisms that support cooperation and sharing between the nodes. Fundamental to the integration of the nodes is a distributed **Object Storage System** (OSS) and a network wide **Single Level Store** (SLS).

Object Storage System

The DOMAIN **Object Storage System** (OSS) provides a flat namespace for accessing and storing *objects*, which are containers for abstract data types, such as text, mailboxes, directories and executable modules. Associated with every object is its length, type, an access control list and attributes related to the storage of the object. Objects are distinguished by **unique identifiers** (UIDs), enabling uniform access throughout the network because above OSS only UIDs may be used to address an object[5].

[5]Both the type and the access control list for an object are denoted by UIDs that in turn refer to other objects.

The OSS has two parts, one for controlling access to local objects and the other for remotely located objects. The former is mostly concerned with a **volume table of contents** (VTOC) system data structure that translates object references to disc block addresses. The second layer of OSS allows all objects to be accessed in a location-independent manner, using the local or remote accessing procedures, as appropriate. For remote objects, the first stage is to locate the node on which the object currently resides by means of a heuristic algorithm and a **hint file.**

Once the object's location has been determined, the node issues a request to the *remote paging server* process of the node holding the object. Every node has remote paging and file server processes that satisfy **(UID, page number)** requests for objects on that node. Since all remote accesses are achieved in this manner, only the holding node needs to be aware of the actual disc addresses involved.

Single Level Store

The DOMAIN network wide **Single Level Store** (SLS) is comparable in purpose to the one level stores of some existing centralized systems. The SLS enables a uniform mechanism to be used for accessing objects, irrespective of whether the objects are resident in local memory, on an attached disc or located on another network node.

Objects are introduced to a process address space by presenting their UIDs. Thereafter, the pages of objects are demand paged over the network as they are required. Virtual memory is supported in the CPU by **dynamic address**

translation (DAT) hardware, that allows up to 128 processes to exist, each with 16 Mbytes of virtual memory.

The most important feature of SLS is the **concurrency control mechanism**, which detects whenever an attempt is made to access different versions of a single object, constituting a *concurrency violation.* In an environment where nodes are sharing access to objects, with the possibility that many nodes will be holding different portions of that object at any one time, some synchronization of these caches is essential in order to retain consistency in the storage of objects.

The SLS achieves control over concurrency through the presence of a **data-time modified** (DTM) attribute for each object. This is a time stamp based version number, and is attached to every page of an object. Whenever an object's page is requested, the node holding the object records the DTM of the version issued to the requesting node. Any updates performed by the new node will be accompanied by changes to the DTM. So, when this node tries to write back the amended pages of the object, the home node can readily detect whether a consistencey violation has occurred.

Network

The authors of [60] describe the Apollo DOMAIN local area network as the "system integration point", analogous to the common backplane bus in some systems. The nature of the network remains the same in different implementations of the DOMAIN, but flexibility in the construction of the nodes has allowed Apollo to exploit new technology, as it became available.

The network requirements were :

- high transmission speed
- decentralized control
- fault tolerance
- support for efficient fault locating.

A 12 Mbps baseband token passing ring was chosen, with extra reliability achieved by incorporating a relay in each node's repeater that closes when a node crashes or the power fails. This results in a "star-shaped ring" configuration and "quick-disconnect" hardware is used for the node connection to facilitate simple removal and replacement of nodes in the event of failures.

The network design includes features to improve the efficiency of data transfer over the network. These include :

1. There are 2 acknowledgement fields interpreted by the transmitter.
2. The ring hardware supports virtual memory demand paging, section 2.6.1, by discriminating between the header and data portions of a packet, and copying each into separate areas of memory using different DMA channels.
3. Every packet has a *type field* and the ring controllers have corresponding *type mask* registers to filter the messages directed to the node.

The self-clocking biphase encoding used in the ring means that broken or faulty links are easily detected. The node *downstream* to a broken link notifies all other nodes on the ring. Combining the information from a network topology map,

produced by the DOMAIN, with the notification of link failures simplifies the location and repair of faulty links.

Inter-Process Communication

The low level **inter-process communication** (IPC) mechanism is based on an unreliable datagram service that uses Unix-like *sockets*. These sockets are identified by (node ID, socket ID) pairs, and appear to the processes as queues of network packets. The sockets have the standard two types :

Well-known : statically assigned to system services, such as the file server. The socket ID for well-known sockets is the same on all nodes.

Reply : dynamically allocated for the receipt of replies from network services.

The Apollo DOMAIN network protocols [78] follow the LOCUS example of being *problem oriented*, such that the protocol used by each operation is tailored to its needs. A generic classification may be made for these protocols with respect to the level of reliability achieved.

1. Idempotent operations may be guaranteed through repeated retries until the operation is successful. There need be no state information or connection.

2. Operations that have an associated state may be made idempotent by use of a transaction identifier. Every new request has a new transaction ID, but retries use the same ID so that they will only be performed if the previous request was not satisfied.

3. Synchronized database updates are achieved using a *request-reply-acknowledge* (RRA) type of protocol. However, the DOMAIN exploits the hardware acknowledgements provided by the ring to obviate the need for an explicit software acknowledgement.

The DOMAIN socket abstraction is cheap because of its simplicity and the absence of guarantees about the quality of service. The service is made more reliable by the end-to-end protocols used, and since these protocols are application specific, the resulting communication system is very efficient.

A higher level IPC service is provided in the form of *mailboxes* (MBX). The MBX object is a storage container for undelivered IPC messages and the service provides a full-duplex virtual circuit service. Nodes form connections to a mailbox simply by specifying its UID. The MBX remote protocol filters out duplicate or mis-ordered packets as part of the service.

In [60], the authors compare the DOMAIN SLS and IPC facilities in terms of the computational models to which they correspond :

1. Moving the computation to the data, as with message passing systems such as the IPC.

2. Moving the data to the computation by transparent data access of the form of the SLS.

As will be apparent from the description of the DOMAIN given here, the design emphasis has been on the SLS support for network paging. However, the DOMAIN uses IPC for the name server for reasons of efficiency. Additionally,

IPC is viewed as preferable when the data needs to be protected or has a complex internal structure. In other circumstances, the authors of [60] argue that the advantages of SLS easily outweigh the merits of IPC because of the simplicity of the model.

2.6.2 The CRAY Station Software Service

Cray Research have developed a system to allow the users of Apollo DOMAIN workstations to gain access to the computational power of a Cray mainframe, whilst still benefitting from the amenable environment of the workstation. This system is the **CRAY Station Software Service for Apollo DOMAIN** [4, 5]. The service allows both interactive and batch access to a CRAY-1 or CRAY X-MP mainframe, and provides for exploitation of the graphics capabilities of the Apollo workstations.

Figure 2–14 shows a configuration incorporating a number of nodes on a DOMAIN network, a Cray system, an NSC HYPERchannel and the devices needed to bridge between the two parts of the system. The A130 and A400 are HYPERchannel adaptors, the DSP-80 is a DOMAIN node that acts as a communications gateway.

Facilities

From the DOMAIN multiple process environment, it is possible to create a number of simultaneous interactive sessions with COS, the Cray operating system, by typing **CINT** at the workstation. Within an interactive session, use may be made

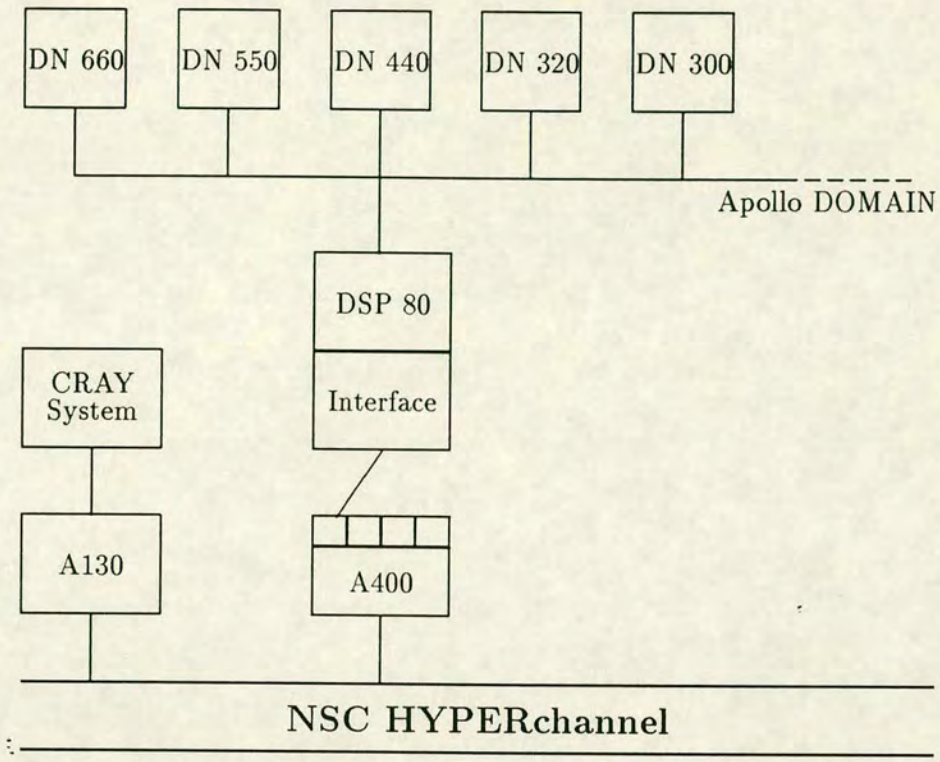

Figure 2–14: A Typical Configuration

of a Cray program development environment. Additionally, the service is capable of providing interactive graphical output from the CRAY.

A user may submit a batch job to the Cray system using the station command **CSUBMIT** <pathname>. The job running on the CRAY is able to access and create files on the DOMAIN network using further station services : **FETCH**, **ACQUIRE** and **DISPOSE**.

The transfer of files between the Apollo DOMAIN and the Cray system is designated *dataset staging*, and may be performed using the **CSAVE** station command. For the system manager, a range of commands is provided to allow the activation, deactivation and normal operation of the system to be controlled.

Software Structure

The components of the station software are modular and most are written in FORTRAN, with some in Pascal. The components communicate via the DOMAIN high level IPC facility, mailboxes, section 2.6.1. The structure formed by these software components is illustrated in figure 2–15.

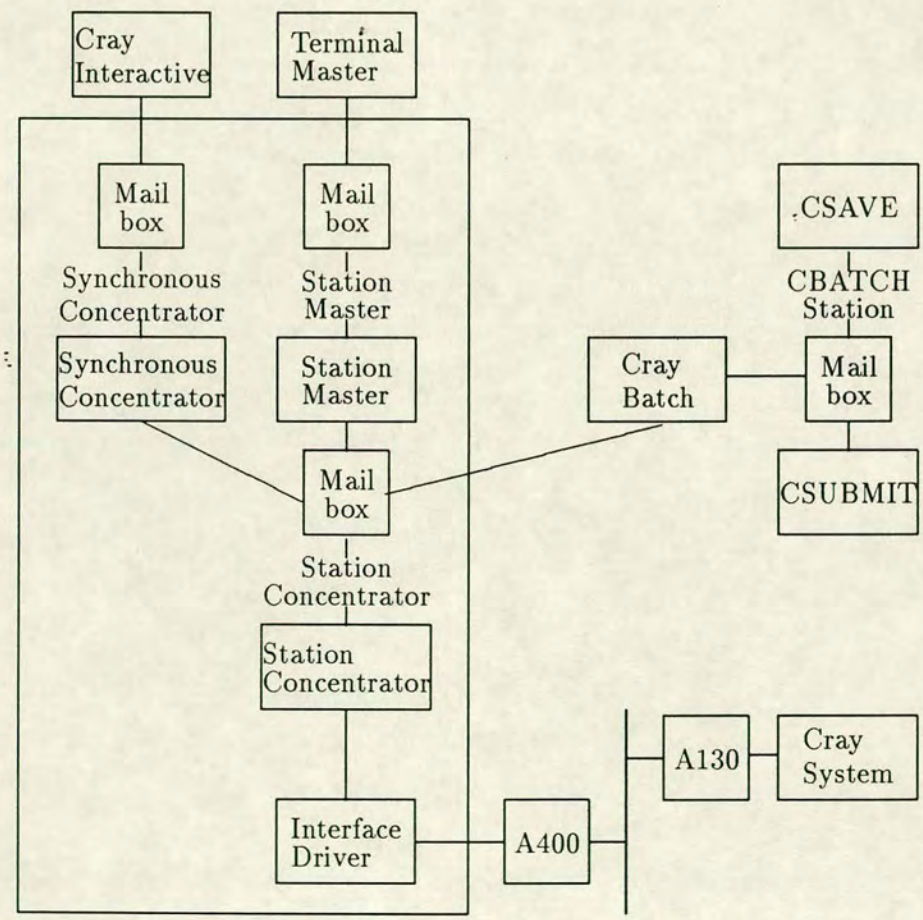

Figure 2–15: Structure of Station Software

The **Station Concentrator** calls the **Interface Driver** to transfer data to and from the Cray system. The other processes communicate with the Station

Concentrator using an Apollo mailbox. The **Synchronous Concentrator** coordinates all synchronous requests and replies.

The **Cray Interactive Module** (CINT) handles all interactive communication with the user and also supports the use of the Apollo workstation screen for displaying graphical output. The **Station Master** manages the operation of the station. The **Cray Batch Module** (CBATCH) runs on all nodes that access the station. CBATCH communicates with **CSAVE** and **CSUBMIT**, via a mailbox, to obtain parameters for dataset staging and job submission respectively.

Observations

The use of mailboxes for most of the communication between software modules means that the CRAY Station Software Service integrates comfortably into the Apollo DOMAIN. The submission of batch jobs is quite straightforward and file transfers are performed very easily. The support for interactive operations is much more restrictive, but this is understandable because of the wildly different natures of the two systems. In addition, the ability to exploit the Apollo workstations' graphical capabilities using programs executing on the CRAY is extremely useful.

Taking a different perspective, though, it could be argued that the full potential of the workstations, and for that matter the CRAY, is underused. The workstations would appear to be treated as super-intelligent terminals with an RJE (remote job entry) capability. The CRAY is burdened by administrative chores, such as accounting, scheduling and resource allocation. Furthermore, the station forms a bottleneck because all communication between the DOMAIN network and the HYPERchannel involves the one node.

2.6.3 Merits

The latest Apollo workstation, the DOMAIN Series Super 4000, looks set to be adopted by companies requiring powerful CAD/CAM facilities. The integrated nature of the DOMAIN network, together with the substantial range of software and experience that is available, and combined with the possibility of exploiting supercomputers for more intensive processing, must serve as a model for high performance computer installations. It remains to be seen whether or not the approach taken by Cray Research will be followed by other supercomputer and superminicomputer manufacturers.

2.7 Concluding Remarks

This chapter has looked at a variety of multiple computer systems with a wide range of features. Although none of these systems embodies all of the characteristics of the target system, introduced in the first chapter, they contribute a number of facets that are pertinent to this direction of research.

The next chapter opens with a description of a network structure that epitomizes many of the features of the systems described in this chapter. The chapter goes on to concentrate on a subset of this structure and defines a triadic network model to represent the essence of this subset.

Chapter 3

A Network Model

3.1 Introduction

The central element of this chapter is the definition of a Triadic Network Model of communication and cooperation between network devices. The model describes the interactions between three classes of device in a manner that integrates the features of heterogeneous machines to form a unified system.

Background

The introductory chapter to this thesis presented a target system composed of multifarious computers cooperating in the provision of a powerful and versatile computing environment for the users of the system. In the following chapter, a study was made of existing multiple computer systems in order to foster a deep awareness of the policies and concepts that are predominant in this field. Many of the topics contribute towards the development of the target system, and they will be referenced in subsequent sections of this thesis.

However, whilst some of the systems discussed in chapter 2 have achieved good standards of close participation between multiple computers, the machines involved tended to be of similar power and ability. With the exception of the CRAY Station Software Service for the Apollo DOMAIN, section 2.6.2, the inclusion of powerful processors was usually achieved by providing a terminal emulation facility for access to the conventional time-shared operating system of the mainframe. The IBM PC/VM Bond, section 2.2.3, possesses features that are characteristic of these facilities.

The Apollo DOMAIN is a fine example of an integrated distributed system. The CRAY Station Software Service is a fully participating member of this system, thanks to its use of the DOMAIN high level IPC facility, the mailbox (MBX). Ultimately, the involvement of the Cray system is limited by its own operating system, COS. This is because COS is designed for use in an environment of computers comprising the CRAY and a number of support machines. COS is not readily amenable to the *community spirit* of the Apollo DOMAIN. Hence, the achievement of the CRAY Station Service is commendable in that it enables access to the CRAY for batch processing from within the DOMAIN environment.

Bringing Together Workstations and Mainframes

It would appear that a great leap is needed to achieve a larger involvement by powerful processors in a network of computers with inferior processing power. The disparity in the computational ability and architecture of workstations and powerful processors is only part of the difficulty. Of greater significance is the difference in the way in which the two types of computer have been used. Per-

sonal computers are owned and used by individuals, but can communicate over networks and share access to peripherals and storage devices. Special purpose super or superminicomputers were usually connected directly to general purpose time-sharing computers, which acted as *front-ends*. More recently, mainframe computer networks have been formed, as described in section 2.2.1.

The two types of network have been brought together to form a configuration that is similar to that of figure 3–1. There is a distinction between the communications network used by the workstations and small computers, called the *front-end* network, and the high speed interconnections between the mainframes and their peripherals, the *back-end* network.

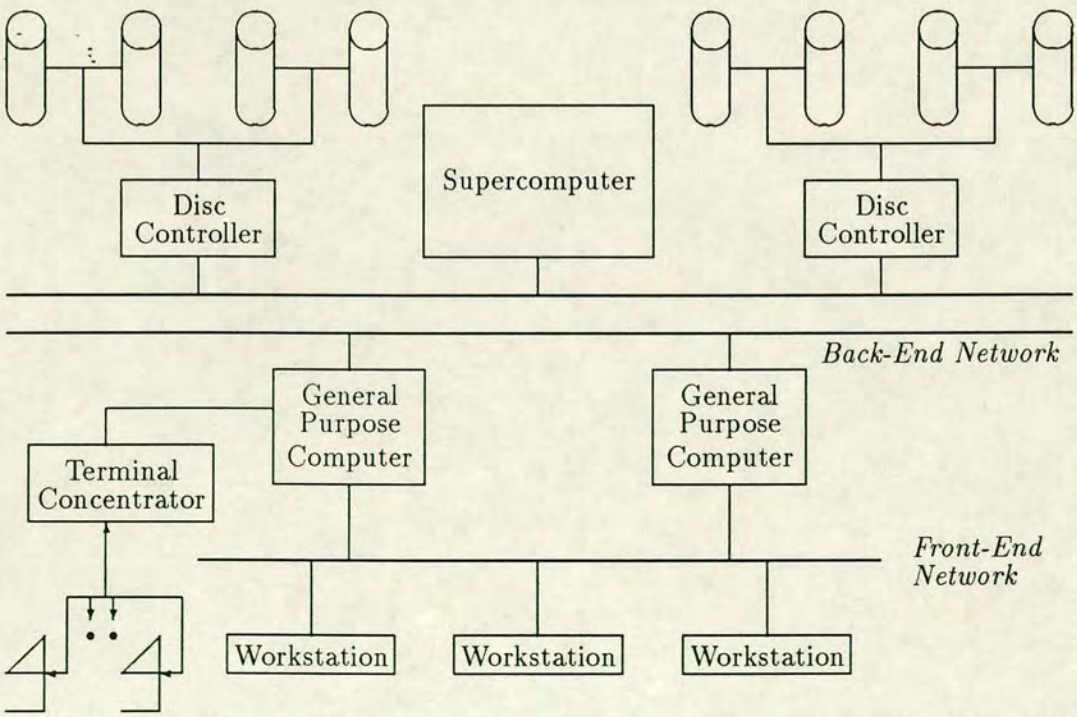

Figure 3–1: Front-end and Back-end Networks

The partitioning of the network into *front* and *back* portions is appropriate to

the needs of the corresponding network nodes. The workstations and general purpose computers frequently number hundreds and are disparately sited. Their use of the network is for small packets, and data rates of about 10 Mbps are adequate. Indeed, many installations have found that only a small percentage of the network capacity is exploited. In contrast, back-end networks have a small number of nodes, but individual operations may require the transfer of large quantities of data at I/O channel speeds, such as 50 Mbps.

In [48], the incremental cost of adding a new device to the network is used to contrast the two types of network. The authors claim that, to be cost effective, the incremental cost should be in the order of one tenth that of the device itself. For workstations, priced at a few thousand pounds, this ratio limits the network interface to, say, a 10 Mbps Ethernet. This is quite appropriate to the requirements of workstations, but mainframes require more specialized network interfaces. The expense of large computers means that the high cost of networks such as HYPERchannel and LCN, section 2.2.1, can be easily justified.

Higher Network Speeds

With the development of powerful engineer's workstations, higher data rates will be demanded of the local area networks. Additionally, the introduction of teleconferencing and the possibilities of transferring voice and video traffic will further increase the required network bandwidth. These needs are satisifed by new high speed networks, such as the **Fibre Distributed Data Interface** (FDDI) specified by the ANSI X3T9.5 local area networks sub-committee. The FDDI is a 100 Mbps token passing ring that uses fibre optic cables. Already there are a

number of companies that are developing VLSI chip sets to interface to the FDDI; AMD is one such manufacturer [50,51].

Within two years it should be possible to produce an interface to the FDDI for about the same price as an SSI/MSI Ethernet interface - just a few hundred pounds. Already, there are LEDs, costing less than $10, that can drive an optical fibre at 100 Mhz, but fibre optic cables are still expensive. However, reductions in the cost of fibre optic cables are continuing because of increases in demand. This is attributable to some of the obvious advantages of using optical fibres :

Bandwidth : data rates of several hundred megabits per second are achievable.

Attenuation : little signal loss is experienced, so repeaters are not required to the same extent.

Noise : they are immune to electromagnetic interference.

Security : fibre optic cables are difficult to tap.

As the distinction between the rôles of front-end and back-end networks fades, the advantages become apparent of using a single common network to interconnect all of the devices in the system. The primary merit is that the sharing of data between the different machines is more straightforward, and this in turn permits closer participation by all types of computer in the operation of the system.

3.1.1 A General Network Structure

The network structure illustrated in figure 3–2 is considered to be representative of configurations that may soon be in widespread existence. The network could

have any of a number of topologies and use many different types of media, with the implementation adopted for a specific system being selected according to its requirements.

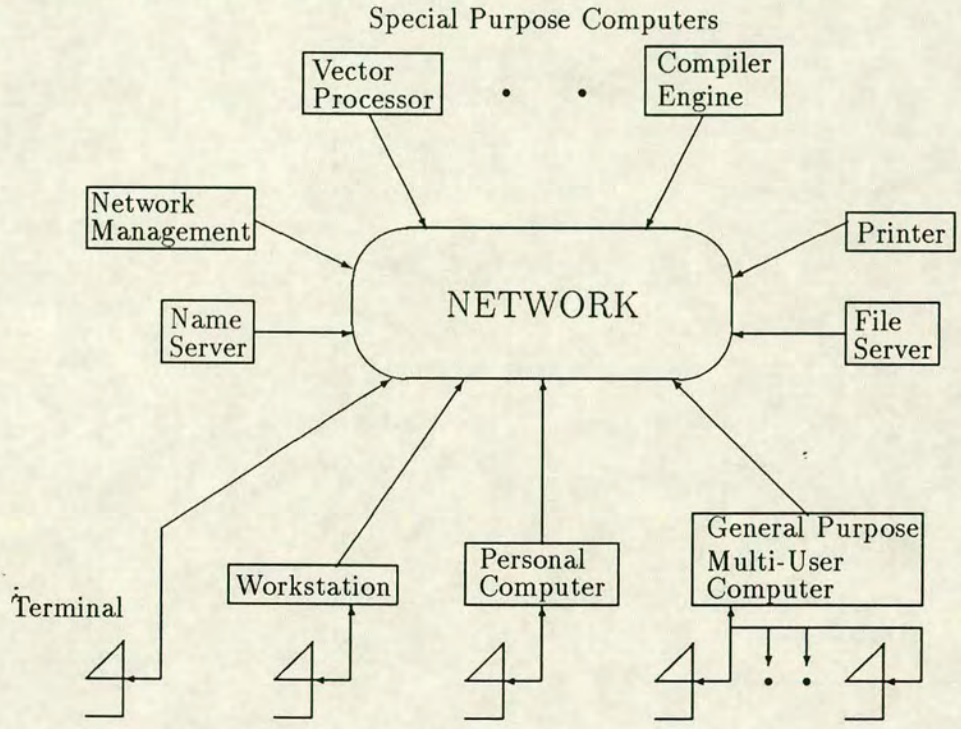

Figure 3–2: A General Network Structure

Common Network

The underlying feature of the structure is that all nodes are interconnected by a common network. However, it is possible, if not desirable, for the implementation of the sub-net to be tailored to the needs and usage of specific devices. Considering the FDDI, introduced in the previous section, the ANSI standard defines two classes of network station, in accordance with the FDDI being formed from two fibre optic rings.

Class A stations are connected to both the primary and secondary rings, offering a greater degree of reliability. Also, the network bandwidth available to these stations is larger because the two rings function independently and counter-rotationally. Hence, class A stations are appropriate for high performance processors, such as the vector processor and compiler engine in figure 3–2.

Class B stations are only connected to the primary ring, which affords them network access with a lower cost and complexity than that of the class A stations. The class B stations are sufficient for the needs of the workstations and smaller computers.

Taking an alternative example, Ethernet could be used by those machines that do not need high data rates but require an inexpensive connection to the network. Centrenet, section 1.2.3, could provide a high performance network for the special purpose computers, and communicate with the Ethernet via a gateway. Furthermore, the topology of Centrenet is such that different technologies may be used for the Starpoints, in response to the requirements of the wide range of devices.

The potential for using different sub-nets to satisfy the various cost and performance requirements is not illustrated in figure 3–2. It is considered that the presence of a unifying network is more significant in this context. Indeed, it would be inappropriate to make apparent any variation in the nature of the network because it is expected to provide a uniform communications substrate.

Network Devices

The wide variety of devices that may be connected to the network ranges from the humble 'dumb' terminal up to powerful supercomputers. With such a broad array of machines, their only common feature is that they are all attached to the same network, and have the potential for communication with each other. Although, as will be explained later, the desirability of permitting this degree of freedom is questionable.

The terminals will have no inherent computational capabilities, and for most networks this is adequate for network communication. Hence, it is most likely that the terminals will operate in small groups, sharing access to a semi-intelligent network adaptor. This adaptor will appear to the terminals as a conventional terminal multiplexer or concentrator. However, rather than communicating with a single host computer via a high speed I/O channel, the adaptor will *assemble* terminal character sequences into packets suitable for transmission over the network. On receipt of network packets, the adaptor will *disassemble* them and route the information to the appropriate terminal lines.

The **Packet Assemblers/Disassemblers** (PADs) used for wide area network communication over the **Joint Academic Network** (JANET) are typical of network adaptors for terminals. For local area networks, the Centrenet Terminal Multiplexer [82], section 1.2.3, represents the features that may be provided.

In addition to the unintelligent terminals, the users will also have direct access to a number of other machines. These machines include personal computers and workstations that have reasonable levels of processing ability together with the facilities for providing a good user interface. Additionally, there are medium per-

formance multi-user computers, to which terminals may be connected directly or via the network. The disparity between personal workstations and these multi-user computers has diminished to the extent that they may comfortably communicate with each other using standard networks, such as Ethernet.

Of the machines connected to the network that the user cannot directly access, some are high performance special purpose processors that have been designed for particular types of computation, such as vector processing or compiling. The architectures and software structure of these machines are optimized for their respective applications. Hence, they are not suitable for direct interaction with the users of the system.

The remaining network devices provide services for use by all network nodes in support of the general operation of the system. There are devices that are dedicated to the provision of specific services and others that will also have other rôles. Examples of the former include file storage and nodes providing access to peripherals, such as printers and graph plotters. Additionally, there are machines that are purely concerned with the management and control of the system, performing functions such as scheduling and resource allocation.

Network nodes that provide a variety of services are best illustrated by networks of disc-less workstations using a shared filserver. The Edinburgh University Computer Science Department's network of **Advanced Personal Machines** (APMs) [2,56] is a typical case. Here, the filestores not only provide storage for files and directories, but they are also responsible for system authentication and peripheral spooling.

Finally, there are gateways from this network to other networks. The network

devices that provide this service are primarily concerned with protocol conversion. More sophisticated nodes will perform *address filtering* such that only those packets that are intended for nodes on the other network are converted and retransmitted. Extra facilities, such as address translation, may also be supported.

Summary

To summarise, the network structure outlined here is composed of a wide range of devices that share a common network. At this stage, there are no patterns or operation of conventions to govern interactions between network nodes. So, the structure does not have a rigid formation and benefits from a significant degree of flexibility in its behaviour.

However, the objective of bringing together this wide range of machines, by connecting them all to a single network, is to encourage close cooperation between the machines. Before such as system may be realised, it is necessary to define a set of conventions to govern the way in which the network nodes interact with each other. A well structured set of conventions will enable the individual devices to work together as a unified system; without any government at all the devices operate disjointly, contesting for use of the network.

3.1.2 Classification of Devices

With such a broad variety of machine types, it may be considered appropriate to develop a small and simplified communications protocol that is generally applicable to all network nodes. This is because the level of design effort involved in

producing a protocol to cater for the needs of all machine types would be inordinately complex. Furthermore, the quantity of code to be developed would be very large and cumbersome.

The strategy of defining a simple common protocol has the advantage of requiring only a small design time, with subsequent development for all network nodes of a compact kernel that supports the full protocol. To accomodate the diverse range of devices, higher level software would need extra code to compensate for the simplicity of the lower level protocol.

An alternative approach is to combine the performance benefits of a protocol tailored to the specific needs of devices with the low cost and high efficiency of a simplified protocol. This is achieved by dividing the wide variety of network devices into a limited number of classes. Then, it is possible to isolate the requirements of communication within and between these different classes. The ultimate objective is to provide a set of protocols, each of which is oriented towards efficiently satisfying specific needs.

In [60], the authors state their belief in the use of application oriented protocols in order to achieve high levels of performance. In the Apollo DOMAIN, a number of protocols exist, each of which is intended for use with a specific sort of service. There is no need in the DOMAIN for account to be taken of differing machine types because only personal workstations of similar power are used on the network[1].

---

[1]The CRAY Station Software Service is the exception here, but is inappropriate to this argument, as explained in section 3.1.

In considering the general network structure of figure 3–2, three distinct classes of device may be distinguished, as illustrated in figure 3–3. These classes identify a level of polarization that is occurring in the development of devices for use on a network. In future network environments, these classes may be expected to correspond to the full range of devices on the network. Those devices that do not fall neatly into these divisions are of diminishing importance in networked computing environments.

Three Classes

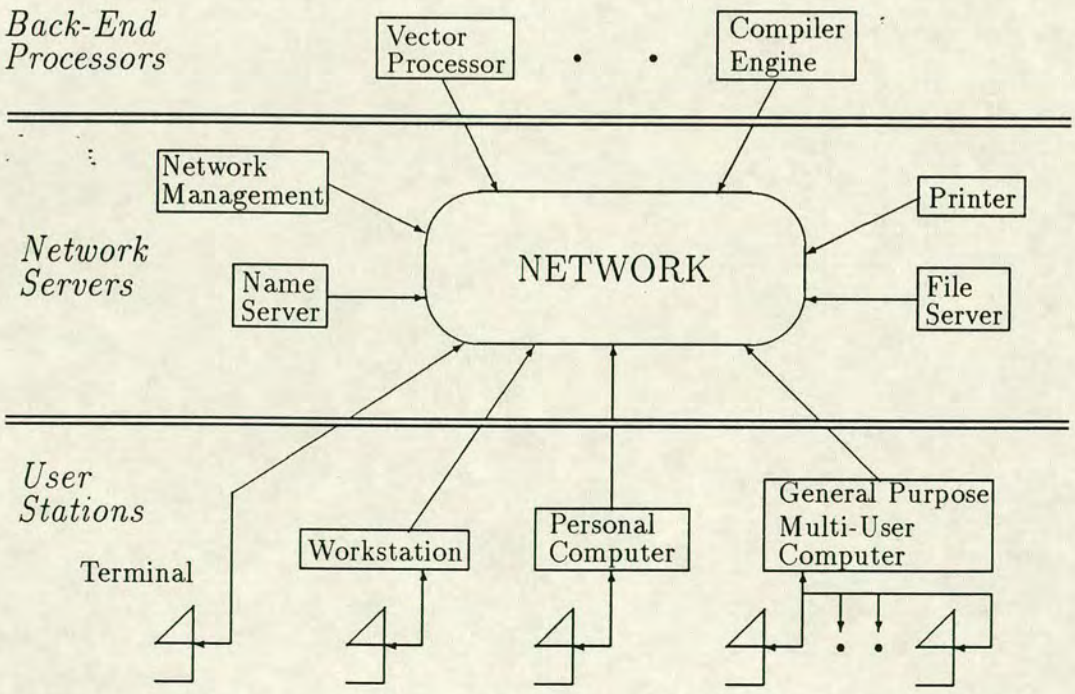

Figure 3–3: Device Classification within a General Network Structure

The first of these classes is described in figures 3–3 as *User Stations.* Included in this division are personal computers, such as the IBM Personal System/2, and scientific/engineering workstations like those of Apollo and SUN. These machines

fall naturally into this partition, but simple terminals and multi-user computers are less well suited.

The case for maintaining specific allowance for the presence of 'dumb' terminals is minimal because it is now more cost effective to purchase, say, an IBM PC with the necessary interface cards and software. This fact is reflected in the strategies of many companies. Furthermore, as explained in the previous section, terminals are not sufficiently powerful for direct network connection, but require a more intelligent network adaptor.

It was stated in section 3.1.1 that the difference, in terms of computational speed, between multi-user minicomputers and personal workstations is rapidly disappearing. Consequently, an increasing number of installations are opting for a computing environment derived from a multiplicity of personal systems rather than a single shared central machine. Although this trend will inevitably proceed further, the case for retaining the shared system remains, as explained in section 2.2.3. However, communication between minicomputers and personal systems is achievable using standard networks and protocols. So, it is reasonable to group these multi-user machines together with the personal systems.

The second class encompasses all those devices that are dedicated to the support and control of the operation of the network. These *Network Servers* include most peripherals and file storage devices as well as the nodes that are reponsible for the network management. The machines in this division are responsible for activities that would normally be associated with the operating system of a conventional machine. Consequently, another distinguishing feature of these devices is that no user programs execute within them.

To give an indication of the type of device that could be found in this category, consider the **Centrenet Network Intelligence Module** (NIM), section 1.2.3. The NIM has duties concerned with the physical operation of the network, such as polling sequences and clock signals. Additionally, the NIM may perform name-serving and virtual circuit management as well as collecting statistics about the behaviour of the network and performing error recovery. Hence, it is obvious that the behaviour of the NIM is inexorably associated with the operation of the network.

The machines that comprise the final division are the special purpose *Back-End Processors*. Computers that satisfy this description include fast numerical processors, such as **MU6-V**, section 1.2.2, as well as specialist mathematical processors, compiler engines [12], simulators, image processors, language translators and inference processors.

As mentioned earlier, a primary characteristic of these machines is that, by virtue of their highly optimized nature, they are unsuitable for direct access by the users. In this respect, MU6-V is a fine example of such a machine because its architecture incorporates mechanisms for achieving a high performance in vector calculations, but the machine possesses virtually no operating system software.

Also included in this category would be commercial supercomputers, such as the Cray Research, CDC and Eta Systems machines. All of these computers incorporate processing capability for supporting a substantial operating system and hence it is possible (though unusual) for direct terminal connections. However, it is regarded that the performance of these machines is impaired because of this support. So, cheaper, more specialized systems, such as the latest Intel range

of Hypercube-based machines, have architectures and software that favour high performance in particular types of calculation in preference to supporting a general purpose operating system.

Review of Classification

The three classes of device that have been highlighted in the general network structure provide a focus for further considerations of the way in which these devices will require use of the network. However, this classification is inadequate for the stated intention of "isolating the requirements of communication within and between these different classes". This is because the simple three-way partitioning requires any device attached to the network to fall into one, and only one, of these categories. Merely the degree of variation possible in the nature of personal systems serves to illustrate the imprecise nature of this simple division.

The objective of classifying the network devices is to provide a framework in which to develop a set of protocols. Hence, to strengthen this framework, the model defined in the next section takes the three classes described earlier and isolates those features that are of primary concern. The resulting categorization is sufficiently rigorous for the subsequent definition of interaction requirements. However, although this classification is more precise, it will be seen that the model does not restrict the involvement of devices that do not conform to any individual class of module.

3.2 A Triadic Network Model

The **Triadic Network Model** (TNM) is an idealised representation of a networked computing environment such as that of the general network structure described in section 3.1. Within the model three types of module are defined that represent the three classes of device described in the previous section. The correspondence between the devices and the modules is

User Station	⟶	**Personal Workstation Module** (PWM)
Network Server	⟶	**Network Service Module** (NSM)
Back-End Processor	⟶	**Back-End Module** (BEM)

The model defines the three types of module to be *logical entities*, and so it is possible for an actual network device to be composed of a number of these modules. In this way, it is possible to accomodate the significant differences in the form of the machines that approximate to these modules, without having to unduly relax the definitions of the modules.

The description of the characteristics of the three types of module is an essential part of the model, but primarily the model exists to define the nature of the interactions between these modules. This specification of inter-module communication, together with guidelines on the intended operation of the system, provides sufficient foundations for the development of protocols oriented towards this environment.

3.2.1 Characteristics of the Modules

Before considering the interactions between the modules, the nature of the modules should be explained.

Personal Workstation Modules (PWMs)

The **Personal Workstation Modules** are notionally general purpose single-user workstations. Each PWM is considered to have good computational capabilities and a fairly substantial amount of memory. They possess powerful input and output facilities, possibly including colour graphics and sound for output, and using a keyboard, mouse, digitiser or speech for input.

The PWMs are very much user oriented and contain a number of resources, both hardware and software, that are specifically intended to support a good interface between the user and the system, i.e. a good *Human-Computer Interface.* These resources allow the development and use of *Intelligent Front-Ends* (IFEs) within the PWMs themselves.

> "An intelligent front-end is a user-friendly interface to a software package, which uses artificial intelligence techniques to enable the user to interact with the computer using his or her own terminology rather than that demanded by the package." [16]

In [16], Alan Bundy discusses the basic techniques required in an intelligent front-end but concentrates on support for textual input/output. However, he indicates that more sophisticated hardware features, to enhance the interchange between the user and the computer, can provide a more versatile IFE.

As software packages become increasingly sophisticated, the need for more intelligent front-ends grows. Additionally, to avoid the need for familiarity with a large variety of user interfaces, it is desirable for as much commonality as possible between the presentations of different software packages. The importance of this is reflected in the efforts of the Andrew research team in the development of the **base editor tool kit**, section 2.5.3.

Another move in this direction is visible in the Microsoft development of OS/2. Here, the presentation functions of the operating system form a separate package from the remainder of the kernel. This package, **Windows**, is available separately from the rest of OS/2 to encourage its use on other systems.

Together, a sophisticated presentation package and a general purpose IFE would greatly improve the flow of information between the user and the computer. The PWMs are ideally suited to this rôle; not only do they possess the physical devices needed by a presentation package such as Windows, but they have sufficient local processing power for a useful IFE.

The PWM provides a good starting point for the development of "user access stations" that are, in effect, behaving as highly intelligent terminals. However, it is unlikely that such a radical change in the operation of personal systems is imminent, so the Triadic Network Model does not place such a requirement on PWMs. Yet, a PWM is expected to place the user's needs at a very high priority.

Hence, operation as an IFE and presentation device is not the sole purpose of PWMs, but in fact they have sufficient processing power for many of the user's computational requirements. With the presence of local disc storage, this allows

them to be used as 'stand-alone' computers, operating completely independently of the rest of the system.

This last point raises the question as to whether or not PWMs should have local disc storage, within the guidelines of the model. Workstations with and without local discs are likely to be present on the network, and both must be supported. As far as the remainder of the system is concerned, the distinction between the two types of workstation only arises if direct access to the file storage is to be granted to the system. In this event, an NSM must also be supported by the workstation, figure 3–4. Otherwise, the local discs are considered to be private to the PWM.

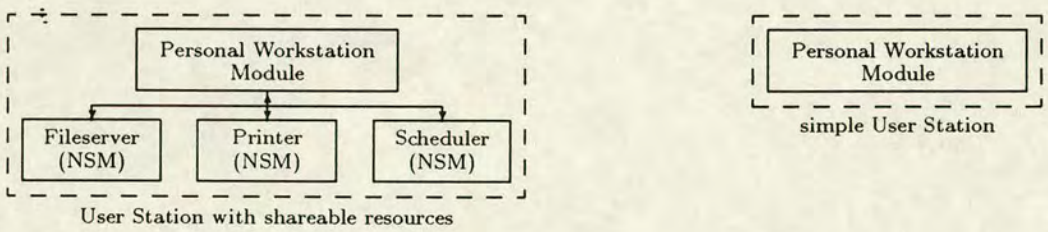

Figure 3–4: Examples of User Stations

The connection of the PWMs to the system provides the advantages to the user of allowing the use of shared network resources such as printers and file storage devices, and access to specialised services supplied by high performance processors. However, an important use of the network by the PWMs is for interaction with each other, both individually and in groups. This interaction may take the form of textual messages (passed by a *mailbox* utility or directly), sound or vision.

Back End Modules (BEMs)

The **Back-End Modules** are special purpose computers. They are designed to achieve a high performance with great efficiency in particular applications. Each BEM is intended for use in a specific type of application. Consequently the BEMs bear little or no resemblance to one other and will probably have quite strict requirements with regards to the form of their input and output. These requirements may be quite complex and the users are unlikely to be able to comply with them without significant effort. Similarly, because of their dissimilarity, the BEMs may find great difficulty in direct communication with each other but inter-BEM communication is not likely to be a frequent occurrence.

The BEMs may be considered to be individual resources, where each BEM corresponds to a single resource. The resource may only be suitable for utilization by a single process at any one time, necessitating some quite sophisticated resource management. It could be expected that the BEMs should perform these management functions themselves, but this would be at the expense of their efficiency. Hence, the model requires every BEM to have an associated manager, provided by an NSM. The manager is responsible for controlling access to its BEM.

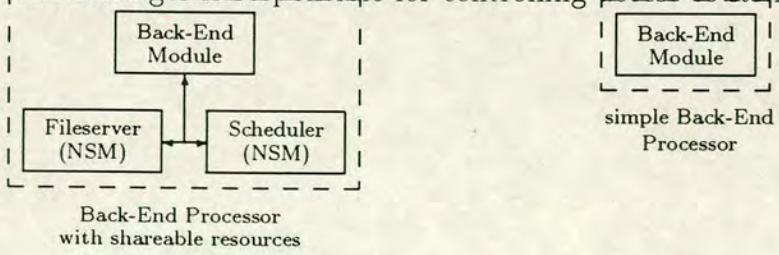

Figure 3–5: Examples of Back-End Processors

The comments about local disc storage on workstations may also be applied

to back-end processors. So, a BEM may incorporate local disc storage for private usage, but if general access is to be given to the system then an NSM must be provided, figure 3–5. Further discussion of this continues in section 3.2.3.

In the case of BEMs, a good deal of the code and data used in processing will be shared with at least one other network device. Hence, it may be more efficient for BEMs to make extensive use of shared network storage rather than private local discs. This suggest two options for a back-end processor with local discs :-

1. Provide an NSM so that general network access is possible.

2. Use the discs for standard libraries of local code, and also fór virtual memory paging support.

Network Service Modules (NSMs)

The NSMs are dedicated network servers whose purpose is to assist in the general operation of the distributed system. The services which they might provide include file storage, data bases, peripheral access, virtual circuit establishment, background resource management (e.g. compilation) and special purpose resource management. An individual network server may be dedicated to the support of a single service, in which case only one NSM is provided, or it may serve a number of different needs, as illustrated in figure 3–6.

Collectively the NSMs provide a **Network Operation Support Service** (NOSS) required for the implementation of a distributed system. Figure 3–7 shows how the **Network Service Modules** may be viewed as providing an *'interface'* between the other two types of module.

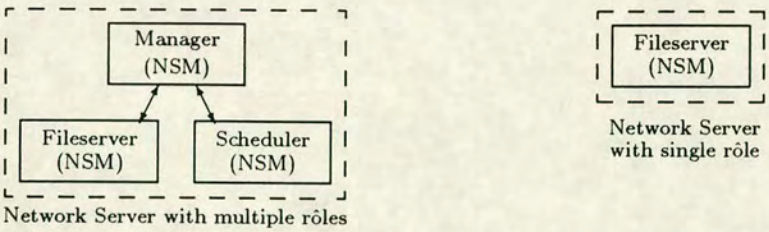

Figure 3–6: Examples of Network Servers

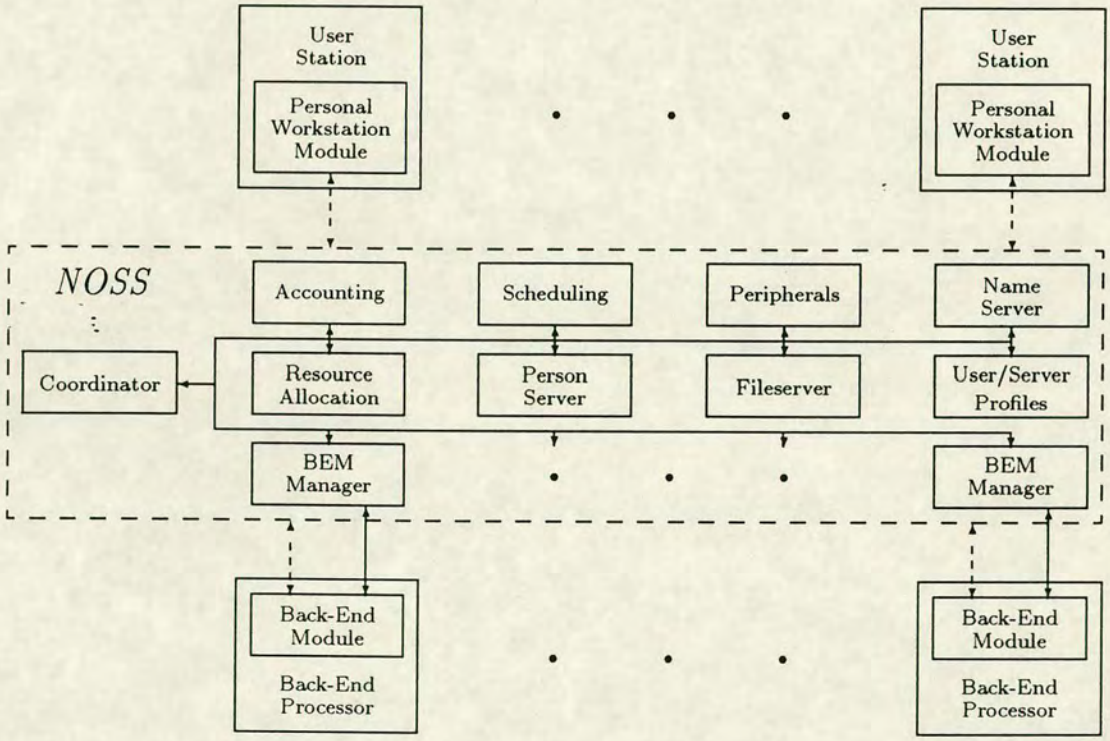

Figure 3–7: A Networked Computing Environment

The NSMs supply resources for use by both the PWMs and the BEMs. The presence of the NSM domain is desirable from the viewpoint of both the PWM and BEM domains since it permits economical sharing of frequently used resources, such as large disc space, printers etc., and it enables the BEMs to operate at their optimum efficiency, by removing all operating system concerns.

The NSMs allow even more advanced types of service to be provided than the individual BEMs are capable of supplying. An advanced network service may be created by utilising a number of BEMs and mapping all requests for this new service onto a set of interactions with the group of BEMs. For example, an individual image processor BEM may not be capable of operating at sufficient speed to cope with real time data. So, by using a number of image processor BEMs, a NSM may take 'live' video pictures and share the job of image enhancement between the group of BEMs. The NSM can then combine the outputs and direct them to the required destination.

Notice that an alternative approach would be for an NSM to multiplex a number of service requests from PWMs onto a single BEM. The resulting service would be of a lower quality than the BEM could provide for a single process, but the level of service desired by the PWMs may actually be quite low, in which case the BEM is being used more effectively by the system through the use of this simple mapping technique.

The organizational part of a distributed operating system comprises the functions that it is inappropriate to locate on either PWMs or BEMs, and therefore these functions are located on the NSMs. Consequently, the NSMs are involved in accounting, scheduling, protection (e.g. store management), synchronization and communication, and the allocation and management of resources. Most of these functions are provided by those NSMs that are dedicated to the management of the system, but some accounting may be performed by the individual resources and many of the NSMs will be concerned with protection. Additionally, the pres-

ence of communication functions is obviously a prerequisite of all NSMs because of their interconnection by the network.

The key position occupied by the NSM domain means that there are three important requirements of the NSMs:-

reliability : since these devices are critical to the operation of the full system, the domain should not be susceptible to total failure through the failure of a single node.

security : the NSMs control access to expensive resources and specialised services, as well as managing communication between PWMs, so they must be capable of protecting the other devices from illicit intrusion by unauthorised users.

integrity : in the case of the use of a BEM service, the NSMs manage the entire transaction so they are responsible for ensuring successful total completion of each request.

Support for other Devices

The modules defined by the Triadic Network Model (TNM) correspond to the sort of devices that are anticipated to be present in networked computing environments of the near future. However, it was suggested in the discussion of the general network structure, section 3.1.1, that other types of device must also be accomodated. So, consideration of the provision for such devices within the model will be made here.

The prospects for the humble terminal have already been expounded upon in earlier sections. The connection of 'dumb' terminals to networks using terminal multiplexers is nevertheless likely to continue for some time. However, although the terminal is more akin to PWMs than to either BEMs or NSMs, it does not have the computational power to be capable of supporting the required standard of user interface.

For a terminal user to partake in the networked computing environment based on the model, figure 3–7, the terminal must be *assisted* by a more powerful network device. So, the terminal user may initiate a connection to, say, a general purpose multi-user computer and thereby to the remainder of the system. For the duration of the session there is a process on the multi-user computer that is associated with the terminal. Effectively, the remote process is emulating a PWM, but for a *virtual* machine rather than an individual personal system.

The approach of providing PWMs on a general purpose machine, so that terminal users can gain access to the system, should be viewed as a temporary measure. Whilst it may appear to provide a more cost effective approach, with the diminishing margin between simple personal computers and conventional terminals, this is not really the case. Furthermore, since the PWMs provided by the multi-user machine must inevitably be of low power, this technique is likely to be much more demanding of system resources, without giving a particularly high standard of service to the user.

Supporting simple terminals is only one potential rôle for multi-processing minicomputers. An alternative would be for the machine to provide a number of NSMs, dedicated to identical or dissimilar functions. This would result in a

similar configuration to that of a collection of Unix 4.2 BSD machines, connected by a common network. Under this system, a number of *daemon* processes exist on each machine. The processes are dedicated to specific services, but they are normally dormant and only demand machine cycles when they are requested to provide the service.

For services that are very simple or are infrequently used, this may be a more appropriate solution than the provision of a multitude of elementary network processing elements. However, more demanding services or popular functions will need higher performance than this approach can provide.

It should be clear from this discussion that it is not necessary for an exact correspondence to exist between a logical TNM module and a physical network device. More important is that the processes communicating over the network as part of the system should comply with the requirements of the model.

3.2.2 Interactions of the Modules

The Triadic Network Model (TNM) describes the nature of interactions between the three types of module defined in the previous section. The specification of communication strategies derived from the model is used as the basis for a set of protocols. The freedom of communication exhibited by the presence of a single unifying network, section 3.1.1, is welcomed and exploited, but some restrictions are needed to control communication between the nodes, and better utilize the resources. The fundamental restrictions imposed by the model are :-

1. **Back-End Modules** never interact with each other.

2. **Back-End Modules** do not communicate directly with **Personal Workstation Modules.**

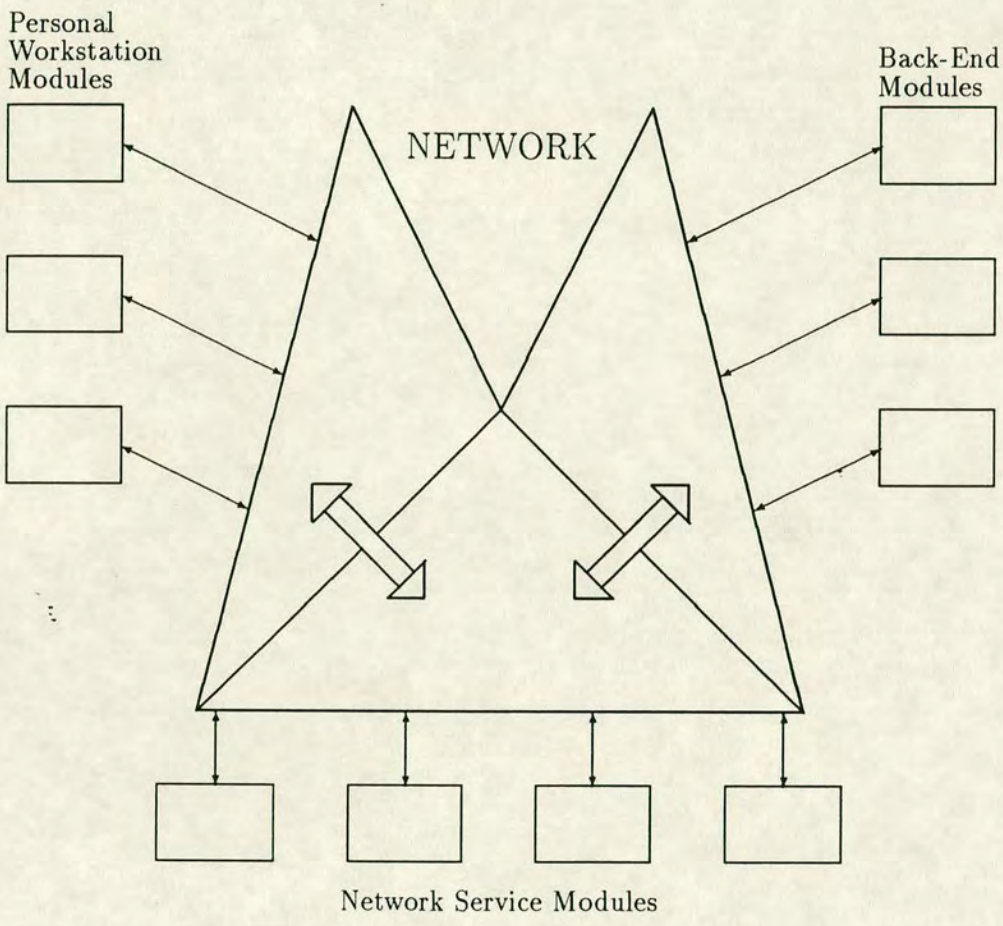

Figure 3–8: Triadic Network Model

Figure 3–8 represents the model of communication : the domain of each type of module is shown as a triangle and where the triangles touch it indicates that the domains may interact. This diagram may also be used to illustrate the supportive rôle played by the NSMs, and it shows how the NSMs provide an interface between the other two domains. Table 3–1 indicates the extent of the model's restrictions - an 'X' signifies that valid communication is possible.

		TO		
		PWM	NSM	BEM
FROM	PWM	X	X	
	NSM	X	X	X
	BEM		X	

Table 3–1: Possible communicating pairs

The justification for the inter-BEM and PWM-BEM communication restrictions is explained in section 3.2.3, together with an indication of why these limitations are beneficial rather than inconvenient.

Personal Workstation Module Intercommunication

A key feature of Personal Workstation Modules is the presence of facilities for human interaction. In early discussions, these facilities were viewed as important in the context of the users' interaction with the system. However, another rôle is in allowing communication with other users of the system.

Where a computer configuration has a collection of terminals, there is usually provided a facility for allowing text messages to be transmitted between terminals. PWMs, though, may have audio and visual capabilities that allow more natural communication to occur between users. A possible consequence of this is that inter-user communication may form an even greater proportion of network traffic than it does at present. Hence, in this section, communication between PWMs is considered.

The communication between the PWMs may have the following forms:-

- interactive transfer of small amounts of text (c.f. VAX/VMS 'phone' utility)
- digitized speech [37]
- digitized pictures, whether 'live' video or computer generated graphics [38].

For efficiency, it is desirable that there is minimal involvement of NSMs in this intercommunication. However, a more fundamental network requirement, that the topology of the network and the physical locations of the network devices remain invisible to individual nodes, seems to conflict with this aim. This is because only the NOSS is considered to be aware of the actual state of the network. This is partially for reasons of security, so that any individual node is only aware of those devices to which it is allowed access. More important, though, is the flexibility that this limited awareness brings to the system in its provision of services.

Before any two devices on the network may interact, they must first determine their respective physical addresses by communication with the NOSS. Hence, the rôle of the NSMs in this type of interaction should be restricted to that of *'name server'*. So, the PWM wishing to make contact with another PWM initially contacts the NOSS **Name Server** module to determine the physical address of the other node; the subsequent establishment, utilization and termination of the actual connection are controlled by the PWMs. An alternative situation, where multiple PWM connections are required (for teleconferencing, for example) would justify the use of an NSM to assist the 'chairman'. Similarly, PWM intercommunication between users speaking in different languages may be assisted by the use of a translator BEM. In this case the involvement of an NSM is essential.

Hence, the initial enquiry sent from the PWM to the NOSS should state the type of intercommunication to be used. For basic communication the NOSS Name Server module would just return the address of the destination PWM. If a more advanced type of communication is required the NOSS returns an acknowledgement to the source PWM and then endeavours to provide the desired service.

Use of Standard Network Services

The standard network services are those functions that the NSMs provide for use by both the PWMs and the BEMs. Included in this category are the network filestore and peripheral access. Access to any of these services is initiated by the sending of a request to the NOSS **Coordinator** module of figure 3–7. This request is validated and then, possibly after having been queued temporarily, it is forwarded to a suitable NSM. Any subsequent interactions between the customer and the supplier pass directly without any involvement of the NOSS Coordinator module. On completion of the service request, the NSM that has supplied the service indicates its availability to the Coordinator and it may then be used by another customer.

For most of the transaction, the nature of the interactions is largely as described by the **Client-Server** model. The differences lie in the initiation and the termination procedures. Here, the protocols are of a three-party form, with the NOSS Coordinator module acting as mediator in negotiations between the requesting PWM and the supplier NSM.

Use of Special Network Services

"Special" network services are those functions that necessitate the involvement of Back-End Modules. This may be because the NSMs are unable to provide the service, or it could be that the NSMs can provide the service but not at the standard of performance required by the requester. Also, of course, the fact that NSMs are dedicated to specific tasks means that a user's own program must be processed by a BEM.

As far as the PWMs are concerned, the mechanism for accessing specialized network services is the same as that for use of the standard services. The PWM sends the service request to the NOSS Coordinator module, as before. The Coordinator treats the service request largely as for a standard service and, after validation and scheduling, the request is passed to another NOSS module. This module acts as the BEM's **Manager** and any subsequent interactions with the customer PWM, needed for the effecting of the function, are made by the BEM's Manager rather than by the BEM itself. So the PWM is never made aware of which BEM is acting as the supplier.

Since the BEM's Manager is responsible for all high level interactions with the PWM, the only network protocols that need to be implemented on the BEM are of quite a basic nature. Additionally the BEM need only be concerned with the sort of computation for which it was designed; there is no requirement for a sophisticated user interface or for advanced operating system functions. Hence the BEM may operate with a high degree of efficiency.

3.2.3 Functional Aspects

In the preceding sections, the nature of the communication between TNM modules has been considered and the characteristics of the modules themselves were defined. However, certain restrictions were imposed on both the communication and formation of the modules. In this section, these limitations are shown to be appropriate to the target environment, and techniques are introduced to reduce the inconvenience of these boundaries.

Relating Devices to Modules

The distinction between a module, as defined by the model, and a device to which it is a close approximation has already been emphasised. Figure 3–9 shows the examples, given earlier, of network devices that present a much larger range of resources to the system than those of a single module. Hence, it is evident that, for example, a back-end processor is more than just a Back-End Module.

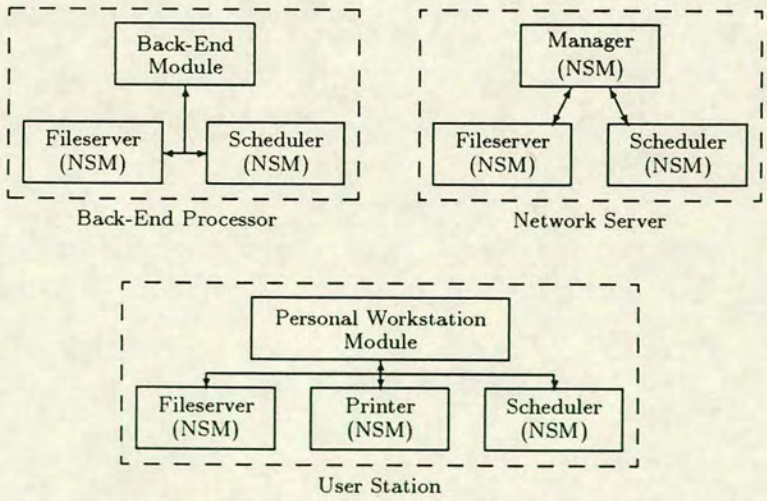

Figure 3–9: Examples of Network Devices

Re-examining the networked computing environment of figure 3–7, it should be observed that both the Personal Workstation Module and the Back-End Module only form a part of the network device that supports them. This is true even in those circumstances where the device provides no extra resources to the system. Indeed, the aspects of a device that are not encapsulated in its corresponding module may be said to be "outside the bounds of the model".

To clarify this distinction, it may be helpful to consider some practical examples of network devices and identify corresponding network modules.

APM : The Edinburgh University Computer Science Department's **Advanced Personal Machine** (APM) is an obvious example of a user workstation. Most APMs will roughly correspond to the model's view of a Personal Workstation Module, but strictly the basic APM does not provide particularly good facilities for the user interface, as expected by the model. Also, there are APMs with a local disc unit that may be considered as an NSM, depending on its use within the system.

APM Filestore : The APM filestores are network servers, but they do not form single NSMs because of their multiple rôles. For example, the APM filestores are also responsible for maintaining a list of all active users, validation of access permissions and permit access to peripherals such as the machine halls printer. Each of these rôles may be implemented as a separate Network Service Module.

MU6V : This is a back-end processor and the Back-End Module is supported by the I/O-processor.

Meiko Computing Surface : The processor corresponds closely to the model's definition of a Back-End Module, although the actual module would be implemented by the MicroVAX that 'front-ends' the machine because the Meiko machine itself does not have any form of network interface adaptor.

CDC Cyber 205 : Again, the processor meets the model's description of a BEM, but the module itself may be resident on the HyperCHANNEL or LCN interface. It may be worth noting that the machine's front-end (Amdahl 470/V7A, in the case of UMRCC) does not provide the BEM service, but in fact acts as an NSM.

As is apparent from the examples given above, it is not always appropriate for the TNM module to reside with the majority of the software for its machine. So, for the Meiko Computing Surface, whilst the machine satisfies the model's definition of a Back-End Module, the system requirements of the TNM BEM software are such that it is more effective for this software to be implemented by the MicroVAX front-end. The system does not view this front-end as being the BEM, but merely as the interface to the BEM; it is not possible for the other nodes of the system to interact directly with the Meiko machine, and hence the two devices are indistinguishable.

Communication Restrictions

The Triadic Network Model prohibits direct communication between Back-End Modules. For many Back-End Processors this will be inevitable because of significant differences in their representation and use of data. In some cases these differences are less distinct and it may be desirable for the Back-End Processors to

transfer data directly to improve their efficiency. To allow high speed data transfer between Back-End Processors, an intermediary NSM may be used, as illustrated in figure 3–10.

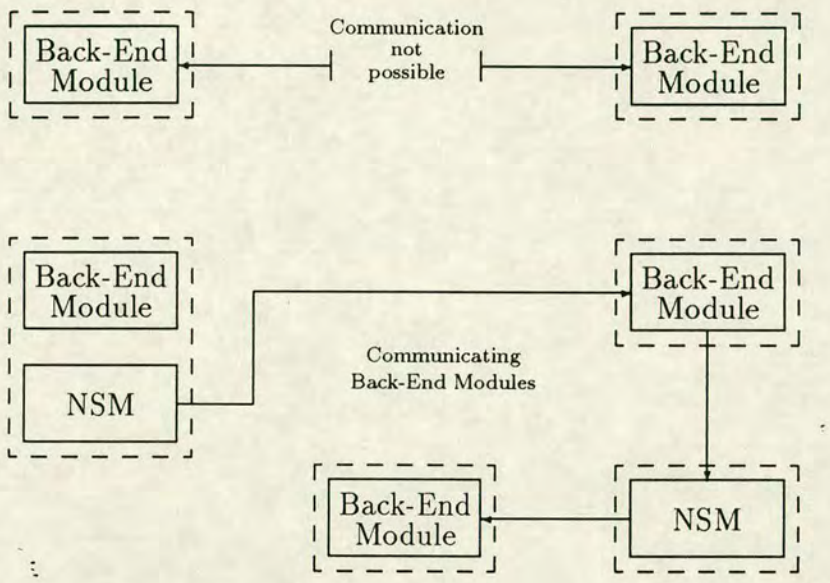

Figure 3–10: Overcoming the Inter-BEM Communication Restrictions

The benefit of not requiring all BEMs to provide the facility for direct data transfer is that incompatible Back-End Processors are not unnecessarily burdened by having to implement extra (intricate) protocols. Those Back-End Processors that are more compatible may provide an NSM for the purpose of establishing and controlling the connection. Alternatively, a remote NSM may be used to translate the data passing between the machines from one representation to the other, as in the second stage of the example in figure 3–10.

The rôle of the intermediary NSM varies according to the needs of the connection it is supporting. So, where the supply rate is no greater than the consuming rate (*direct* use of data), the NSM is only of use in the establishment and close

down of the circuit. However, where the supply rate is greater than the consuming rate (*delayed* use of data), the NSM may provide a form of buffering facility. The buffer may reside at the back-end processor or it may be located remotely at a network filestore.

The main reason for prohibiting inter-BEM communication is because of the differences in data representation, but additionally it should be stressed that direct communication between BEMs conflicts with the normal operation of the system. This is because each BEM is considered to be a valuable resource, for shared access by the users of the system. Hence, the scheduling of requests, together with resource allocataion and accounting, are of great importance in all communication involving BEMs. To allow BEMs to communicate with each other, without the involvement of NSMs, would introduce one of the following two uninviting options :

1. The BEMs would become responsible for management of accounting records and scheduling of requests.

2. It would be possible to circumvent the normal system constraints by direct communication between BEMs.

These considerations also apply to communication between PWMs and BEMs. However, there are also much more important factors. Foremost amongst these is the desirability of isolating the functions supported by the BEMs from the corresponding devices. The immediate consequence of this is that system nodes have to request use of system services by *type* rather than *location*. This gives flexibility to the system in the choice of how to implement the service, and of

which devices to use. To illustrate this, consider the possible approaches towards satisfying a request to compile a Pascal program :

- A general purpose processing node may be used. It would have to load the code for the compiler, as well as the source code for the program, and store the object code on completion.

- A special purpose compiler engine could probably provide an improved performance because of its optimized architecture.

- Distributing the compilation amongst a number of processors, either as a multi-stage pipeline or by partitioning the task into a number of smaller identical tasks for parallel processing, may well provide an even greater performance.

The implementation adopted by the system will depend on the requested *standard of service*, the presence of system processing resources and the availability of those resources. However, whatever approach is taken, the requester of the service should not be required to be aware of the nature of the service provision.

These objectives are relevant to all nodes on the network and are appropriate for all types of service. They are of special importance, though, for PWMs using BEMs because of the extra disparity between the form of these modules. Furthermore, it may be anticipated that the communications bandwidths for PWMs and BEMs are likely to be of completely different orders. Where communication between a user station and a back-end processor is required, the technique of using an intermediary NSM should be adopted.

3.3 Application of the Model

The characteristics of three types of module, representing varieties of network device, have been defined as part of a **Triadic Network Model** (TNM). The nature of the interactions between the modules has also been described. In this last section of the chapter, the utilization of the model is considered. The notion of a **layered network transparency** is used to reduce the complexity of implementing protocols based on the model and finally a study is made of how the model may be applied in practice.

3.3.1 Layered Network Transparency

It is generally agreed that it is desirable to make a heterogeneous multicomputer system appear as if it were a single entity because this greatly simplifies the applications software and provides a more uniform view of the system to the users. It is also considered preferable to make the protocols governing interaction between the different nodes on the system quite basic so that it is straightforward to implement them on a wide range of devices.

The **Triadic Network Model** would appear to violate both of these aims because of the widely differing levels of processing capability of the different network devices, and the distinction between standard and special services. The idea of the various parts of the system having different degrees of awareness of the nature of the network is introduced in an attempt to overcome these difficulties. This use of

a layered interface to the network allows the sophisticated nature of the network structure to be concealed from the user by relatively basic pieces of software.

User's View

From the uppermost layer, the user is aware of having access to a large and comprehensive range of services, which enable him to communicate with fellow users, interrogate a large information base and perform extensive processing of information. Access to these services is provided by the user's personal workstation, and to the user it may appear that this machine is performing functions that are in fact being executed elsewhere on the network. Similarly, the application software is only aware of the functions that are available. The software communicates with all of the processes supplying the functions in the same manner, regardless of whether they are internal or external to the workstation.

Workstation's view

The workstation divides the global set of functions into two categories, local and non-local. It may be possible for some functions to reside in both categories, if different levels of service are recognized and the network is able to offer the same functions but with a higher level of service.

For local functions, communication from the application software is passed directly to the appropriate internal process. For non-local functions, the workstation will first send a service request to the network and then pass the information to the network server. The workstation is only aware of the division between local and

non-local functions; it has no knowledge of the difference in the types of non-local functions.

Network's view

The network distinguishes between standard and specialized services. For standard services, the service request is passed to the corresponding NSM, but for specialized services the request is passed to another NOSS module to manage the transaction.

When the NOSS is regarded as composed of a number of modules, it is evident that the NOSS Coordinator module itself draws no distinction at all. The distinction arises out of the way in which the services are actually implemented: for a standard service, the Coordinator passes the service request directly to the NOSS module that will provide the service, whereas for a specialised service the destination NOSS module uses another machine, a BEM, to implement the service.

Effects and Benefits

The layered structure that results from considering service requests in this way is illustrated in figure 3–11. It is apparent that, at each level, the decisions that need to be taken are relatively easy to make. Furthermore, the nature of the options available at each stage is appropriate to that level. Consequently, the routing of a service request to a suitable provider may be achieved effectively with uncomplicated software procedures.

In studying figure 3–11, it is clear that an extra level has been added between the *workstation's view* and the *network's view* described earlier. This particular

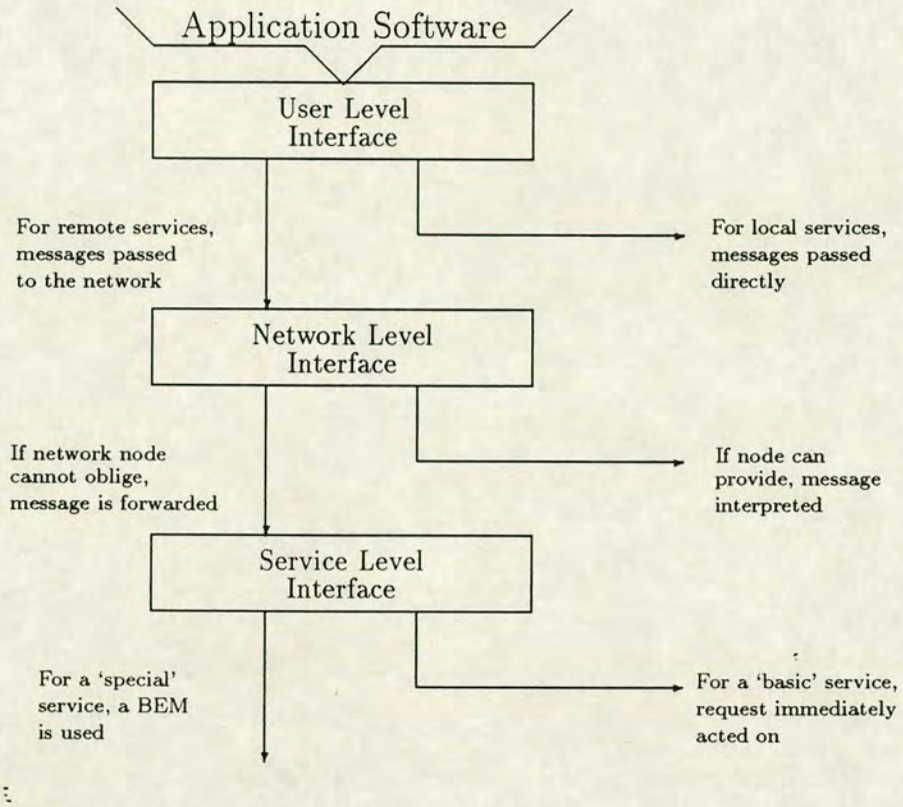

Figure 3–11: Layered Structure for Service Request Analysis

stage is present to allow a node that receives a service request to forward it to another network server if it is aware that this second node is capable of providing the service, whilst the original server cannot. This facility is of greatest use to the NOSS Coordinator module, since, logically, this node is responsible for routing all service requests to suitable servers. In practice, this facility has a much more general applicability and the operation of this level will be explained in much greater detail in the next chapter.

The stated objective, of making the uppermost interface to the system as uniform as possible, is achieved through the use of this layered structure. By producing a broad range of services for network operation, and constructing software packages that are oriented towards the use of these network services, application

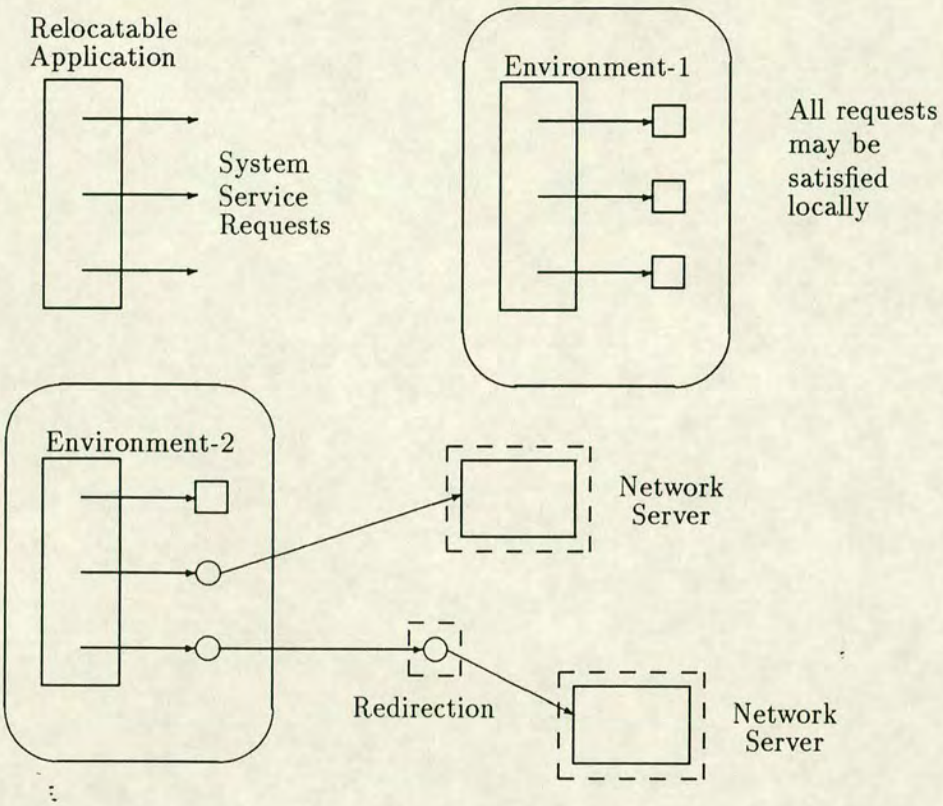

Figure 3–12: Portability of Applications

software may be readily ported between different network nodes. This is true not only of similar nodes, such as two PWMs, but may also be the case for a PWM and a BEM.

Figure 3–12 illustrates the portability made possible as a result of this uniform applications interface. The application software is shown at the top left of the diagram, with the arrows denoting requests for system services. This application may be used in environment 1, in which all service requests are locally supported and no access to the network is required. Alternatively, environment 2 provides only a small number of the required services, necessitating use of the network for provision of the remainder. With environment 2 it can be seen that the application

needs no changes in order to make it operational, even in those cases where some degree of redirection is necessary.

3.3.2 Operation of the System

The *service-based* operation of the system is best illustrated by considering a specific example. So, this section will concentrate on the potential integration into the system of two contrasting varieties of node. Whilst these two devices are extremely dissimilar in nature and rôle, they provide support for similar forms of computation and each will benefit from the involvement of the other.

Characteristics of the Nodes

The first of the devices to be considered here is the **Sun workstation**. A number of different models exist in the Sun range of workstations, but they are all based on powerful microprocessors, such as the Motorola 68020, and have an attractive programming environment with the following features :

- The display supports high resolution interactive graphics, and the system interface incorporates an easy to use window and mouse package.

- An extensive amount of software is provided for use as the building blocks for advanced applications, and assistance in the production of new applications is provided by development tools that are integrated to allow close interaction with each other.

- The large virtual memory and standard networking facilities allow the Sun workstations to fully participate in multiple system configurations.

The networking capabilities of the Sun workstations are of special interest because there are some models that do not have integral discs, and so rely on the ability to demand page between local memory and a network fileserver. The NFS protocol developed by Sun Microsystems has become a *de facto* standard for multivendor Unix-based networks, and is now being implemented on the IBM MVS mainframe operating system.

The second node to be considered here is the **Meiko M40 Computing Surface**, based on 10 4-transputer processor boards and a display system, with a MicroVAX acting as a front-end. The system installed at Edinburgh University has been used in both single and multi-user modes, but the latter is now the more common because of the demand for access to the Occam 2 development system. The M40 system is providing valuable experience in advance of the installation of the **Edinburgh Concurrent Supercomputer**, which will have over a thousand transputers. For particular applications, this larger system will provide a considerable performance advantage over existing computers.

The Meiko M40 allows up to 32 users to access the machine at any one time, with each terminal having a transputer allocated to it. Whilst this mode of operation is ideal for enabling familiarisation with the Occam 2 development system, it is not as suitable for the new system. This is because the Edinburgh Concurrent Supercomputer is intended for use in applications that require a large number of processing elements to cooperate in the computation. In these circumstances, it may be argued that the machine should act as a shareable processing resource to

be allocated to tasks as required. When viewed in a networked computing environment, the Meiko Computing Surface is a good example of a TNM Back-End Module.

Node Interactions

The potential for cooperation between the Sun and the Meiko is perhaps not immediately obvious. Indeed, the design of the Meiko machine is such that is inappropriate for the interactive programming environment offered by the Sun, and the workstation is ineffective in compute bound applications. However, in the event that more processing power is required than the Sun workstation can provide, and the nature of the computation is such that it maps well onto the architecture of the Computing Surface, the code and data could be transferred to the Meiko for processing. Whilst such instances may be few in number, if the workstation were to exploit the Meiko on these occasions then the quality of service provided for the user would be enhanced.

In the R & D division of a company, a group of development engineers, each with a Sun workstation, may share access to a Meiko node, as well as other facilities, using a general purpose network. In this environment, both the workstations and the Meiko Computing Surface are used effectively.

A succession of transputer add-on boards are appearing on the market now [61] and it may be possible to include boards such as these in the Sun workstations. The engineers may then develop code to execute on their local transputer boards before using the Meiko Computing Surface for more demanding activities.

Use of the Model

For an application in which the Sun workstations and the Meiko interacted extensively with each other, but not at all with other network devices, it would be appropriate to develop software on both machines to support this communication efficiently. In the more general, and more typical, case where interactions with a number of different network devices are to be anticipated, the adoption of a framework, within which all of these devices may communicate, is desirable.

Figure 3–13 represents one possibility for a configuration of TNM modules involved in the provision of a Meiko service for the benefit of workstation users. The NSMs used to mediate between the Sun and the Meiko, perform resource allocation and scheduling tasks, to allow the Meiko Computing Surface node to be shared amongst a number of clients. The network fileserver provides the most suitable buffer for data to be tansferred between the workstation and the back-end processor. The workstation may build up a set of data on the fileserver as a result of interactions with the user. Once processing of this data is required, a request is issued to the managerial NSMs which, in turn, direct the Meiko to perform the computation. Since the results are returned to the same network fileserver as the one holding source data, the user may readily access the output, exploiting the powerful display features of the Sun workstation.

3.3.3 Summary

This chapter began by considering the bringing together of workstations and mainframes. The approach of having two distinct networks, *front-end* and *back-end*,

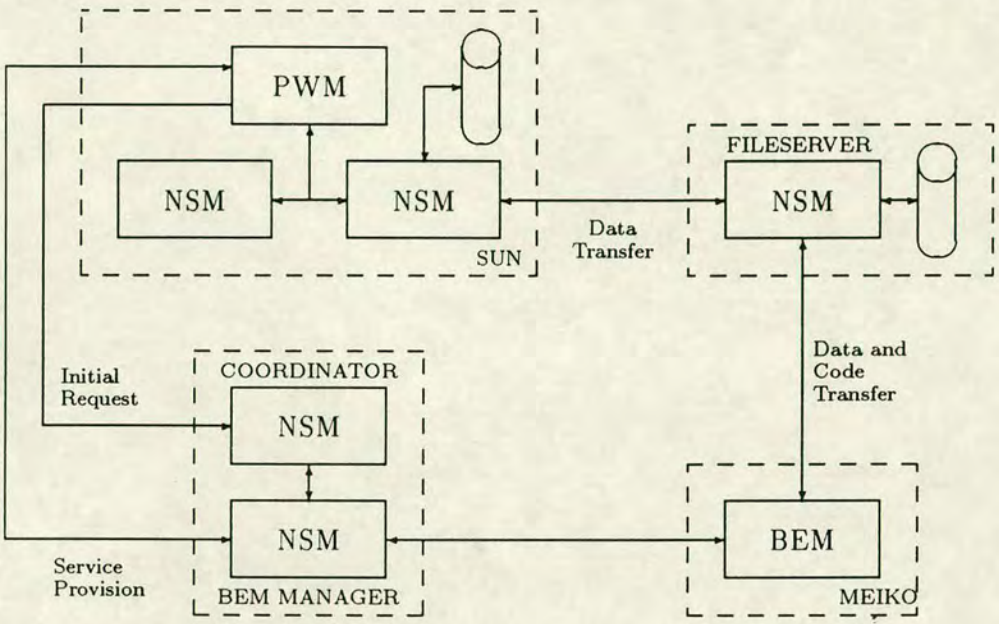

Figure 3–13: Interaction of Sun workstation and Meiko Computing Surface

figure 3–1, should be compared with the MU6 philosophy, section 1.2.1. This division of the networks is undesirable because it restricts the sharing of data between the machines on the network, and ultimately it limits the level of cooperation that may be achieved by these machines. The adoption of a single common network, figure 3–2 in section 3.1.1, permits much greater freedom of communication.

Whilst the presence of a unifying network allows data to pass freely between workstations and mainframes, true cooperation between the machines is not achieved because of their great dissimilarities. The classification of all network devices into a small number of groups permits the communication requirements within and between these divisions to be identified. Subsequently, protocols may be devised that efficiently support these communication requirements.

The simple three-way device classification of section 3.1.2 is insufficient for

these purposes because of the degree of variation that occurs in each division. The Triadic Network Model addresses this problem by making a more rigid classification, and allowing individual devices to be formed from a number of modules, each of which complies with this definition. The versatility of this approach has been illustrated by considering specific examples of network device. In studying the rôle of the model in allowing interactions between a workstation and a specialized processor, it was apparent that the potential for cooperation could be realised.

In the next chapter, the principles of a set of protocols, founded on the Triadic Network Model, are described. These protocols are oriented towards the interactions defined by the model, and incorporate the idea of a layered network transparency.

Chapter 4

Protocol Set Principles

4.1 Introduction

This chapter explains the principles of the protocols that have been derived from the Triadic Network Model. The concepts described here form the basis for a set of protocols intended for use in a heterogeneous multiple computer system. The application of these ideas, together with the details of the implementation, are discussed in the next chapter.

Before delving into the intricacies of the principles behind the TNM protocols, it is worthwhile to consider the context in which this work should be viewed. In this respect, the relationship to other network protocols needs to be studied. There are two aspects to this :

1. The nature of the interfaces between the TNM protocols and other parts of the system.

2. Where the TNM protocols lie in the world of protocol standardization.

4.1.1 Perspective

The first priority is to consider how the TNM protocol set is related to the ideas described in the previous chapter. Thereafter, the nature of the interfaces to other protocols and software layers will be more apparent.

Relationship

In section 3.1.1, a general network structure was presented as representative of configurations that are becoming increasingly common. The most important characteristic of this structure was that all nodes were interconnected by a common network. The network nodes comprised a wide variety of devices, ranging from simple terminals to sophisticated processors, but the underlying structure did not have a rigid formation and there were no constraints imposed on interactions between nodes.

The networked computing environment, illustrated in figure 3–7 of section 3.2, is strikingly dissimilar to the general network structure. The environment has a well defined formation, derived from a multitude of modules that are dedicated to the support of a particular function. This modular, service-based computing environment may be readily exploited by users and application packages alike. However, it is not immediately apparent how the networked computing environment may be provided by the general network structure.

As explained in the remainder of section 3.2, the Triadic Network Model exists to *'bridge the gap'* between these contrasting formations (figure 4–1). In particular,

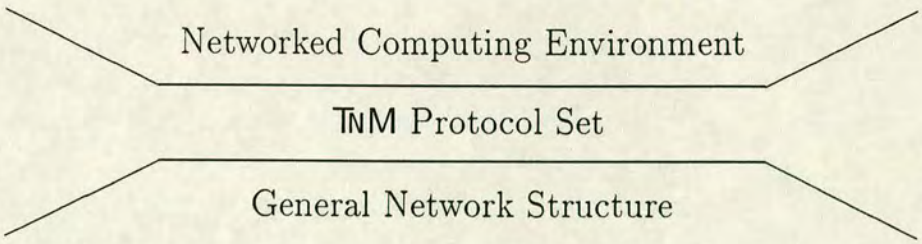

Figure 4–1: "Bridging the Gap"

the protocol set, based on the model, forms an interface between the networked computing environment and the general network structure.

Lower Interface

The model assumes that the underlying network communication substrate provides a reliable means of passing information from one node to another. Hence, the TNM protocol set is intended for use above a transport service that guarantees no duplication or mis-ordering of packets. However, there is no requirement for the transport service to be *connection-oriented.* [1] In fact, as explained in section 4.3, the TNM protocols may exploit the different merits of a number of transport services.

The primary requirement is only for a reliable communication mechanism, and not for a specific type of transport service. Hence, if the sub-net could provide a high reliability without the need for extensive coding and error checking, it would

[1] A connection-oriented protocol requires a virtual circuit to be established between the two communicating nodes, and that this circuit be maintained for the duration of the communication period.

be acceptable to implement the TNM protocols directly above the network. This would greatly increase the performance of the system by reducing the software overheads associated with network communication.

Minimisation of protocol layering so as to achieve greater system performance has the effect of removing the potential for interaction with other systems. However, in many distributed computing systems the emphasis tends to be on performance at the expense of compatibility because the systems are much more likely to remain *closed*. Where communication with other systems is desired, it is usual to provide gateway nodes for this purpose.

The Apollo DOMAIN, section 2.6.1, and the Amoeba distributed system, section 2.4.3, are examples of other systems that have opted for non-standard network protocols that are oriented towards specific applications. In both of these cases, the network protocol has had to accommodate some of the facets of the lower level protocols that have been bypassed. In this respect, the TNM protocols are different in that they assume the reliability that the lower level protocols normally provide. So, to bypass these lower levels, a sub-net must be used that already possesses a high reliability.

The Centrenet high performance network, described in section 1.2.3, together with the Centrenet Burst Protocol, is capable of satisfying the communications requirements of the TNM protocol set without the need for a sophisticated transport service. So, in figure 4–2, the transport service and network layer serve only to increase the connectibility of the system and, for a system based entirely on Centrenet, they may be omitted in the interests of performance.

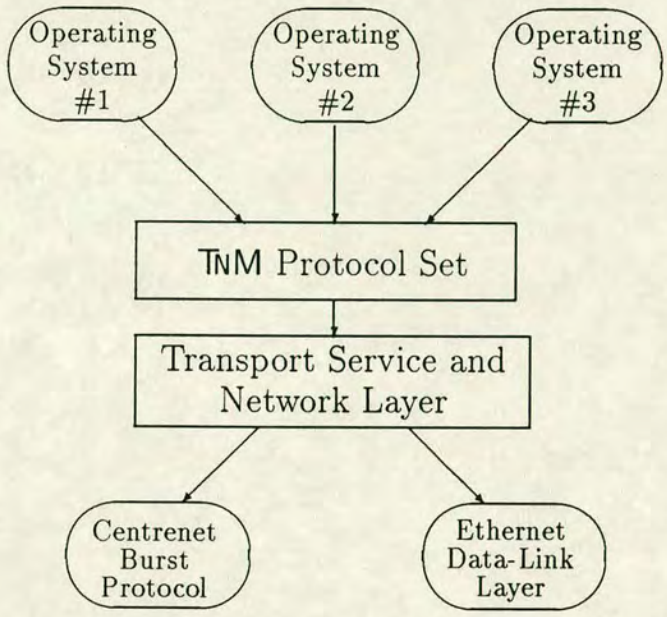

Figure 4–2: Protocol Set Interfaces

In the next chapter, it is explained how, in the definition of the operation of the TNM protocols, the dependence on a specific transport service is avoided.

Upper Interface

From figure 4–2, it is apparent that the TNM protocol set may be used by a number of different operating systems, acting concurrently within a single networked computing environment. This is to enable existing high level software to be used wherever possible with the aim of increasing flexibility in the implementation.

To allow a large degree of independence in the use of TNM protocols, the upper interface is simple, functional and oriented towards the needs of the networked computing environment described earlier. Since the primary use of the protocol set is for accessing network services, the upper interface allows a *request* to be *issued* to the system and a *reply* to be *received.* This permits either a remote

procedure call [77] or a message passing [31,74] mechanism to be implemented, as desired.

4.1.2 OSI

Background

In 1978, the **International Organization for Standardization** (ISO) began work on developing a communications architecture that would help to remove the difficulties involved in networking machines of different manufacturers together. In 1980, a proposal was produced for a **Reference Model for Open Systems Interconnection**, and in 1983 this proposal was accepted as an international standard. Since then there has been a worldwide concentration of effort on implementing networked systems that comply with this reference model.

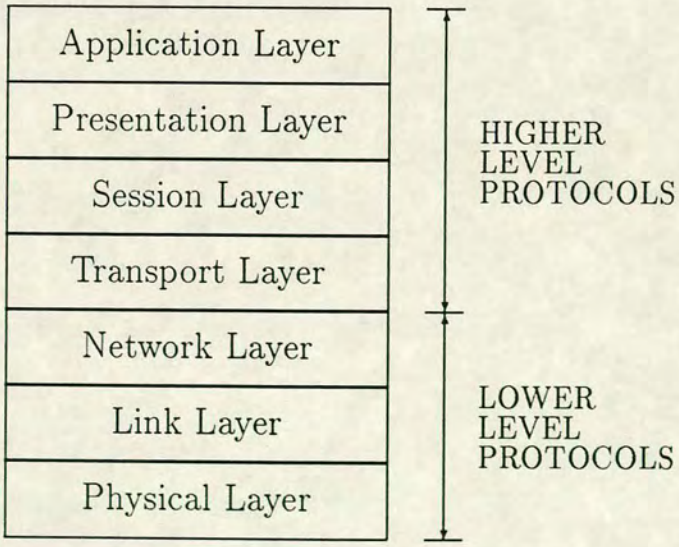

Figure 4–3: ISO Model for Open Systems Interconnection

The ISO reference model for OSI is based on a seven layer communication architecture that is similar in form to the network architectures of IBM's SNA and DEC's DNA, as well as those of other manufacturers. Figure 4–3 shows the seven layers of the ISO reference model. The seven layers may be considered in two groups :

- The **lower level protocols** are concerned with ensuring that data is correctly transferred from one node to another.

- The **higher level protocols** must ensure that the data is in a suitable form for its proper use at the destination node.

The merit in dividing the activities of these two groups into a number of layers is that it is then possible to isolate specific duties for each layer. The rôle of each layer may be clearly defined, together with the interfaces to higher and lower layers. In the case of ISO, since the OSI reference model became an international standard in 1983, specifications have been produced for each of the layers of the model in turn. A number of these specifications have subsequently been adopted as international standards.

Merits of OSI

A great many manufacturers are endeavouring to provide network systems that conform to the specifications for the layers of the ISO reference model. This is particularly true of small and medium sized companies who do not have an established and well-supported proprietary network on offer. For these companies, OSI affords them access to a wide marketplace, interested not so much in the

identity of the individual manufacturers, but more concerned with the features of the products on offer. In turn, customers that choose an OSI approach for their networks have the benefit of a broad range of products and suppliers and the flexibility of being able to spread their networking requirements amongst a number of companies.

In the light of the significant advantages of open systems, it may seem surprising that there remain a number of institutions that refrain from full commitment to the ISO/OSI. However, there are several good reasons for this.

1. The family of protocols developed on the US Defence Department's ARPAnet have been in existence for longer than the ISO/OSI. Furthermore, the Berkeley 4.2 version of Unix, popular amongst universities, is supplied with an implementation of these protocols. Hence, a wide community of TCP/IP users has emerged, and the breadth of products supporting these protocols is large enough to allow open systems to be constructed from them.

2. Large corporations, such as IBM, have their own well-established approaches towards computer communication. For many customers, who are already tied in to the product range of a particular manufacturer, the networking solution of that manufacturer may be the only approach that it is viable to consider.

3. Highly specialized applications can afford to rely on non-standard network products when the emphasis is on efficiency rather than interconnectivity with other systems.

4. For similar reasons to the previous point, distributed computing systems may adopt a reduced layered communications architecture in order to provide a suitable operational performance and behaviour of the composite system.

Protocols Based on the Triadic Network Model

The TNM protocols are more concerned with the management of data and job control information being transferred over the network than with the issues of reliable sequential transmission. Hence, they reside in the higher level protocol division of the ISO reference model. Furthermore, from section 4.1.1, it ought to be clear that, for most sub-nets, the TNM protocols reside at a level in the ISO hierarchy above that of the Transport Layer. However, the range of services provided by the TNM protocol set do not fall neatly into the defined rôles of any of the three highest layers of the ISO protocol stack.

Primarily, the functions of the TNM protocols mirror those of the ISO reference model's session layer. These functions are oriented towards the control of dialogues between different users of the session service. More exactly, a dialogue may be initiated and terminated; it may be interrupted and then resumed later at the same point. The different stages in the dialogue may be synchronized with respect to each other so that an entire session need not be repeated in the event of failure, but only the affected stages.

These dialogue control facilities are most appropriate for lengthy sessions, especially where long distances exist between the various users. The Triadic Network Model is based on more localized communication of shorter duration, and so some of the features of the ISO session layer are inappropriate.

- The resource control and allocation considerations for the suspension and continuation of the use of a service in the Triadic Network Model are too intricate for protocol functions at this level. Hence, they are more appropriately controlled by the operating system and need not be implemented by the TNM protocols.

- The partitioning and synchronization of different stages during the period of communication is inefficient for short control messages, and unnecessary in the transfer of compact, typed blocks of data. It is more appropriate for the transfer of large quantities of data. So, the TNM protocols incorporate a synchronization mechanism for traffic such as bulk data transfer, but this mechanism is not used for the majority of messages.

In figure 4–2 and in section 4.1.1, it is indicated that the upper interface of the TNM protocols is expected to be to the operating system or applications software. However, it is anticipated that, for some environments, a thin application layer may be needed above.

The ISO presentation layer is concerned with data format conversion to accommodate the differences in data representation of unlike computers. Since all data communicated in the Triadic Network Model is accompanied by its type, as described in section 4.3.4, it is envisaged that dedicated servers will exist to convert between different data representations. Consequently, the functionality of the presentation layer may be provided in the hardware and firmware of a network node.

As with many experimental distributed computing systems, the primary concern here is for a high level of efficiency. Furthermore, in this instance a high performance is also required. Therefore, compliance with the protocol layers of the ISO/OSI is not of great importance. Yet, to enable the ready inclusion of a number of different machines and existing software, it is desirable to interface cleanly to the ISO protocol stack at some level. The most pertinent layer is the Transport Layer, as indicated in section 4.1.1. However, for the implementation described in the next chapter, the unavailability of ISO protocols on the development machine meant that UDP/TCP/IP formed the lower layer interface.

4.2 Three-Party Mechanism

Fundamental to the operation of the entire TNM protocol set is the **Three-Party Mechanism**. The rôle of the mechanism is to re-route systematically any single request for use of a network service to a node capable of providing that service. The implementation of this mechanism in the TNM protocol set allows a uniform network interface to be provided to the applications software.

Relationship to the Model

In the previous chapter, the concept of a layered network transparency was introduced as a means of reducing the complexity of implementing protocols based on the Triadic Network Model. Figure 3–11 illustrated the various levels of two-way decision making that could be identified :

- The **User Level Interface** determined whether the service request was to be implemented locally or remotely.

- The **Network Level Interface** was concerned with routing the service request to a suitable node.

- The **Service Level Interface** distinguished between services that could be provided by a *standard* server and those needing access to *specialized* servers.

The Three-Party Mechanism is the realization of the Network Level Interface of figure 3–11, and consequently holds together the entire protocol set. The operation of the mechanism is quite straightforward and is detailed in the next section. Subsequent sections discuss various aspects of the Three-Party Mechanism that serve to enhance its functionality.

4.2.1 Operation of the Mechanism

The three parties involved in the operation of the mechanism are :

Service Requester (*S-R*) : initiator of the service request.

Service Provider (*S-P*) : the node that actually satisfies the service request.

Service Manager (*S-M*) : if a service request has to be routed to another node, then the node that performs the re-direction becomes the Service Manager for the duration of the session.

In the implementation of a service, there may well be even more than three nodes involved. This would be as a result of a number of re-directions. In this instance, there would be several instances of the Service Provider and Service Manager but only one Service Requester.

An example of this more complicated use of the Three-Party Mechanism will be considered later. First, though, the principles of operation of the mechanism should be explored.

Simple Case

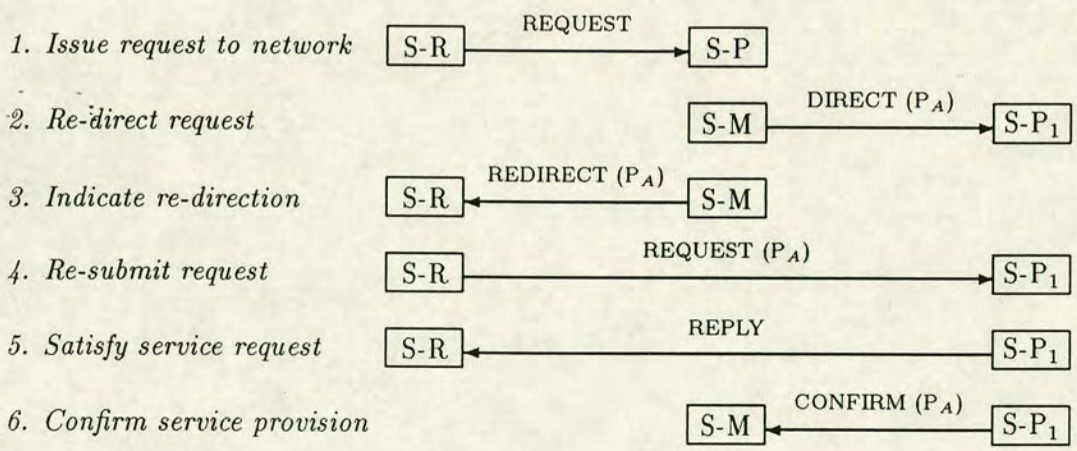

Figure 4–4: 3-Party Mechanism

Figure 4–4 illustrates the various stages involved, with the sequence being from top to bottom. Every transfer of a protocol primitive is shown on a new line, with each node involved in the transfer represented by a box containing the node's identity. For clarity, each network node involved in the provision of the requested service is always shown in the same column. An arrow indicates the direction of the transfer and the mnemonic above the arrow is the name of the protocol primitive. Where

a character sequence appears in brackets after the protocol primitive's name, it represents the presence of a service permit, section 4.2.4, and the sub-scripted character identifies the permit. The conventions used in figure 4–4 are followed for other illustrations of protocol transfer sequences.

The six stages that comprise a single instance of the Three-Party Mechanism are :

1. When a network-supplied service is required by an application, but cannot be satisfied locally by the network node, a **service request** is made to the system. The network node then becomes the **Service Requester** *(S-R)*.

2. If the recipient of this request is capable of providing the service, it satisfies the request and then sends a response back to the *S-R*. If it cannot provide the required service then it will either return an indication of its failure to the *S-R* or it may re-direct the request to a node which it believes to be capable of supporting the desired service. If this node re-directs the original request, it becomes the **Service Manager** *(S-M)* for the duration of this **session**.

3. To re-direct the request, the *S-M* must send an indication to the intended node that it should anticipate a service request. The *S-M* must also supply the intended node with a means of verifying the service request. It achieves this by sending a **Service Permit**, section 4.2.4, with the indication of the request re-direction. The node to which the service request is re-directed is the **Service Provider** *(S-P)*.

4. The *S-M* must also return an indication to the *S-R* that the service request should be re-directed, and again the service permit should be provided.

5. If the *S-R* receives, in reply to a service request, an indication that the request is to be re-directed then it should re-submit the service request, together with the corresponding service permit, to the substitute server, *S-P*.

6. The *S-P*, upon receipt of the request, should first ensure that the service permit is correct and then satisfy the request. The response should then be returned to the *S-R* and finally confirmation provided to the *S-M* that the service request has been satisfied.

In figure 4–4, there would appear to be two instances of *S-P*, one of which has a subscript of 1. This represents the fact that, at the initiation of the service request, the node to which the REQUEST is sent is viewed as a prospective provider of the service. It is only when this node decides to re-direct the request, and therefore assume the rôle of *S-M*, that a new provider becomes apparent. This new provider is suffixed by a 1 to distinguish it from the previous provider.

The fact that the un-subscripted *S-P* appears in the same column as the *S-M* serves to illustrate the fact that they are both associated with the same network node. Consideration of a slightly more complicated use of the mechanism may help to clarify this point.

Complex Case

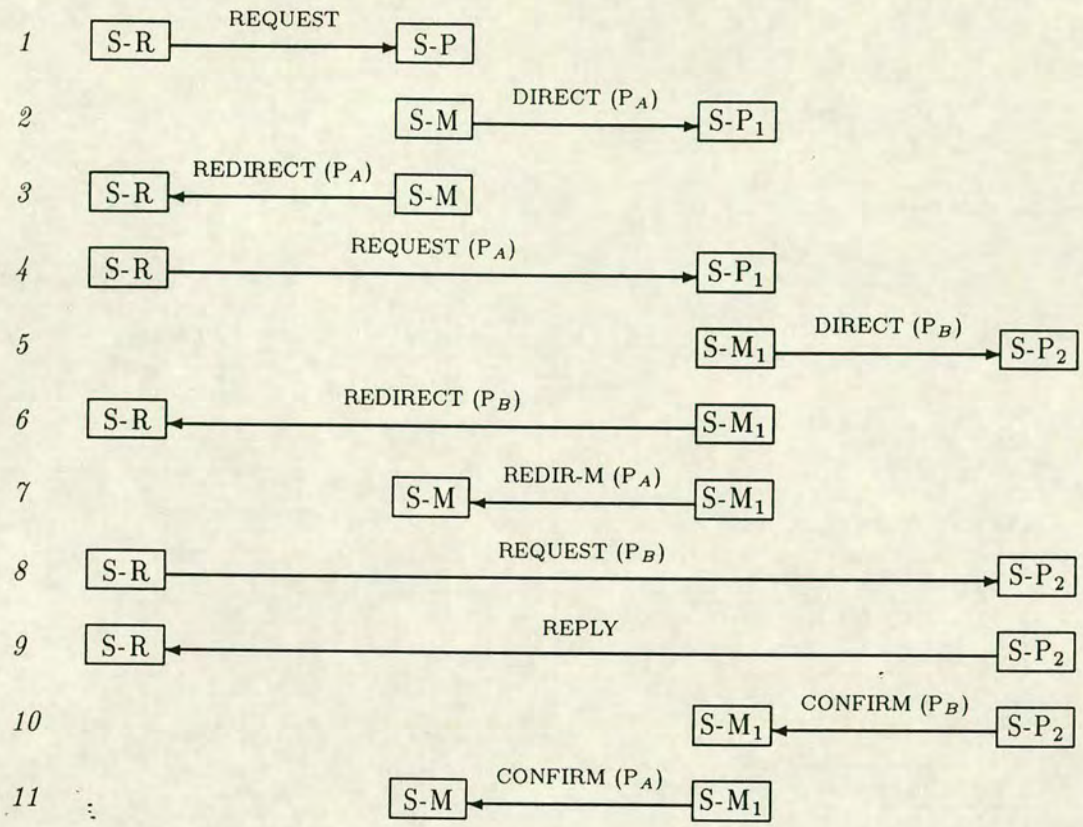

Figure 4–5: More Sophisticated Use of the 3-Party Mechanism

In figure 4–5, the service request is re-directed twice before eventually being satisfied by $S\text{-}P_2$. This may occur when the first recipient of the REQUEST has a limited awareness of suitable nodes that may provide the service, and so it refers the request to a better informed node. The presence of four columns of boxes indicates that four network nodes are involved in this service provision; there are three instances of *S-P* and two of *S-M*.

Stage 7 of figure 4–5 involves the transfer of a primitive not encountered in the simpler case of figure 4–4. $S\text{-}P_1$ has received an indication from *S-M* that a service request should be expected with permit P_A. However, when the request is received, $S\text{-}P_1$ discovers that it does not support the required service. So, it

assumes the rôle of a service manager, sends an indication of an impending service request to S-P_2 and informs S-R of the re-direction. This much is common to both examples. However, because S-M_1 is the result of a re-directed request, it must indicate to S-M that the request has been further re-routed. This ensures that S-M is in a position to fulfil its rôle as service manager when required, as explained in the next section.

Algorithms

The behaviour of the three parties may be expressed in an algorithmic form.

```
S-R :  n ← 0
       permit ← STANDARD-PERMIT
       satisfied ← FALSE
       repeat
            issue request to S-P_n with permit
            receive response
            if response = REPLY
                 then
                           decode response
                           return response to user
                           satisfied ← TRUE
                 else
                           n ← n + 1
                           permit ← permit of response
            endif
       until satisfied
```

```
S-P :  receive request
       if can satisfy request
            then
                 implement request
                 send response
            else
                 become S-M for request
       endif
```

```
S-M :  select provider S-P
       select permit
       send DIRECT to S-P with permit
       send REDIRECT to S-R with permit
       concluded ← FALSE
       repeat
            receive response
            if response = CONFIRM
                 then
                           concluded ← TRUE
                 else
                           concluded ← FALSE
            endif
       until concluded
```

These algorithms reflect the behaviour of the three parties when there are no problems in the service provision. As such, the algorithms may form the basis for implementations of the Three-Party Mechanism, and thereby for the TINA-M protocols also, but additional measures are needed to cope with any error conditions that may arise.

4.2.2 Rôle of the Service Managers

Of the three parties involved in the provision of a service using the Three-Party Mechanism, the Service Manager *(S-M)* is of special significance. The actions of the service requester and the service provider have much in common with the standard **client-server model**, which forms the basis for most network communication protocols. The presence of a node that performs a supervisory rôle in the interaction is more unusual.

There is some similarity between the rôle of the CSS node in the LOCUS *Open protocol*, section 2.3.2, and that of the Service Manager. However, the CSS node is particularly concerned with controlling file accesses, especially the problems of synchronization of different copies of a file. In this respect the operation of the CSS is very important in the *Open protocol* but the Service Manager has a less critical function.

Recovery from Failure

In the general case, as described in the previous section, the manager's only purpose is to route the initial request for use of a network service to another node that is more capable of providing the service. This action of re-direction is short-lived, and if this were the limit of the manager's ability it would be appropriate for the *S-M* to discontinue its involvement in the communication session. However, in the event of a failure in the communication between *S-R* and *S-P*, the *S-M* is in a position to assist in the recovery of the session.

A Service Manager is *'accountable'* for the provision of a service until the request for that service has been satisfied. This responsibility is of greatest importance in the event of a failure in the provision of the service. In this case, the *S-R*, the *S-P*, or both, will communicate with the *S-M* to indicate the evidence that a service failure has occurred. The recovery actions that may be taken by the manager will be dependent on the degree of information available to the manager and the nature of the failure. The use of Service Permits is of assistance in the event of failures, as described further in section 4.2.4.

Multiple Service Managers

In the more complex example of the operation of the Three-Party Mechanism, figure 4–5, it is apparent that there may be more than one service manager in operation at any one time during the session. This situation arises as a result of a number of re-directions being made to the service request. However, despite there being a multiplicicty of *S-M*s involved in the session, the *S-R* and *S-P* are only concerned with the *current S-M* at any stage in the provision of the service.

In figure 4–5, if $S\text{-}P_2$ failed during stage 8 of the communication session, the *S-R* would select $S\text{-}M_1$ as the first line of enquiry towards recovery of the service. If $S\text{-}M_1$ has also failed the *S-R* will then communicate with *S-M*. For $S\text{-}P_2$, if *S-R* were to fail in stage 8, $S\text{-}M_1$ is the only manager with which it may interact to recover from the service failure.

This use of a *chain* of service managers serves to improve the robustness of the system. So, if there is a breakdown in the process of re-directing a service request to a small succession of network nodes until a suitable *S-P* is selected, the *S-R* is in strong position to be able to work back up the chain of *S-M*s until being further re-directed to a healthier *S-P*.

Burden of Re-direction

It is important to dispel any undue concern that may have arisen over the potentially large overheads that could be present in a system that relies heavily on long sequences of re-direction. For a well-established system, most of the nodes will have a high level of knowledge about ideal devices for particular services. In

such an environment, service request re-direction will be of greatest importance for new members of the network and ill-informed network devices.

The Three-Party Mechanism may be used to improve the security of a system. In requiring all devices to send requests to specific 'validation' nodes, which in turn re-direct the requests to suitable service providers, access to valuable resources may be restricted to privileged users only. The use of Service Permits and the Coordinator, described in the following sections, serve to assist in this aim.

It is considered unlikely that the number of nodes involved in the provision of a single service will extend beyond the more complex example of section 4.2.1, illustrated in figure 4–5. The only exceptions to this rule would be for services requiring use of Back-End Machines (BEMs).

BEM Managers

The manager of a BEM service will act, wherever possible, as if it is providing the service itself. This is so as to receive all of the information from the *S-R* necessary for the BEM manager to instruct one or more BEMs to provide the required service. The limit of the BEM manager's *'boast'* is that, where the transfer of large amounts of data is involved, it would be inefficient to introduce an additional delay by requiring this data to pass through itself.

So, after initiating the implementation of the required service by suitable BEMs, the BEM manager may then re-direct the original service request to an NSM to be used as a data buffer for the duration of the service. Hence, the BEM manager assumes the rôle of *S-M*, but with a higher level of responsibility for the service than the *S-M*s already considered.

In the event of a failure in the provision of the service, the BEM manager is the most suitable node to be required to recover the service because of its detailed knowledge of the means of implementing the service. With the BEM manager as the *current S-M*, it will be selected by the *S-R* as the first line of enquiry towards failure recovery. If the nature of the failure is such that the BEM manager cannot help *S-R*, the alternatives presented to *S-R* are to either abort the request or move back up the chain of *S-M*s in an attempt to re-route the request to a new BEM manager.

This instance provides a good illustration of the merits of maintaining the chain of service managers.

4.2.3 Coordinator

All of the nodes on the network need only be aware of a single *logical* node, known as the **Coordinator**, to which all requests for use of network services may be sent. In turn, the coordinator may use the Three-Party Mechanism to forward any service requests to other network nodes capable of providing the required service.

Multiple Coodinators

If this single *logical* node were to be implemented by a single *physical* node, the coordinator would form an unacceptably vulnerable bottleneck in the system. So, in reality, there should be a number of coordinator modules within the system to spread out the load on this service. Additionally, this provides extra reliability

and redundancy in the system, reducing its susceptibility to total system failure resulting from the failure of a single node.

The notion of a single coordinator serves to simplify the interface between network services and the software on a user node. Furthermore, it helps to conceal the heterogeneous nature of the multiple computer system. By implementing the coordinator module using a number of nodes, the software controlling a node's requests for network services must map all requests to the coordinator onto the set of coordinator modules. Alternatively, where the network allows the use of *generic addressing*, this provides a more efficient approach.

Centrenet

In Centrenet, the Network Intelligence Modules NIMs are ideally suited towards use as coordinators. Every Starpoint of the network has the capacity for a NIM, and the NIM is intended for mapping calls to virtual addresses onto real devices.

The Centrenet hardware, as it stands, requires that all NIMs be explicitly addressed. However, a straightforward modification to the interface logic of the NIMs would allow other devices to address the nearest operational NIM by presenting a single generic address, eg 0000. The tree-like structure of Centrenet would ensure that, at the very least, a request to the coordinator would be answered by the SuperNIM at the root Starpoint.

Operation

It would prove to be unnecessarily burdensome for all network service requests to have to pass through the coordinator. This is especially true of requests for elementary services, for which the indirection introduced by the coordinator would seriously deteriorate the performance.

The solution is to allow some *'caching'* of network addresses of certain services so that service requests may be made directly to the provider without any need to access the coordinator in the first instance. For this solution to operate, there must also be some measures taken to ensure that the service permits continue to be correctly validated and monitored. This is explained in the next section.

The coordinator is effectively just a specific instance of a service manager, but is considered by the higher levels of software to be permanent rather than transient, as is the case for most service managers. It is the only service manager visible to the higher level software, in that the software may presume that any service request sent to it will in turn be forwarded. However, no special consideration need be taken of this fact since, as far as the TNM protocols are concerned, it behaves just like any other NSM.

4.2.4 Service Permits

The use of **Service Permits** enables some necessary flexibility to be introduced into the use of network services, whilst retaining security and control over resources. A service permit is, in essence, a *capability* or *token* enabling a node to

make use of a particular network resource, but the applicability of service permits is much broader than this.

Currency and Security

Service permits are issued to customer nodes, that have requested use of a service, by a service manager when it is incapable of satisfying the request, but is aware of a more capable node, *S-P*. Prior to issuing the permit, the *S-M* must first issue an *indication* to the *S-P* that it is to answer a re-directed service request. The *S-M* then replies to the original service request, returning the service permit. The presentation of a service permit by *S-M* to the *S-R* implies that the request has been authenticated and scheduled, if this was necessary. On receipt of the permit, the *S-R* re-submits the service request, including the permit, to *S-P*. Having re-submitted the service request, with the permit, *S-R* has effectively *presented* the permit to *S-P* and thereby relinquished it. On completion of the service, *S-P* confirms completion of the service to *S-M*, and *returns* the used service permit.

Hence, the service permit can be viewed as a form of currency and the *indication*, as passed from *S-M* to *S-P*, as a receipt for permits issued. The permit only has a limited period of validity and the interval between the issuing and re-issue of a permit is much greater than this validity period. This aids the security of the system by making it difficult for 'rogue' nodes to break security keys embedded within the service permit. For a system where security is not a major concern, it is possible for 'block allocations' of service permits to be made, such that a given node may make direct requests for use of a service without resort to obtaining a

permit from an *S-M* first. The node must also have formed a table of the network addresses for the services corresponding to the service permits.

Service Requester in Control

Service permits, as used in the Three-Party Mechanism, allow the requester of a service to retain control over its use of that service. This is because the node in receipt of the service request must either implement the requested service, signal its inability to do so or return a service permit to allow the *S-R* to re-issue the request to another node. In the latter case, the new *S-M* can make arrangements for the re-direction of the request, but it is not empowered to actually forward the request to other nodes.

Hence, at every stage in the service session, the *S-R* is always aware of which node is acting as *S-P*. [2] This allows the *S-R* to terminate the request at any time, either as a result of an explicit instruction from the user or due to the *expense* of the service exceeding a pre-defined limit. In most cases, the *expense* will be measured in terms of the time taken to satisfy the request. In systems where access to certain resources is limited, and accounting is therefore more important, the ability of the customer of the service to decide on the level of re-direction that it can *afford* is essential.

There are also practical considerations that reinforce the decision to have a mechanism that requires the customer of the service to re-submit the request

---

[2]When an *S-P* has transformed to an *S-M*, the new $S\text{-}P_1$ is not recognized as such until the *S-R* has received the re-direction information and re-issued the service request.

whenever re-routing is necessary. If the *S-M* were to be required to forward the request to the *S-P*, and act as an intermediary on behalf of the *S-R*, an extra delay would be introduced in the path between the source and destination of the data transfer. Additionally, the *S-M* would need to be able to interpret the intitial request in order to be able to forward it properly. This extra knowledge would futher burden the *S-M*.

For the software implementing the network protocols on each node, the requirement for service re-direction to be referred to the *S-R* ensures that the delays between the *issue* of a request and the *receipt* of a response are reduced. This means that reasonable limits may be used for timeouts.

Failure Analysis

The service permit may also be of help in the detection and analysis of service provision failures. The receipt of an unknown permit, the delivery of an expired permit and the expiry of a permit receipt are all signs of a failure in the service provision. So, for example :-

1. *S-R presents permit but S-P refuses to accept it*	⟶	*S-P failure or expiry of permit validity period*
2. *S-P awaits permit that is never presented*	⟶	*S-R failure*
3. *No confirmation received by S-M*	⟶	*S-P failure*
4. *S-P returns confirmation but rejected by S-M*	⟶	*S-M or S-P failure*

Stage	Nodes		Detection	Action
	Fail	Find		
1. Issue service request to network	S-R	S-M	S-R rejects reply from S-M	S-M may report this to the network monitor
	S-M	S-R	S-R receives no reply from S-M, times-out and sends enquiry to S-M	if no reply to enquiry, or S-M acknowledges failure, then S-R may notify the network monitor
2. Redirect service request	S-R	S-M	as for 1	as for 1 + S-M may terminate session
	S-M	S-R + S-P_1	S-R detects as S-P_1 for 1; notes permit expiry	S-R acts as for 1; the S-P_1 notifies S-M of non-use of the service
	S-P_1	S-R	S-P_1 rejects request from S-R	S-R notifies S-M of service failure; S-M may try to recover
3. Indicate redirection	S-R	S-P_1	S-P_1 notes expiry of permit	S-P_1 notifies S-M that service was not used; S-M may notify the network monitor
	S-M	S-P_1	S-M rejects S-P_1 confirmation	S-P_1 may notify the network monitor
	S-P_1	S-R	as for 2	as for 2
4. Re-submit request	S-R	S-P_1	S-R rejects reply from S-P_1	S-P_1 notifies S-M of service failure
	S-M	S-P_1	as for 2	as for 2
	S-P_1	S-R	S-R times-out on reply	S-R sends enquiry to S-M; S-M may try to recover
5. Satisfy service request	S-M	S-P_1	as for 3	as for 3
	S-P_1	S-M	S-M times-out on confirm	S-M sends enquiry to S-P_1 and receives reject

Table 4–1: Analysis of the Effects of Single Node Failure

Further exchanges of information between the various nodes are needed to determine the exact nature of the failure. Table 4–1 illustrates how single node failures may be detected.

4.3 Communication Mechanisms

The mechanisms used to transfer information from one part of the system to another may take many different forms. Each method of communication has advantages over the others for particular applications. Where two or more approaches to data transfer are equally suited to a specific need, the decision as to which is used is dependent on the implementation of the system. It is rare for more than a small number of different mechanisms to be supported in an individual implementation.

This section takes a brief look at some of the better known communication mechanisms, and discusses their relative merits. Subsequently, a *fundamental unit of communication* is identified, and the other forms of communication are described in terms of this basic unit. Two important facets of this unit of communication are then considered, and the exploitation of these features by a TNM based system is discussed.

4.3.1 Common Mechanisms

The following list of communication mechanisms is representative of those used in the majority of networked computing systems.

Messages

The primary use of **messages** is for the transfer of control and status information between network devices. Messages are characterized by having a small size with a

well-defined structure. This enables protocol modules on different network nodes to exchange highly specific information in an efficient and compact form.

Pages

In contrast to messages, where their content is often for use by the operating system or networking software, the information contained in **pages** is primarily for use by applications software. In having a fixed, known size, pages lend themselves naturally to efficient transfer of data between machines. This is because they may be copied directly into preallocated areas of memory without further processing.

Paging mechanisms have been shown to operate very well over a local area network [60,97]. However, pages are an inefficient means of sequentially transferring a large amount of data from one node to another. This is because of the overhead of transferring an identification header with every page and the fact that every received page needs to be explicitly acknowledged.

Bulk Data Transfer

A **bulk data transfer** mechanism involves the movement of a large quantity of data, of known size, from one network node to another. It is usual for an error correcting protocol to be used, together with the periodic exchange of synchronization information. Wide area network connection-oriented protocols, such as X.25 [64], are designed for this purpose.

Streams

Bulk data transfer mechanisms are inappropriate when the size of the data to be transferred is not known in advance. In these circumstances a **stream** communication mechanism is better. Stream mechanisms have much stronger synchronization procedures that effectively partition the data into more manageable sections. To ensure that the transmission is as reliable as possible, special markers are inserted into the data at regular intervals, together with error checking codes. Where data is lost or corrupted, it is a simple matter to determine the last marker that was successfully transmitted and to then retransmit all of the subsequent data.

Pipes

For transient data of an unknown size, a **pipe** must be used for movement of data between network nodes. The pipe mechanism is best known for its implementation in Unix, where it is used for single machine inter-process communication. In Unix, the pipe is implemented as a circular buffer, of fixed size, with one process writing to the buffer and another reading. When the buffer fills, the writing process is suspended until the reading process has consumed more of the data in the pipe.

The provision of a pipe style data transfer mechanism over a network is quite rare. This may be because some of the synchronization difficulties present in the movement of data between discrete devices are further compounded for pipes. Contrasting streams and pipes, the essential difference is that the data contents of a stream may reside at the source node until confirmation of successful transfer has been received. Whereas for pipes, much less buffering of data is used and the majority of the data is considered to reside either at the source or the destination.

Only the most recently transferred portion will continue to be held at the source awaiting confirmation of receipt by the destination node.

4.3.2 Fundamental Unit of Communication

The various forms of network data transfer mechanisms listed in the previous section are well suited for particular needs of communication between applications. Whilst it may be possible for an individual type of mechanism to satisfy a much wider set of demands, it will be much less efficient than the use for which it was designed.

For optimum performance in a wide range of applications, a different network communication protocol could be developed for each of the various forms of data transfer mechanism, but the expense, in terms of development and implementation effort, would be prohibitive. So, for the Triadic Network Model based protocols, all interactions between network nodes are based on a fundamental unit of communication, designated a **record**.

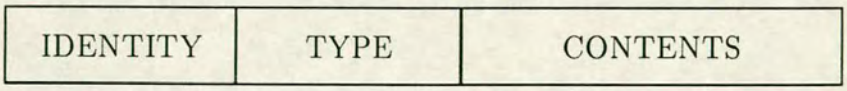

Figure 4–6: Format of Record

Every transfer of information that occurs in the networked computing environment is formed from a record. All records have a basic structure, as illustrated in figure 4–6.

In the next two sections, the use of the **type** and **identity** fields is discussed. The remainder of this section is concerned with how the data transfer mechanisms already described may be implemented in terms of records.

Messages

A message maps readily onto a record because each field of a message is synonomous with a field of a record. The type of a message is effectively the name of a protocol primitive.

The identity field may be used to ensure that every transfer is distinct and thereby idempotent. Therefore, a transmitting node may re-send a message in the knowledge that the action will not be repeated if the previous message was successfully received. Hence, an unreliable but lightweight transport protocol may be used because the identity field, together with retransmissions, enhances the quality of service provided.

Pages

Every page is required to possess a unique identifier within the system so that it may be safely located, transferred and stored by the page management system. This identifier may occupy the identity field of the record. The type field would correspond to the nature of the fixed size block of data within the contents field.

Bulk Data Transfer, Streams and Pipes

The transfer of large amounts of data is best transferred using a connection-oriented protocol. So, for a record with a *stream* as the type and a file identifier in the identity field, the TNM protocol modules would include extra synchronization information when transferring the data in the contents field. This additional control information is transparent to the application and is only used by the TNM protocol modules.

Where there are a variety of transport services available, the TNM protocols may exploit the relative merits of the different services, as appropriate. So, for the experimental implementation on a Unix system, UDP is used for messages and TCP is used for pages and streams.

4.3.3 Identity

The identity field of the TNM record serves two main purposes :

1. To allow all transfers of information to be uniquely identifiable, and thereby remove the possibility of duplication or misordering of data.

2. For holding system-level object identifiers, where data is being transferred.

The identity field allows all units of communication on the network to be distinguished. This avoids confusion when a number of sessions are active concurrently and removes the possibility of a receiver accepting duplicate information. Furthermore, recovery from failures is easier because interacting nodes have knowledge of the stage of communication that was reached before the failure.

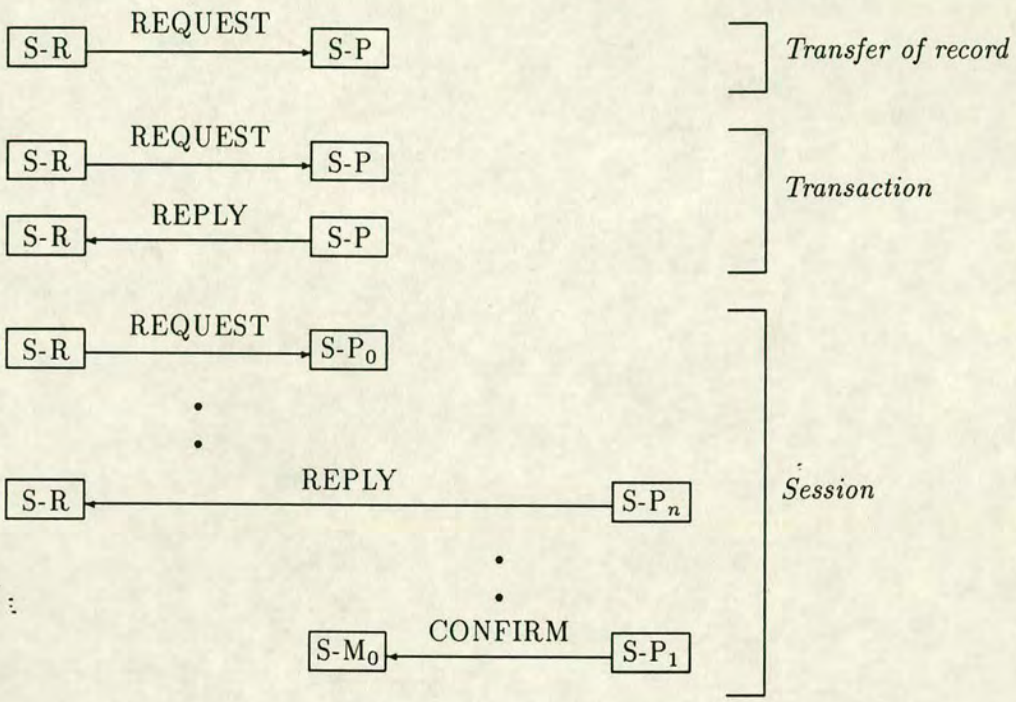

Figure 4–7: Stages of Communication

Taking the record as the most fundamental unit of communication, the various stages of an interaction are considered to be grouped as illustrated in figure 4–7.

The smallest stage is the **transfer** of a single record. A transfer may be 'stand-alone' and not have any other transfers associated with it (eg. a status report) or it may form one member of a **transaction**. A transaction is a group of record transfers between two nodes, such as a *Request-Reply* pair for service provision.

A **session** is a collection of transactions that together implement the provision of a single service-use. So, it is possible for a session to comprise a single transac-

tion or a multitude, depending on the complexity of the provision of the service concerned.

Note that it is possible for a number of sessions to commence as the direct result of a single session. The 'spawned' sessions may exist concurrently with the original, but the end of the initial session occurs when the first service request has been satisfied, not when the last of the created sessions has ended.

To allow a compact field to be used for the transfer identifier, some form of date/time-stamp should be used. This means that a small range of session numbers may be used with an added degree of confidence that the identifier value will not recycle within quite a long period.

System Level Identifiers

Where system level information is being transferred using the TNM protocols, corresponding identifiers will always accompany the data. Although this relies on a system-wide naming system being adopted by the operating system(s) in use, it has the merit of allowing high level interactions between system processes whilst using lightweight protocols.

The presence of identity and type fields are of prime importance in object-oriented systems, but the additional presence of capability information is usually required, as with the Amoeba system described in section 2.4.3. For the TNM protocols, the capability of an object should be included with the contents field and managed by the higher level protocols.

4.3.4 Type

The **type** ensures that all of the fields are correctly interpreted, and allows a wide variety of fields to be used, as appropriate to the information being transferred. Additionally, the presence of the type field permits some degree of optimization by the lower level software or the hardware.

It would be possible for the extent of the error-checking performed to vary according to the nature of the data being transmitted. The responsibility for the error-checking lies with the reliable communication mechanism provided by the underlying network. Hence, the type field may be used to select the *level of service* provided by this communication mechanism.

Optimization

The hardware of the network interface adaptors may improve the performance for network communication by handling the different types of data in an appropriate fashion. An example of the optimization that may be performed is given for the MU6V I/O-processor, described in section 1.2.2. In this case, three different classes of data were identified and the hardware would have treated each in the most suitable manner :-

messages : copied into a small area of memory local to the I/O-processor.

pages : make use of a large formatted area of virtual memory.

streams : routed directly onto the MU6V communication highway, with each element of the stream being copied into a single register.

Different Representations

By requiring that everything transferred over the network has a type associated with it (and explicitly transferred as part of it), when data is transferred between nodes that have different ways of representing the type concerned (eg. different formats for real numbers), the type carried with the data allows the receiving node to readily determine how to translate the data.

The definition, allocation and administration of type information is the responsibility of the operating system. However, it is considered that the emerging ASN1/CCITT X.409 standards are most suitable to the task of ensuring that all communicating nodes can handle the accompanying information.

4.4 Functional Division

The Triadic Network Model benefits from the identification of three distinct types of network node because of the clarity and simplicity that is introduced into considerations of the inter-node communication requirements. The consequences for a protocol set based on the model are that a smaller set of primitives is required and the protocols themselves are more streamlined.

To utilize fully the advantages of the model's discrimination between classes of network node, the TNM protocol set is partitioned into three. Each of these three protocols is oriented towards the needs of a particular type of network communication, as described by the Triadic Network Model.

This section considers the reasons for the division of the TNM protocol into three and the rôles of each of the protocols. The relationship of the protocol set division to the model is described later.

4.4.1 Purpose of Division

A key element in the discussion of the Triadic Network Model in chapter 3 was the identification of three distinct types of network device. The objective behind this was to allow the requirements of both *inter*-class and *intra*-class communication to be identified. This would permit the development of communication protocols that were optimized to these needs.

Reviewing the communication requirements and restrictions, as defined by the model, it becomes apparent that three different protocols are needed :

1. A protocol to support Personal Workstation Module communication.

2. A protocol to control the provision of conventional network services. This should provide the three party mechanism, described earlier.

3. A more sophisticated protocol to allow the specialized resources of the back-end modules to be exploited.

The three TNM protocols that are oriented to each of these needs are, respectively :

- the **Inter-User Communication** *(user)* protocol.

- the **Basic Service Provision** *(basic)* protocol, based on the three party mechanism.

- the **Special Service Support** *(special)* protocol.

The rôle of each of these protocols is considered in the next section, and their operation is described in the next chapter.

The primary merit of the division of the TNM protocol set into three more specialized protocols is that each module in the system need only support the most appropriate protocol. The consequence of this is that minimal software development is required for the inclusion of a new type of device into the system. Furthermore, the network communications software may be simpler and more efficient in terms of memory usage because a much smaller range of primitives needs to be accommodated. This may lead to higher performance levels for the network software.

The strategy of sub-dividing the communications software into smaller functional units may be extended still further by discriminating between the different parties involved in the interaction. This is most clearly illustrated by the observation that the *requester*, *manager* and *provider* of the service in the Three-Party Mechanism, section 4.2, all play different rôles. It is conceivable that a requester of services may be incapable of actually providing any services of its own. Hence, it would only be required to support the *requester* part of the protocol.

This approach has been adopted in the experimental implementation of the TNM protocols. Each *party* in a given protocol has a separate software module. These may be combined or left isolated, as appropriate for each network device.

4.4.2 Protocol Rôles

In this section, the rôle of each of the three TNM protocols will be considered in turn.

Inter-User Communication

The TNM **Inter-User Communication** *(user)* protocol is only concerned with the transfer of information directly between Personal Workstation Modules (PWMs). As described in detail in section 3.2.2, this communication is between the users of the system, as distinct from the user's programs, applications or operating system modules. For maximum flexibility, the system needs to support visual and audio transmissions in addition to textual transfers. Hence, the key objective of this protocol is the smooth but rapid establishment or termination of communication sessions, and very efficient data transfer for the duration of those sessions.

The approach taken for the TNM *user* protocol is to rely on streamlined lower level protocols for the control and transportation of the data. The higher level protocol modules need only to be involved at the beginning and end of a communication session, where the means of conducting the conversation needs to be agreed.

As Trevor Hopkins explains in [37], the protocol overheads for this form of communication can readily be minimised because the reliability of the connection is of lower importance that the network latency. This statement may be illustrated by the public telephone system : the clarity of the audio signal is often quite poor, but it is possible for the users of the telephone to understand each other. However,

when a delay is introduced into the passage of the signal from one telephone to the other, the users find greater difficulty in communicating. Even greater problems would arise if the telephone communication network were to insist on making retransmissions whenever some of part of the signal had been lost.

The model calls for quite sophisticated means of communication between users to be supported. An example of this is teleconferencing, where a multitude of users are all partaking in a 'single' conversation. The TNM *user* protocol allows a **mediator** node to be used for such communication.

The principle of operation is that the TNM *basic* protocol, briefly described in the next section, is used to request use of a 'mediation' service. The details of the conference, such as the users' identifications, are all supplied at this stage. All of the authenticated users may then 'call' the mediator just as if it were another user, and all subsequent communication is broadcast to all other users in the conference.

Other, more involved, means of communication between users may be established in this manner. The definition of the protocol is given in appendix A, but no further discussion is made of the TNM *user* protocol.

Basic Service Provision

The TNM **Basic Service Provision** *(basic)* protocol is the most important of the TNM protocol set and is likely to be used by the majority of the attached network devices. The TNM *basic* protocol forms the basis for the operation of the system because of the orientation of the model towards the use and provision of network services. The TNM *basic* protocol governs the way in which any network node

requests, and obtains, the use of a particular service provided by another network device.

Provision is made in the protocol for the re-direction of requests for services to more suitable nodes. Additionally, there is support for failure recovery with the assistance of the designated service manager for each service request. Hence, in effect, the TNM *basic* protocol is a practical implementation of the Three-Party Mechanism.

The principle of the Three-Party Mechanism was described in section 4.2. The implementation of the protocol is discussed in greater depth in the next chapter, and a more rigorous definition is given in appendix B.

Special Service Support

The TNM **Special Service Support** *(special)* protocol allows the functionality of the services provided by the system to be extended further. The protocol enlarges on the rôle of the service manager, as used in the TNM *basic* protocol, to allow the provision of more sophisticated services. These services may involve the close cooperation of a number of network nodes in the implementation of a single service. The service manager acts as if it were providing the special service, whilst in reality making use of a number of other network devices. The consequence is that the service manager is required to be much more closely involved in the provision of the service.

The primary users of the TNM *special* protocol are the high performance computers formed from back-end modules, as defined by the model. A characteristic of these machines is that direct communication with other modules is complicated.

Hence, every back-end module has a corresponding service manager which controls all access to the services provided by that module. Communication between the back-end module and its manager is governed by the TNM *special* protocol.

Using the TNM *special* protocol, the service managers can initiate the provision of intricate services by the back-end modules. In return, the managers receive regular *status reports* to indicate the progression of the service provision. The information provided in these status reports allows the service manager to provide a good *quality of service* to the customer, whilst concealing the sophistication of the details of the service provision.

The example session given in the next chapter provides a good illustration of the function of the TNM *special* protocol. The definition of the protocol is given in appendix C.

4.5 Summary

This chapter has concentrated on the principles behind a protocol set based on the Triadic Network Model. A key feature of the TNM protocols is the Three Party Mechanism. The mechanism provides a simple yet effective means to allow requests for services and their provision to be supported. A means is provided for rerouting service requests to the most suitable node. The use of service managers and service permits allows the security and fault tolerance of a TNM based system to be enhanced.

The data transfer mechanisms centre on a fundamental unit of communication - the record. The presence of identity and type fields within the record enable the TNM protocols to provide a good *quality of service.*

The next chapter looks at how the principles described in this chapter have been applied in an experimental implementation of a TNM based system.

Chapter 5

Implementation

The central theme of this chapter is the experimental implementation of a protocol set based on the Triadic Network Model. The main principles of this protocol set were considered the chapter 4. In particular, the operation of a three party mechanism was described, together with the rôle of service managers and permits. The benefits to the user's *quality of service* from identity and type fields, and the reasons for the functional partitioning of the TNM protocols, were also covered.

This chapter continues the description of the TNM protocol set. However, the emphasis is more on the practicalities of the implementation than on the theory behind the design. In this respect, the development environment is discussed, but the facilities produced to support the development of the implementation are considered in the next chapter.

The details of the three TNM protocols are given later in this chapter, but first the development environment is described, because this explains some of the design decisions for the implementation. Second, attention is given to the way in

which the modularity of the model is exploited in the process and coding structure of the implementation.

5.1 Configuration

5.1.1 Computers

The main system used for the development of the experimental implementation was a **High Level Hardware** (HLH) **Orion** minicomputer. The Orion runs the Berkeley 4.2 version of Unix, [27,84], with extensions to support the novel features of the machines hardware. These features are oriented around the use of the AMD (Advanced Microprocessor Devices) 2900 bit sliced family of devices, [1].

The Orion is constructed from a processor board based on the Am2900. The processor board is programmable at microcode level, but the system and application software use a macro instruction set that is implemented using the microcode. HLH supply an instruction set [32] that is well tuned to the needs of the C programming language [55] and Unix. However, the Orion's main asset is that the user may also supply microcode for his own instruction set. This feature is of particular relevance to the development of an instruction set and compiler for EUSP [33].

The ability to change the microcode of the Orion was not of great importance to the development of the TNM protocol set. However, the intended use of the machine as an emulator for the EUSP would have provided a means of experimenting with a 'genuine' back-end processor. Furthermore, the processing requirements

for evaluation of the TNM protocols were such that having almost sole access to a minicomputer was essential.

Some efforts were made to port the implementation onto a Gould mainframe and a Sun workstation. Both systems ran variants of the 4.2 BSD Unix, which simplified the task of porting. The porting of software onto the Gould provided a means of verifying accesses to complicated structures in the code. This is because the Gould had a completely different word size and byte sex from the Orion. Furthermore, the compilers had different, and not entirely overlapping, error checking procedures. The port to the Sun workstation proved to be more difficult because of the networking software already present in the system. This is discussed further in section 7.4.6.

5.1.2 Unix

The ability to port applications readily onto a wide range of machines was only one of the merits in using Unix as the operating system for the implementation. A more important asset was the networking software that is supplied with 4.2 BSD Unix. This implementation of TCP/IP is well documented and is structured to allow network applications to be developed easily. Additionally, TCP running over IP provides a reliable transport service for higher level protocols - this is a requirement for the TNM protocol set, as explained in section 4.1.1.

The **socket** mechanism provides the interface to the network software, and, more generally, is the 'basic building block' for interprocess communication in 4.2 BSD Unix [62,63]. Sockets were referred to in the discussion of the Apollo DOMAIN in section 2.6.1. In fact the socket interface is of great significance in

many examples of communication systems - principally, those based on Unix 4.2 BSD, such as Sun NFS. The use of sockets by the TNM protocol set is considered further in section 5.2.2.

Further benefits of basing the initial implementation on Unix include :

1. The implementation of pipes as a means for data transfer between processes may be extended for use between computers on a network. The TNM protocol set incorporates a stream communication mechanism, mentioned in section 4.3, that is suitable for network pipes.

2. The structure of the networking software allows for the user to supplement or replace the lower level communication protocols. Were a Centrenet connection to be provided for the Orion, this flexibility would have permitted a more efficient implementation of the TNM protocol set using a lightweight transport service. This route was discussed in section 4.1.1.

3. The availability of a wide range of applications means that Unix is an ideal vehicle for experimenting with a means of distributing these applications over a range of machines.

Since Unix and the relevant networking software are all written in the C language [55], it was natural for the implementation of the TNM protocol set to follow suit. Furthermore, the development tools provided with Unix are all oriented towards programs written in C.

The modular nature of the implementation meant that the **make** utility [30] was invaluable. **Make** relies on the presence of *makefiles* that describe the depen-

dencies between the various object and executable modules of the target system and the source and header files of the original code. So, when a particular source file is updated, all dependent files (and *only* those files) are recompiled and linked to form the new executable modules.

The modularity of the TNM protocol set also provided some problems for development using C. Primarily, these problems were associated with differences in variable and parameter usage between different modules. The C compiler does not perform the kind of checks that would have highlighted these differences. Fortunately, the **lint** utility [49], when used sparingly, made up for these restrictions. **Lint** is solely concerned with checking for potential problems in C code that are passed by the compiler because they have valid syntax. The difficulties present in developing a modular implementation are more than offset by the advantages however, as described in the next section.

5.2 Modular Structure

In section 4.4, the partitioning of the TNM protocol set into three was discussed. The primary merit of this division is that each of the modules in the system is only required to support the protocol most appropriate to its rôle. This results in networking software that may be simpler and more efficient because of the smaller range of communication primitives that must be supported.

This section considers the modular structure possessed by the experimental implementation of the TNM protocol set. The reasons behind the various levels of subdivision are explained in relation to the Triadic Network Model.

5.2.1 Vertical Partitioning

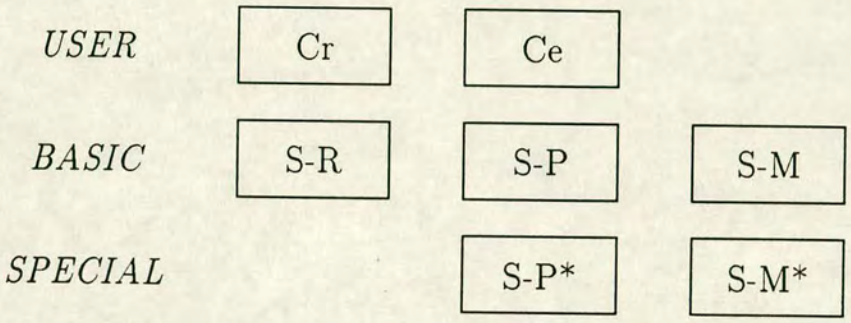

Figure 5–1: Division into Modules

The subdivision of the communications software into smaller functional units is extended beyond the simple partitioning of the TNM into three. For all of the TNM protocols, each *party* in the communication is implemented by a separate software module. Combinations of these software modules may be formed, as appropriate to the needs of the specific network devices. Figure 5–1 shows the software modules present for each of the three TNM protocols. The rôles of these modules are as follows :-

TNM *user* protocol :

- *caller (Cr)* - the node that initiates the contact.
- *callee (Ce)* - the target of the *caller's* attempt to make contact.

TNM *basic* protocol :

- *requester (S-R)* - the source of a request for a network service.

- *provider (S-P)* - the node that satisfies this request.
- *manager (S-M)* - overseer for a service request that has been rerouted.

TNM *special* protocol :

- *manager (S-M*)* - the node that takes responsibility for the provision of the service by mapping service requests onto one or more providers.
- *provider (S-P*)* - one of the nodes used by the manager to satisy the service request.

The *asymmetric* nature of the TNM protocols lends itself quite naturally to this partitioning by rôle. A result of this is considerable flexibility in the way in which the communication software is configured for each network device. Additionally, new devices may be connected to the network with a minimum of software development.

In the implementation, duplication of code development was avoided through the provision of libraries of procedures for common functions, such as the formation and separation of protocol headers. Where there was a much larger degree of commonality between different parties, conditional compilation was used. This is supported by the C language preprocessor, which includes or removes code from the compilation according to directives embedded in the code. For example,

```
# ifdef CONDITION
        /* use this section of code if CONDITION is true */
# else
        /* otherwise use this section of code */
# endif
```

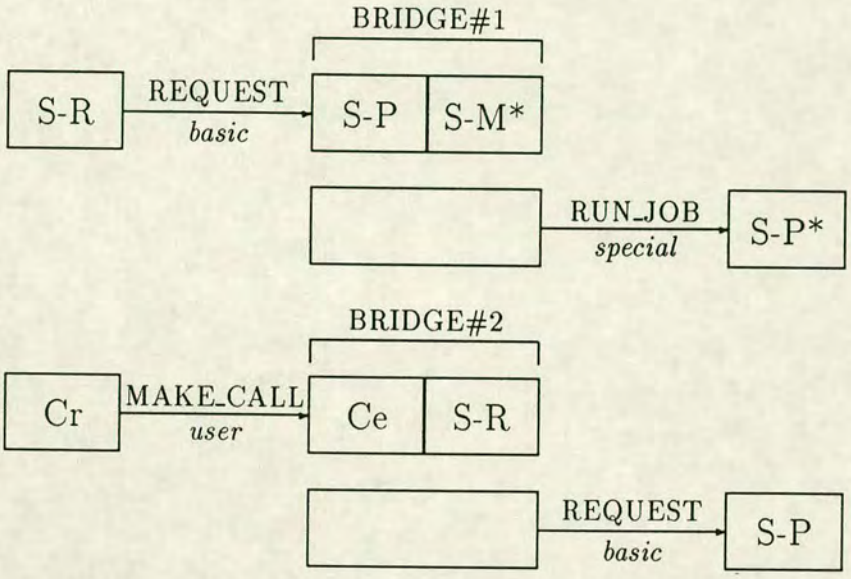

Figure 5–2: Subset Interworking

This facility is of particular advantage for a certain group of nodes that act as *bridges* between two different protocols. These bridge nodes will accept primitives from one protocol and transmit those of another. Figure 5–2 illustrates the rôle of these nodes.

In the first case, a node requires a service provided by a back-end machine. The node acts as the service requester (*S-R*) in the TNM *basic* protocol but cannot interact directly with the provider of the service (*S-P**), which uses the TNM *special* protocol. So, the *bridge* node behaves as a normal service provider (*S-P*) to the requesting node, and is the service manager (*S-M**) for the actual provider of the service.

In the second case, the bridge node allows a caller (*Cr*) to use a service provided under the TNM *basic* protocol by (*S-P*). This can be of use where, for example,

communication between two users requires some form of translation, or where security considerations necessitate encryption.

The communication barrier imposed by requiring an intermediary node in these examples merely reflects implicit differences in the usual nature of communication between nodes for the different protocols. Section 3.2.2 considered this disparity in communication requirements in greater detail.

The experimental implementation uses a bit-field to specify the TNM protocol for each primitive transferred. A mask is set by each node at initialisation to indicate the corresponding protocol. A bridge node is formed simply by setting more than one bit in this mask.

The software modules that implements the TNM protocols use lower level library procedures to perform these tasks, thereby simplifying the code. The layering of the software in the experimental implementation is described in the next section.

5.2.2 Software Structure

The structure of the software implementation of the TNM protocols is important to the portability of the code and the simplicity of the software modules. This section describes this structure and the rôle of each module. The purpose of the configuration files is also explained.

Figure 5–3 illustrates the four layers into which the software is divided. The function of each module shown here is listed in table 5–1. The dashed box in figure 5–3 represents where the library of functions for the TNM *special* protocol

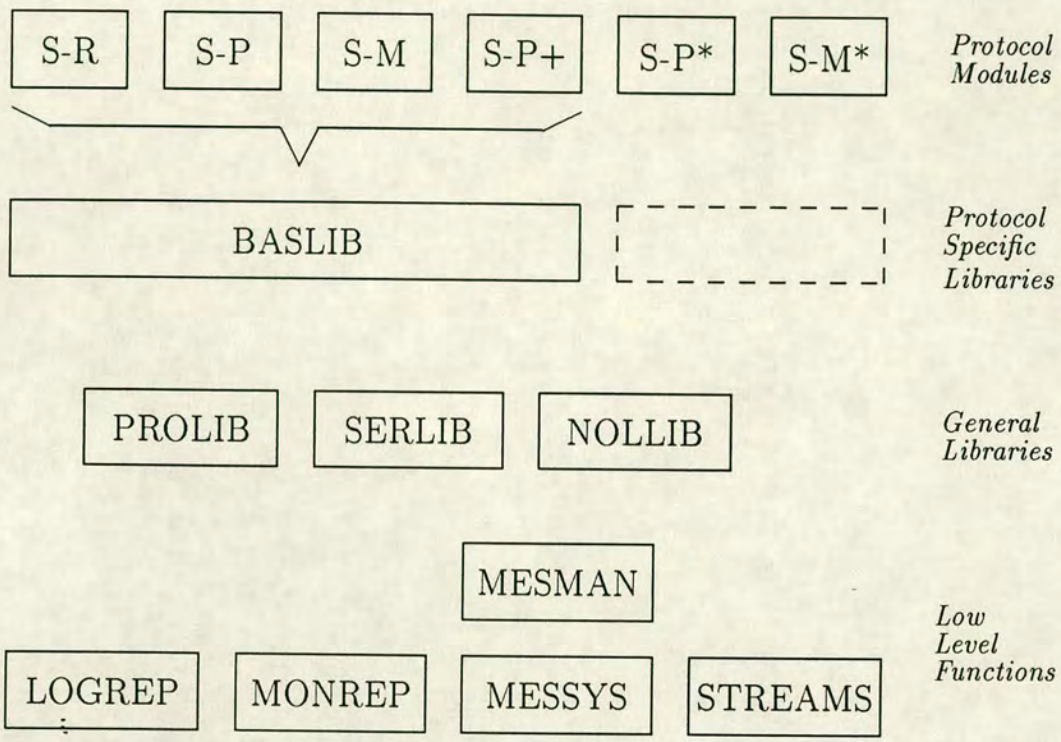

Figure 5–3: Software Structure for Implementation

modules would have been placed. However, for the experimental implementation described here, there was no need for such a library.

Layers

The layering of the software modules bears no relationship to the multiple levels of the ISO model for OSI, described in section 4.1.2, or similar communications architectures. The various layers merely represent different levels of functionality of the implementation. Whilst another implementation might structure the software in a different manner, this example has certain merits that should be exploited.

S-R	*basic* service requester
S-P	*basic* service provider
S-M	*basic* service manager
S-P+	bridge between *basic* and *special*
S-P*	*special* service provider
S-M*	*special* service manager
BASLIB	library of functions for *basic*
PROLIB	library of general functions for protocols
SERLIB	library of functions for service handling
NOLLIB	library of functions for node list handling
LOGREP	functions for making log file reports
MONREP	functions for making reports to the monitor
MESMAN	interface to communication system
MESSYS	functions for message based communication
STREAMS	functions for stream based communication

Table 5–1: Functions of Software Modules

Upper Layer : The highest layer in this structure is occupied by the modules implementing the communication parties for each protocol. Each module embodies the changes in state described by the diagrams and tables given in appendices A,B and C.

Libraries : The libraries provide sets of functions that help to simplify the code of the upper layer modules. Some of these functions are for the assembly, decomposition and analysis of protocol primitives, e.g. **baslib** has functions that are only useful for the TNM *basic* protocol. The general libraries provide procedures that are used by modules for all of the TNM protocols, e.g. **prolib** has functions for the formation of headers and the management of type and identity information.

Low Level : The lowest level of modules in this structure provide functions that have to be tailored to the specific implementation. For this implementation,

a great deal of status information is required for maintaining the behaviour and performance of the system. So, **logrep** is used for making reports to a log file for each node in the configuration, and **monrep** enables reports to be made to the **monitor** node. This node is described in section 6.1.

The remaining modules at this level, **mesman**, **messys** and **streams** are all concerned with the interprocess communications mechanisms provided by the operating system. Hence, their functions provide the interface between the TNM protocol set and the underlying communications system.

Communication System Interface

Section 4.3 described how the Triadic Network Model's fundamental unit of communication, a record, may be used for the transfer of messages, pages and streams. For the experimental implementation, the transfer of messages is supported by **messys** and streams by the functions in **streams**. For the simple applications used for testing the implementation there was no need for page transfers, so the corresponding module was not developed.

For messages, **messys** provides the functions sendm() and recvm() for the sending and receiving of messages respectively. Both functions return a count of the number of bytes transferred and take two parameters : the network address of the other node and a pointer to the record being transferred. An error value is returned if the operation cannot be completed. Other functions are provided by **messys** for the initialisation and close down of the communication system. These functions, startms() and endms(), are required for the correct use of state information by some communications systems.

This set of four functions provides the interface between the TNM protocol set implementation and the underlying communication system. The interface is common to many systems and thereby simplifies the task of porting the implementation. Furthermore, since the modules that implement the TNM protocols are written to use this interface, the implementation is independent of the underlying communications system. Hence, the dependence on a particular transport service is avoided.

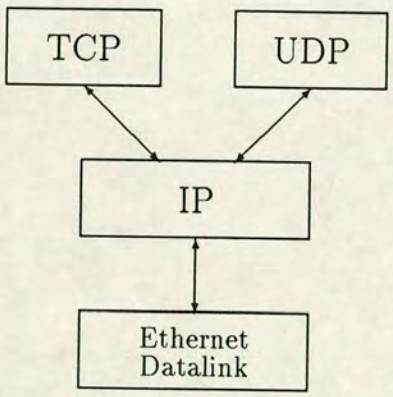

Figure 5–4: TCP/IP Protocol Stack

The **streams** module provides a similar set of functions to **messys**, but is tailored to the needs of stream based communication. For this implementation, both **messys** and **streams** used the Unix 4.2 BSD socket mechanism as the basis for interprocess communication. However, **messys** opens sockets to use UDP (User Datagram Protocol), and **streams** uses TCP (Transmission Control Protocol). Both the UDP and TCP protocols are based on the DARPA Internet Protocol (IP), as shown in figure 5–4. The UDP provides unreliable connectionless communication, whereas the TCP supports reliable ordered end-to-end communication.

The **mesman** module provides message management functions at a higher level than those of **messys**. The protocol modules use the enhanced services of **mesman** as the interface to the communications subsystem. The two primary functions of **mesman**, sendmx() and recvmx(), correspond to the **messys** functions sendms() and recvms(). Similarly, these functions have parameters for the record to be transferred and the identity of the other node. However, the node that is the destination or the source of the record is given as an integer that indexes into the node list table.

The node list table is setup initially by the **nollib** module functions, described in the next section, but is modified according to subsequent primitive exchanges. So, the protocol modules need only refer to other nodes by an index into the node list, rather than the full network addresses of the nodes.

The **mesman** recvms() function has two additional parameters : *xsource* indexes into the node list to indicate the node from which the message is expected; *time* is used to determine the time-out period to be used. There are a number of checks performed by recvmx() for every invocation :

1. If there is no response within the specified time-out interval, the procedure returns RECVMX_TO to indicate a time-out. For a **time** of 0, recvmx() will return immediately with a message, if one is waiting, or RECVMX_NO if the input queue is empty. A value of 9 for *time* causes recvmx() to block until a message is received.

2. If *xsource* is non-zero, the network address of the source of the required message is compared with the specified entry in the node list. If there is a

mis-match, RECVMX_MIS is returned. A value of 0 for *xsource* indicates that all incoming messages are to be accepted.

3. The protocol class for the incoming message is compared with that of the node's. This uses the bit-field mask, described earlier in this section, so that bridge nodes are possible. If the class match fails, RECVMX_CF is returned.

These checks illustrate the added functionality of the **mesman** functions. Two of the other **mesman** functions, initms() and finims(), control the initialisation of the node's environment configuration using the **nollib** and **serlib** modules.

Configuration Control

The general purpose libraries **nollib** and **serlib** provide functions that allow each node in the system to determine its operating environment. The **nollib** module allows for the set-up and manipulation of the list of node names and addresses for use by each node. The **serlib** module is responsible for monitoring the list of services provided by the node, and information about services provided by other nodes.

Both **nollib** and **serlib** initialise the node's environment by reading configuration files (suffixed **.nol** and **.ser** respectively). This means that an experimental system can be formed by creating a number of node processes, each with a different pair of configuration files. The behaviour of this experimental system is easily changed by making modifications to the configuration files of the nodes.

The operation of this experimental system is described in section 6.1. The manipulation of processes is explained in the next section.

5.2.3 Multiple Processes

The experimental implementation of the TNM protocol set is based on multiple processes running on a single Unix minicomputer - the HLH Orion. However, since the entire implementation is based on the 4.2 BSD socket mechanism, the transfer of TNM protocol primitives is representative of what would occur if the processes were resident on different machines. The main benefit of basing the initial implementation on a single processor is that it was possible to achieve a more complete monitoring of the behaviour of the system. The monitor process is described in section 6.1, but the use of processes is considered here.

Configuration

The experimental system is initiated by the **sim** module. This reads a configuration file, config.nol, that lists the name and type of each TNM protocol dæmon to be created. For each entry in the configuration file, a dæmon is created by spawning a process to run an executable module for the specified TNM protocol node. Once all of the required processes have been created, the **sim** module terminates, but leaves its child processes running as background processes. Each of these processes represents a node in the system. So, for example, three NSM nodes are created by spawning processes running the TNM *basic S-P* module with three different identities: *server*, *d_serv* and *co-ord*.

Each process assumes the appropriate operating environment by reading its configuration files using functions in the **nollib** and **serlib** modules, as described earlier. So, for *server*, the configuration files are server.nol and server.ser. These files list, respectively, the nodes known to the node *server* and the services that

it provides or otherwise 'knows' about[1]. Once initialised, each process moves into an IDLE state in accordance with the state definition for the protocol, and awaits an incoming message.

For an NSM running the *S-P* module, the next change in state occurs when a request is received from an *S-R*. An *S-R* process is created when the operating system determines that a request for a network service has been received. The provision of a simple shell to achieve this is described in section 6.3.

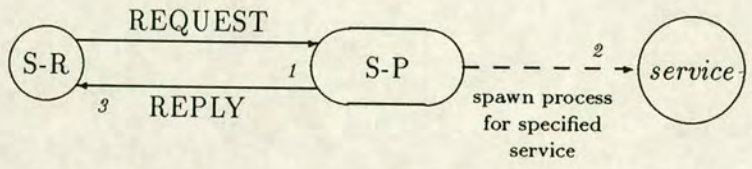

Figure 5–5: Service Provision

For the example illustrated in figure 5–5, the *S-R* sends a REQUEST to the *S-P* that can provide the required service. So, the *S-P* spawns a process to execute the appropriate software to implement the service, and finally returns the result to *S-R* in a REPLY. The process implementing the service *dies* on completion.

The second example, figure 5–6, shows the processes required when the *S-P* receiving the initial request cannot supply the required service. The exchange of primitives complies with the three-party mechanism, described in section 4.2, that forms the basis for the TNM *basic* protocol. $S\text{-}P_0$ spawns a separate process

[1]The .ser file also lists the services provided by nodes to which the node can redirect requests.

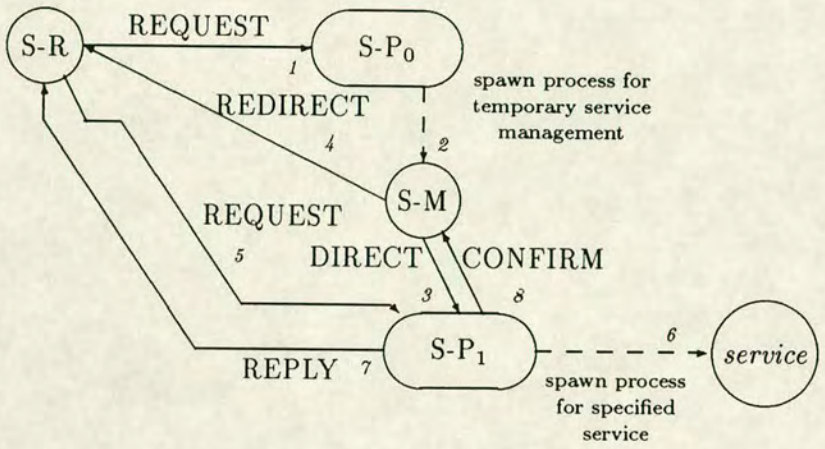

Figure 5–6: Service Redirection

to provide the temporary rôle of *S-M* for this service. The *S-M* then handles all subsequent exchanges of protocol primitives for this service - freeing $S\text{-}P_0$ for other requests.

Once CONFIRM is received by *S-M*, the process terminates normally. $S\text{-}P_0$ receives a signal when a child process is terminated, and updates an internal table that records the status of all its redirected requests. The creation and termination of processes may be clearly observed using the monitor, described in section 6.1.

The two examples given here are appropriate for simple requests and replies. However, where much larger amounts of data have to be transferred, the 2 nodes would have opened a stream communication channel and used the functions provided by the **stream** module. A further two sockets must be created for a reliable

end-to-end transport service[2]. No additional processes are required to monitor the stream based communication because the stream module functions will detect any escape sequences present in the data transfer.

[2]When sockets are created, the type of service required must be specified. The resulting sockets will then offer that particular level of functionality for the duration of their lifetime.

5.3 Protocol Set Details

This section takes a look at the characteristics of the TNM protocols in the experimental implementation and describes their main primitives. The more specialized primitives for the notification of error conditions and attempts at recovery are discussed later in this section. First, though, the introduction in section 4.3.2 to the TNM fundamental unit of communication is expanded here to describe how the implementation uses each field of the TNM **record** structure.

5.3.1 Basic Structure

In section 4.3.2, it was stated that every transfer of information in the networked computing environment is formed from a record. The basic form of the record was described as comprising a **type**, and **identity** and contents. The implementation of the TNM record follows this basic form, and may be broken down into the elements illustrated in figure 5–7.

Datestamp : This field is present to ensure that every record is unique. The use of 8 bytes would be excessive in a real application, but in the experimental implementation it allows for extra monitoring information.

Context : This provides sequence numbering in a format that corresponds to the operation of the three-party mechanism.

Permit : All records must have an associated permit, although it is possible for the permit to be the *standard permit* for initial requests to the network.

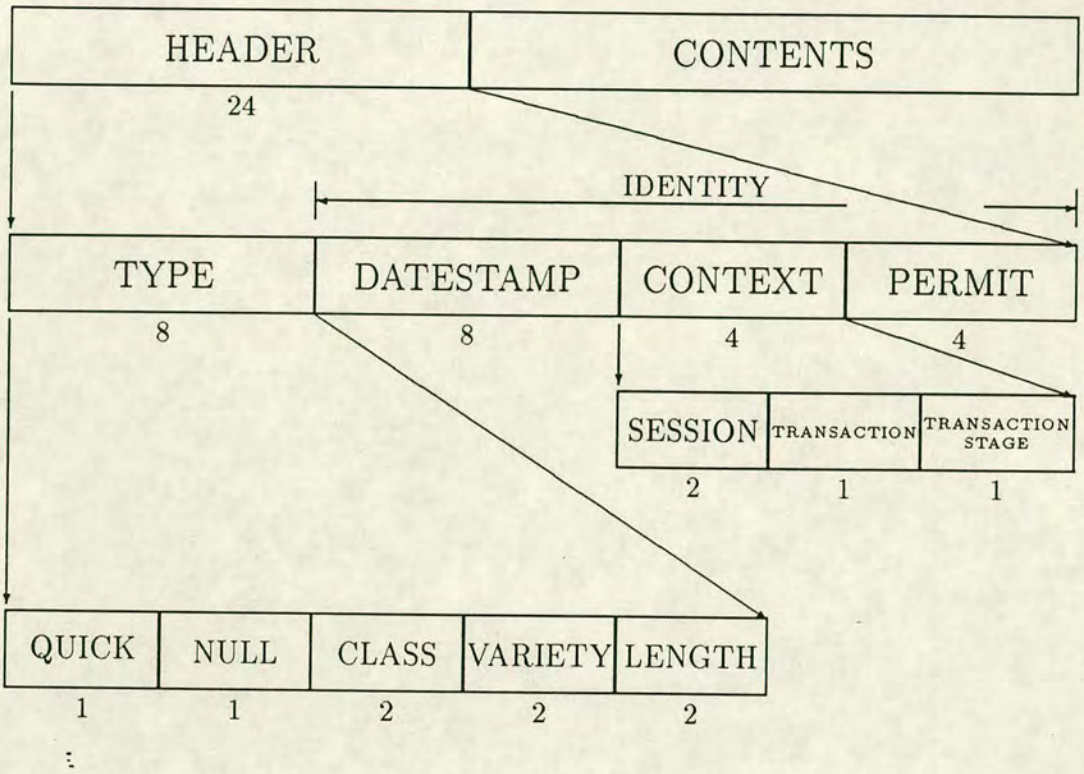

Figure 5–7: Breakdown of Record Structure

Context

The various stages of an interaction were described in section 4.3.3 and illustrated in figure 4–7. To recap, the smallest stage is the **transfer** of a record, a group of record transfers between two nodes is a **transaction** and a **session** is a collection of transactions that effectively implement the service. So, for every new service request, the originating node provides a new **session** number and resets the other two context fields to zero. Each transfer of a record has an incremented value in the **transaction stage** field, and whenever another node has to be used in the service provision the **transaction** field is incremented.

The node that initiates a new session is responsible for the selection of the new

session number in the context field. So, the requester of a network service will first choose a new session number, s_0, and then issue the REQUEST with a context s_0:0.0. If the recipient of the request must redirect it to another node, it will issue DIRECT with a new session number, s_1, and context s_1:0.0. This is because the second transaction is independent of the first, and the new initiating node has no control over the original context sequence.

The sequence numbering resulting from the use of the context fields is illustrated further in section 6.2, where it explained how diagrams are produced automatically by the analysis utilities **pp1** and **pp2**.

Type

The type fields are used in different ways by the software modules within the implementation. The **quick** type is a bit-field that indicates the nature of the communication mechanism - a page, part of a stream, a data object or a protocol primitive. The value of this field determines which low level functions should be used to handle the record in the experimental implementation. The **quick** type field is the first octet in the TNM record so that dedicated hardware, such as a BEM I/O-processor, could select an appropriate channel for the type of communication which follows. For example,

protocol primitives could be handled directly by the I/O-processor, without any involvement of the main processor.

pages should have the data copied directly into a page-aligned buffer in main memory, and the fixed size header inspected by the I/O-processor for the page identifier.

streams may be piped into main memory through a filter that monitors the synchronisation and error correction sequences.

The size of the TNM record header is such that there would be enough time to select an appropriate DMA channel for the remainder of the record. The DMA channels would have to be initialised before the receipt of the start of the record.

The **class** type field indicates which of the TNM protocols is being used. As described in section 5.2.1, this is a bit-field so that it may be matched with a mask at the receiving node. The node will only accept records of the correct class. The **variety** field is the primitive from the TNM protocol indicated by the **class** field. The **length** is the length of the whole record and the **null** field pads the type fields to a multiple of 32 bits.

Data

Where the record is not used to carry a protocol primitive, the remainder of the header may be used to hold system level information about the data which follows. For example, if the record holds a page, the header should indicate the type of data in the page and the corresponding system level identifier. For paged data, it is important that the header is of a known size so that the DMA channels will copy only the data into the main memory buffers.

The transfer of data should be handled as much as possible by the lower level protocols without any involvement of the higher level TNM protocols. This is the reason for dropping most of the TNM record header information for the passage of data. For the most part, it is the responsibility of the underlying transport service to ensure the reliable communication of the data.

5.3.2 Characteristics of Protocols

The rôle of the TNM protocols was discussed in section 3.2.2, and examples of interactions using the primitives have already been used. A concise look at the more important primitives of each protocol is made here. This is supplemented by a more detailed examination of the TNM protocols in appendices A, B and C.

User Protocol

The TNM *user* protocol is essentially a virtual circuit management protocol, with the familiar three phases of operation :

1. Negotiation

2. Information exchange

3. Conclusion.

For efficiency, the second phase is based on the underlying communication system. Lower level protocols undertake the transfer of data between the parties of the interaction with the smallest possible overheads. From section 4.3, it is

clear that the transfer of data may take a variety of forms, but the lower level protocols treat all data transfers in the same manner.

The important primitives for the *user* protocol are associated with the first and last phases of operation. The protocol governs the establishment and termination of the interaction and, as such, must pass on details of the required form of the communication. The initiator of the session, the **caller** (*Cr*), issues MAKE_CALL, supplying details of the service required. Figure 5–2 illustrated how the recipient, the **callee** (*Ce*), might request further services to support the desired form of the interaction. If the *Ce* is both willing and able to take part, it responds with ANSWER_CALL, and returns the neccessary information about the established connection. Normally, both parties then move on to the second phase, in which the *user* protocol plays little part. When the exchange of information is complete, one of the parties will issue END_CALL and the other will respond with CALL_COMPLETE.

The *user* protocol incorporates further primitives for some infrequently needed actions :

- the communication can be suspended for brief intervals, and subseqently resumed.

- the caller can involve an intermediary party in the communication, e.g. for translation.

- the networked system, to which the caller and callee are attached, can enforce a form of call charge.

- the system may break into an established conversation to bring it to a premature end. This may be because the communication is too expensive or because of resource contentions, for example.

Special Protocol

The TNM *special* protocol operates in conjunction with the *basic* protocol to allow sophisticated devices to be supported in the networked system with minimal communications overheads. This is achieved through a *master-slave* hierarchy. The *master*, the service manager (*S-M**), carries the heavier burden of the communications requirements by acting as the provider of the service, whilst using any number of *slaves*, service providers (*S-P**), that concentrate on the real computational needs of the service.

When the *S-M** receives a request for a sophisticated service, it first determines how the request may be satisfied using its associated back-end modules (BEMs). A RUN_JOB primitive is then issued to a suitable BEM, specifying the computation required. The level of detail supplied in this directive will depend on the nature of the original service request and the capabilities of the BEM. A trivial instance could merely require the BEM to execute a module from a locally held library and return the result to the *S-M**. More complicated service requests, or a less sophisticated BEM, may necessitate the transfer of object code from the *S-M** and the establishment of communication channels for the return of the results.

On concluding the specified computation, the *S-P** returns JOB_COMPLETE to the *S-M** with any appropriate accounting or status information. The *S-M** may require an indication of the state of the *S-P** during the course of the execution of

the job. The *S-M** could explicitly issue STATUS_REQUEST on such occasions, or specify the need for these updates in the original RUN_JOB. The *S-P** returns status information during the computation with STATUS_REPORT.

The TNM *special* protocol has further primitives to allow the premature termination of the execution of the job and for the establishment of the bond between BEMs and their managers.

Basic Protocol

The TNM *basic* protocol is derived from the three-party mechanism, described in section 4.2. The main *basic* protocol primitives have been used in numerous examples already, but there are further primitives that provide error analysis and recovery capabilities. Since the *basic* protocol is of primary importance in the TNM system, some consideration of these additional features is warranted.

5.3.3 Error Analysis and Recovery

For the greater part, all transactions using the TNM *basic* protocol will use the REQUEST/REPLY primitive exchange. The flexibility afforded by the three-party mechanism requires DIRECT/REDIRECT/CONFIRM exchanges, with the possible need for REDIR_M. So long as the correct sequences are maintained, and the transaction times are not exceeded, there is no need for additional *basic*

primitives. When, however, timeouts occur, or REJECT_VALIDITY is returned[3], additional primitives are used to :

- enquire about the nature of the problem or state of the service provision
- notify affected parties of a failure in the service
- try to ensure that complete service failure does not occur
- recover from a partial service failure
- terminate prematurely the provision of the service.

A number of different timeout values are used according to the communication party and its state. This is supported by the **time** parameter of recvmx(), described in section 5.2.2. Where the *S-R* times-out whilst waiting for the REPLY from *S-P*, it initially tries to resolve the problem with the *S-P* by sending ENQ_SUPPLY. If the *S-R* fails to receive a satisfactory response from the *S-P*, it will escalate awareness of the problem and issue ENQ_SERVICE to the *S-M*. The *S-M* may then send ENQ_PROVISION to *S-P* in an attempt to continue with the current provider of the service. Alternatively, *S-M* may opt to REDIRECT the *S-R* to use a new *S-P*.

Where the *S-P* is only part-way through the provision of the service, it should return EXTEND_TIME to an enquiry from the *S-R*. The *S-R* may then adopt a

---

[3]REJECT_VALIDITY is returned when a node receives a primitive that is invalid for the current state, has an invalid permit or the wrong context (session, transaction or stage values).

new timeout value for recvmx(). Where *S-P* receives an enquiry from the *S-M*, it should return PROV_ACTIVE, giving details of the state of progress.

Where possible, the parties involved in the communication will attempt to maintain the service provision until it is complete. However, when any node determines that service failure has occurred, it should notify the other affected parties. *S-P* may return SUPPLY_FAIL to an enquiry from *S-R*, or issue PROVISION_FAIL to *S-M*, depending on the circumstances. *S-R* will issue SERVICE_FAIL to *S-M* in the hope that *S-M* can REDIRECT it to a new *S-P*. Where there is a total failure in the provision of the service, *S-M* will return NOTIFY_S_FAIL to *S-R*, which must then abandon the network service request.

The *S-R* can bring the service provision to an end by issuing ABORT_REQUEST to the *S-P*. If the *S-P* has an associated *S-M*, it will return CONFIRM with an error code, indicating that *S-R* terminated the service before normal completion. The *S-M* can also terminate the service by issuing ABORT_PROVISION to the *S-P*. In this case, the *S-P* need not reply.

The error primitives of the TNM *basic* protocol help to make the support for network service provision more robust. The intricacies of the exchange of primitives to prevent service failure may be more apparent from the detail given in appendix B.

The example given in the next section assumes that no difficulties are encountered in the provision of the service, because this would unduly complicate the discussion.

5.4 Example

To illustrate the application of the TNM protocols, an example of the provision of a complex service is given. The example is considered to be representative of some of the more demanding requests that may be made to the system.

> *A user wishes to compile and then execute a program which will the require use of two special purpose processors. The program is held locally by the user's personal workstation and should be compiled using a remote compiler engine. The object code produced by the compiler is not required after this request and so should only temporarily be stored using a network fileserver. The object code will comprise a number of object modules and each module should be directed to the appropriate processor.*
>
> *The initial data is resident on a network fileserver, and this should be piped to the first processor. The intermediary results should be streamed to the second processor and the final results returned to the user's personal workstation, using another pipeline, for further analysis.*

This example is composed of two distinct requests :-

1. Compile a locally stored program using a remote compiler engine.

2. Execute the object code on two special purpose processors.

Since the user does not require the intermediary data produced by the first processor or a copy of the object code processed by the high performance processors, the two requests may be parcelled together and submitted as a single service request to a service manager, which will then allocate the different tasks to network devices as appropriate.

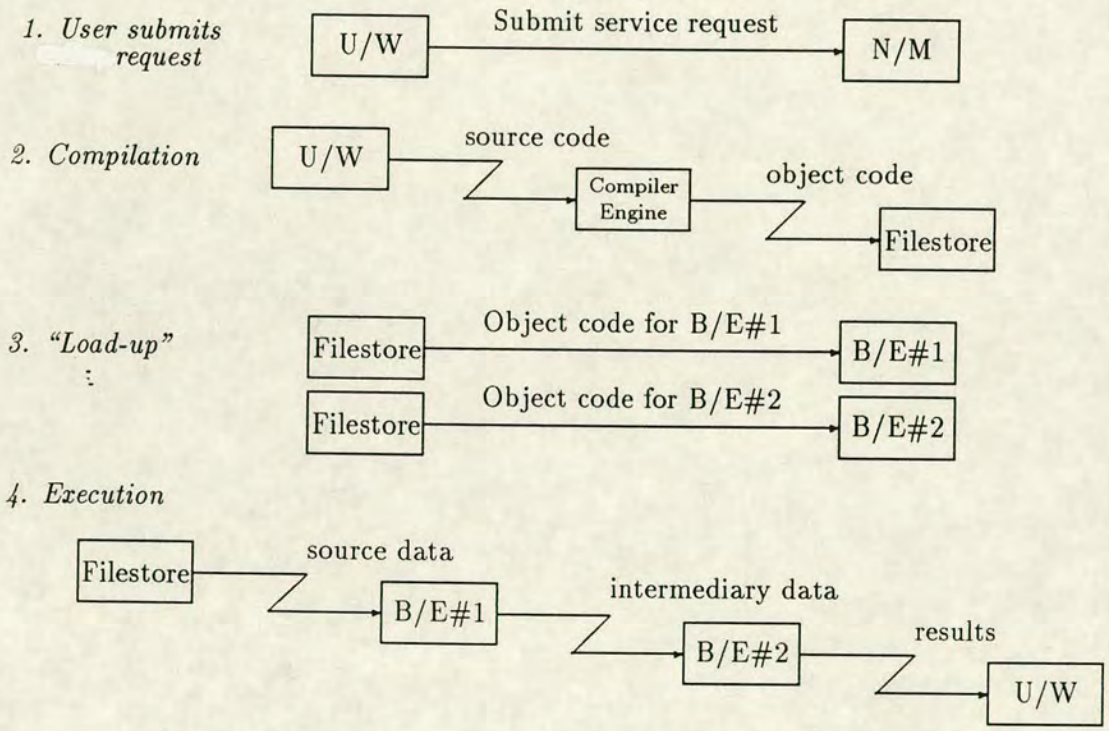

Figure 5–8: Phases of Service Provision for Example

Conceptually, the implementation of this service would comprise four phases, as shown in figure 5–8. B/E#1 and B/E#2 represent the two special purpose processors, U/W is the user's personal workstation and N/M is the network service manager.

The following points should be noted :-

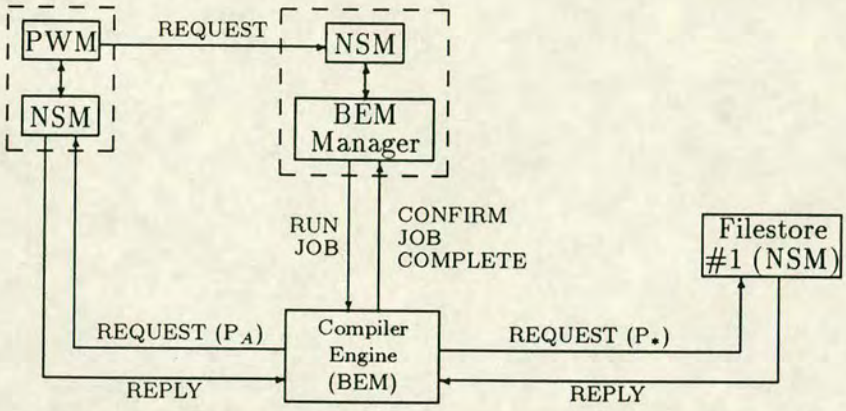

Figure 5–9: Implementation of First Service Request of Example

1. In phase 2, the source code is shown as being piped from the user's workstation to the compiler engine, and the object code as piped to the fileserver. However, if the compilation consists of a number of passes then, in essence, the files are merely transferred and not piped. Conceptually, though, the compiler engine may be viewed as a *filter*.

2. In phase 3, the object code modules are simply transferred to their respective B/Es. They cannot be truly piped because execution cannot begin until the processors have loaded all of their code.

Figure 5–9 shows a diagramatic representation of how the first service request of the example is implemented. The logical sequence of transfers, figure 5–10, enables a clearer understanding of the use of the protocols.

The mnemonics used within the boxes, which represent nodes, are as indicated in the accompanying textual description. The direction of transfer of protocol primitives between pairs of network nodes is shown by the arrows. The numbers given with the description of each transfer correspond to the transaction identifier.

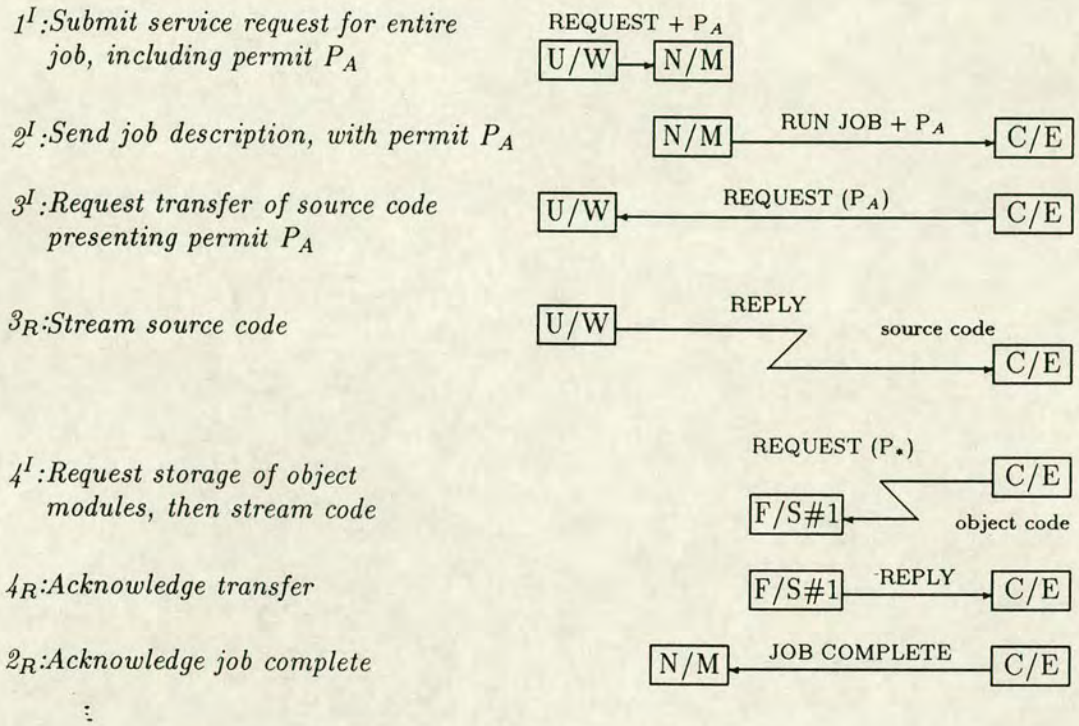

Figure 5–10: Transfers during First Service Request

So, for example, 1^I refers to the 'issue' transfer of the first transaction and 3_R is the 'return' transfer of the third transaction. The transactions involved in the implementation of the service are given in table 5–2.

The user's personal workstation (U/W) initiates the session when the first service request is issued to a network service manager (N/M), which may be the Coordinator. The service manager determines from this request that a number of nodes will be involved in the provision of this service. The network service manager issues a job description, detailing the nature of the compilation required, to the compiler engine (C/E). The service permit P_A, originally provided by the user PWM, is included in the job description so that the compiler engine may gain access to the source code file. The compiler engine issues a request for the

source code to the user workstation NSM, presenting the service permit P_A. The workstation NSM responds by streaming the file back to the compiler engine.

Prior to producing compiled code, the compiler engine issues a service request to a network filestore (F/S#1). The service request, for permission to transfer the object code modules, is sent with a 'standard' service permit, allocated on some previous occasion. The object code is streamed to the filestore as it is produced and, on completion of the transfer, the filestore returns an acknowledgement to the compiler engine. Finally, the compiler engine returns an acknowledgement to the service manager to indicate the completion of the job.

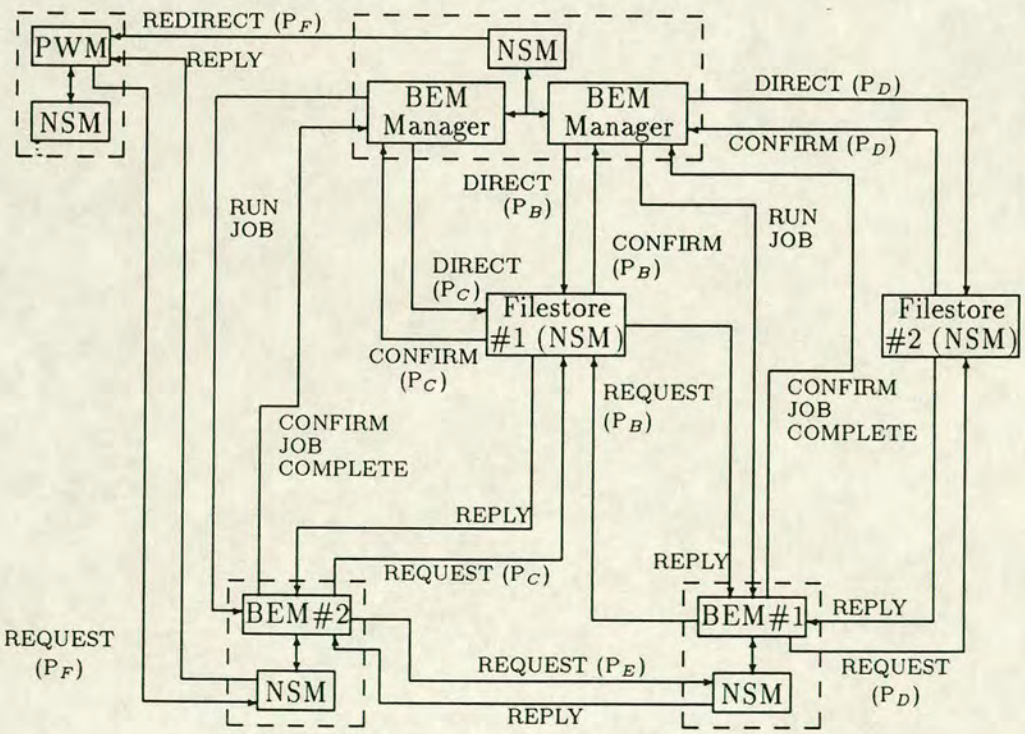

Figure 5–11: Implementation of Second Service Request of Example

Figure 5–11 illustrates the implementation of the second service request of the example. As is obvious from this diagram, this implementation is fairly intricate.

To simplify the explanation, the transfers of primitives are considered in three stages, *initiation*, *execution* and *conclusion*.

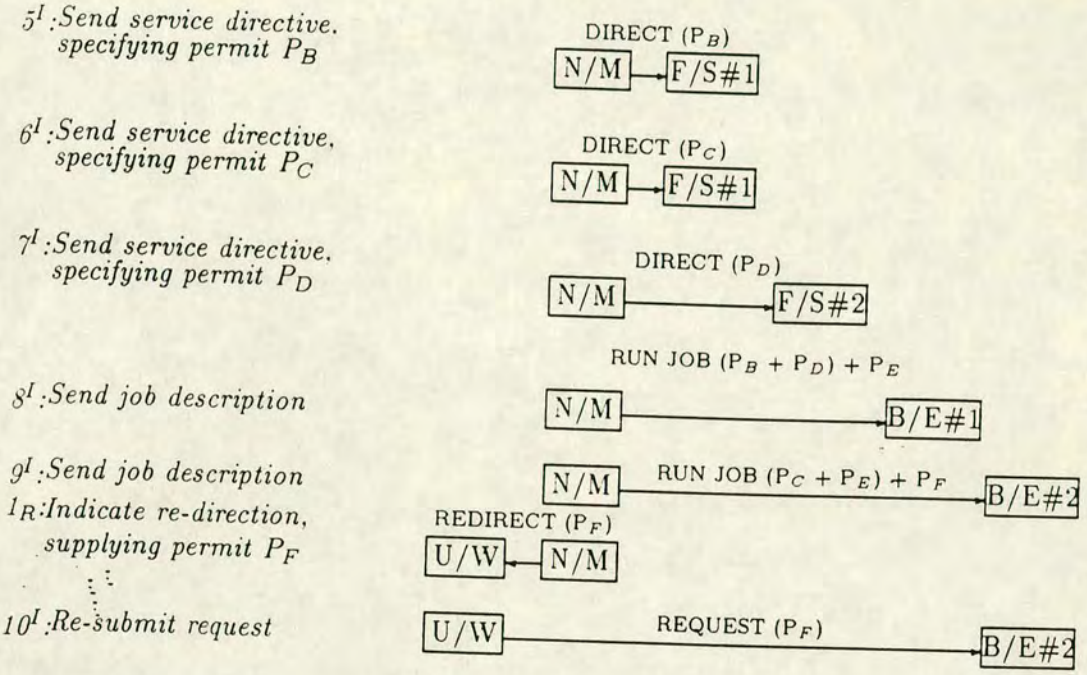

Figure 5–12: Initiation of Provision of Second Service

The importance of the network service manager (N/M) is clear from its rôle in the initiation of the provision of the second service, figure 5–12. The network service manager sends directives to both the object code and source data filestores, indicating to them that they are to satisfy service requests in the future. The filestores will only provide a service to a node presenting a service permit matching one of those issued. The filestore with the object code (F/S#1) receives two such directives, and correspondingly two permits (P_B and P_C). The reasons for this will become apparent during the next stage.

The network service manager then issues job descriptions to both of the special purpose processors (B/E#1 and B/E#2). These descriptions contain the

information needed by the processors to initiate the execution of their respective processes. Included with the job descriptions are the service permits that will be needed for obtaining services from other network devices and also an indication of a service permit that may be presented to that processor. So, for B/E#1 the permits P_B and P_D enable it to retrieve the object code and source data, respectively. The permit P_E will be used by B/E#2 to gain access to the data produced by processor B/E#1.

Finally, the network service manager replies to the original service request issued by the user's workstation (U/W). The reply indicates that the service has been re-directed and includes a permit (P_F) that may be used in obtaining the results of the computation. The workstation then re-submits the request to the second high performance processor (B/E#2), together with the service permit, and awaits the results of the operation.

It is during the "Load-up" and execution stage of the service provision that the service permits are presented, figure 5–13.

Both special purpose processors issue requests for the transfer of the object code modules to the network filestore used by the compiler engine (F/S#1). Each request includes the appropriate service permit, verified by the filestore before it returns the corresponding data. The need for two directives from the service manager is now evident.

Once the object code has been loaded, the processors commence execution. Before any computation may be performed, the processors request that the input data be transferred using a pipe. This involves B/E#1 issuing a request to the source data filestore (F/S#2), together with permit P_D, and B/E#2 sending a

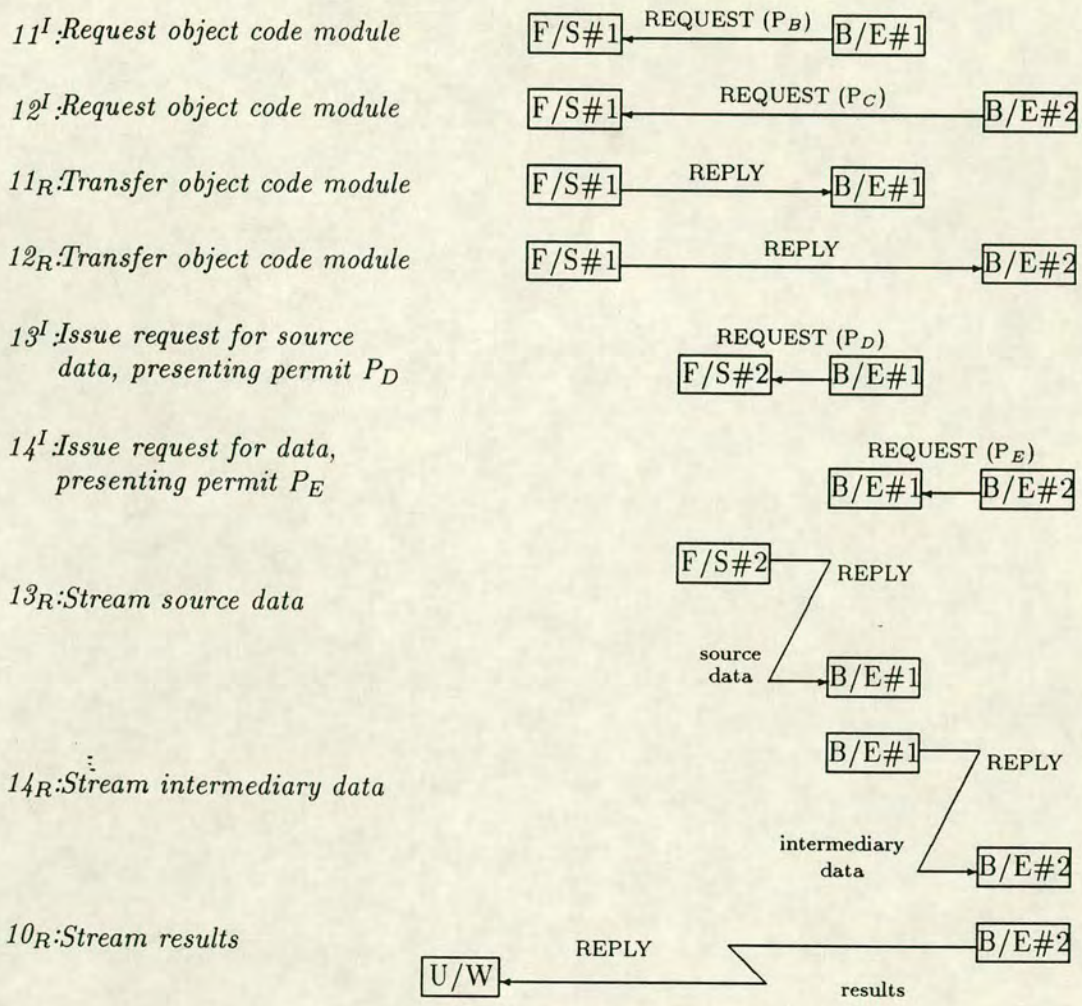

Figure 5–13: "Load-up" and Execution of Object Code

request to the B/E#1 NSM module, with permit P_E. Both of these permits were issued by the network service manager in the previous stage.

The source data is then streamed to the first processor from the network file-store. As the first processor produces data, it sends it to the second processor, which in turn returns the results of its computation to the user's workstation via another pipe.

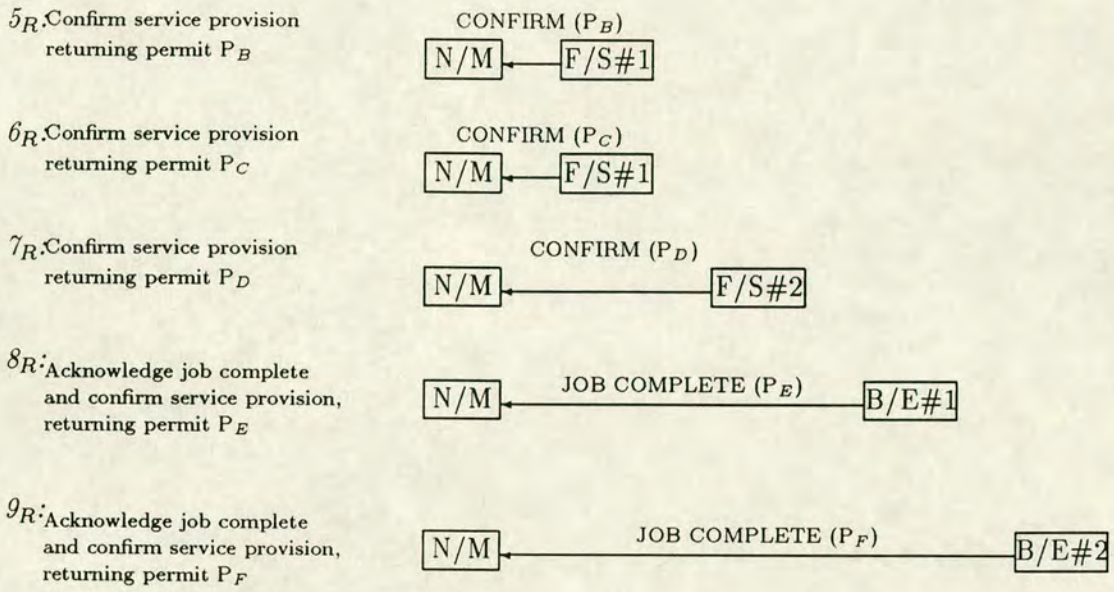

Figure 5–14: Conclusion of Service Provision

The concluding stage of the service provision, figure 5–14, proceeds without any involvement of the user's workstation. Indeed, the workstation is unaware of the transfers involved. Each filestore returns a confirmation to the network service manager to indicate the successful provision of the re-directed service. With each confirmation is included the associated service permit. Since the object code filestore (F/S#1) was originally issued with two permits, it must return two confirmations and permits.

Finally, the two special purpose processors each return an acknowledgement of the completion of their jobs to the service manager. With each acknowledgement is an accompanying service permit. These permits were the ones issued to allow other devices to gain access to the output from the processors.

The entire operation requires 14 transactions, each with an *issue* and *return* pair of transfers, as itemized in table 5–2. This is only one possible implementa-

T. No.	Issue	Return
1	Submit service request	Return re-direction for results
2	Job description for compile	Acknowledge job completion
3	Request source program	Stream source program
4	Request storage of object code	Acknowledge completion of transfer
5	Service directive	Confirm service provision
6	Service directive	Confirm service provision
7	Service directive	Confirm service provision
8	Job description	Acknowledge job completion
9	Job description	Acknowledge job completion
10	Request results	Stream results
11	Request object code	Transfer object code module
12	Request object code	Transfer object code module
13	Request source data	Stream source data
14	Request intermediary data	Stream intermediary data

Table 5–2: Transactions made during Example

tion of the example and the sequence of transfers shown in the diagrams is only approximate, since the exact order is non-deterministic.

5.5 Summary

This chapter has looked at some of the issues raised by the experimental implementation of the TNM protocol set. The benefits of a modular software structure have been illustrated by a consideration of the rôles of the modules that formed this implementation. The fundamental software structures and primitives of the TNM protocols have been explained, and the example showed how the TNM protocols combine to provide a sophisticated network service.

The following chapter looks at the software tools that had to be developed to allow the work described in this chapter to proceed. A common example is used throughout the chapter to highlight the purpose of each software utility.

Chapter 6

Support for Development

The evolution of the experimental implementation of the TNM protocol set, as described in the previous chapter, was only possible because of the presence of additional software facilities. This chapter covers the rôle and purpose of these tools and utilities.

The most important tool is the **monitor**, used for observing the behaviour of the various processes that are created as part of the TNM protocol set implementation. The monitor also produces a log file, recording the creation and termination of processes, as well as certain key details about protocol primitive transfers between them. This log file is then processed by 2 utilities, **pp1** and **pp2**, to produce a diagrammatic representation of the sequence of transfers.

A simple command shell was developed to allow some experimentation with the user's interface to the TNM protocols. The construction of an **Environment Management Utility** (EMU), in parallel with the work on the simple shell, allowed some investigation of a more elaborate user interface. The simple shell was useful in driving the TNM protocol set implementation to generate information

about its behaviour, and EMU aided movement between the various development activities.

6.1 Monitor

The experimental implementation is based on a single machine - the HLH Orion minicomputer, running 4.2 BSD Unix. This has meant that it has been possible to obtain a much greater amount of information about the behaviour of the implementation than would have been practical with a multiple computer installation. The primary means of extracting this information has been through the use of a special process that both controls and monitors the operation of the experimental system. This section concentrates on the key features of this special process and how it functions. The relative position of the monitor is illustrated in figure 6–1.

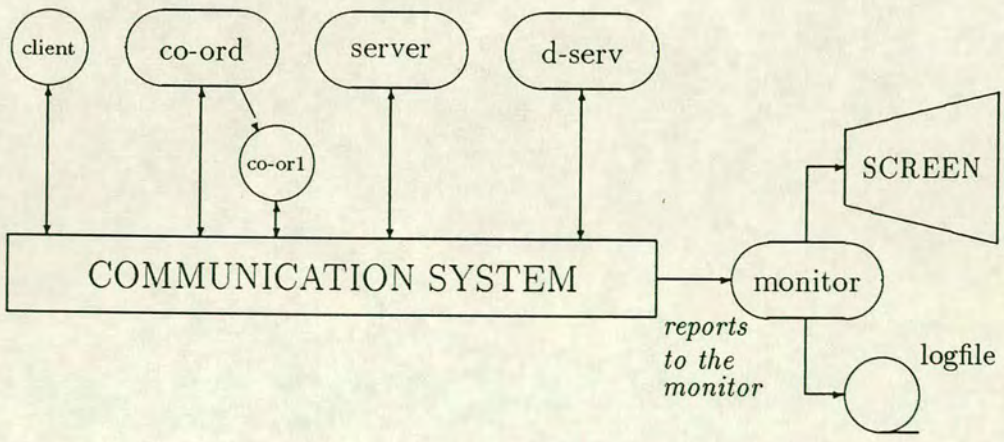

Figure 6–1: Position of the Monitor

6.1.1 Features

The **monitor** is an independent utility program that is used to

1. initialise the experimental system.

2. generate requests for network services, under the control of the user, to stimulate interactions using the TNM protocols.

3. monitor the exchange of TNM protocol primitives, the creation and subsequent termination of processes and the operational states of the nodes in the system.

4. record details of the behaviour of the system to facilitate later analysis by other tools.

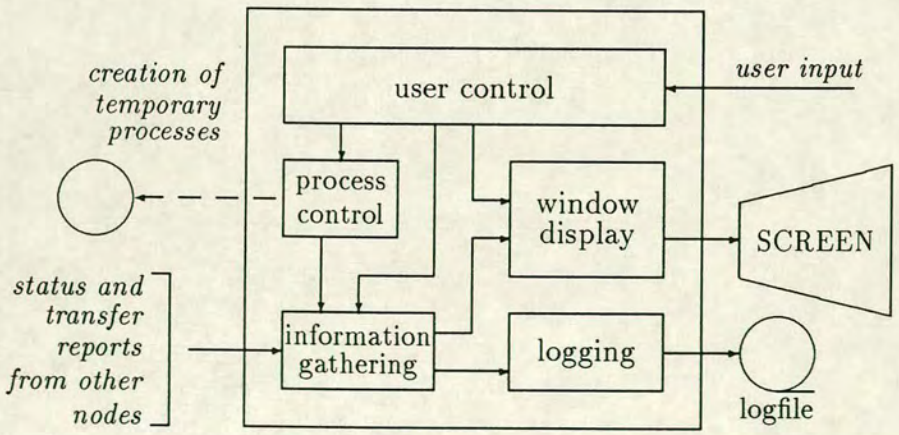

Figure 6–2: Structure of the Monitor

The monitor's rich functionality is provided by the simple structure of figure 6–2. The user maintains control of the monitor through a menu-driven windowing

interface, which is described further in section 6.1.4. The monitor presents the current set of options to the user in a menu in the lower right-hand portion of the screen. The user selects an option by entering its key number, and the monitor control interprets this and sends an appropriate directive to the corresponding functional unit.

The creation and premature termination of processes is handled by the *process control*. This uses temporary processes to either initialise the system or issue a new service request. Process operations are reported to the *information gathering* sub-unit, which is the focus for all event reporting. The other nodes in the system send messages to a *well-known* socket that is created by the monitor at its inception. These messages are decoded and then passed onto the *display* and *logging* functions. The user has some control over what is displayed.

6.1.2 Control

System Initiation

The configuration of the system is defined by a file, **config.nol**, that lists the name and type of every node. When the user elects to initialise the system, by selecting the appropriate option on a monitor menu, the monitor uses the config.nol file to determine how to proceed. First of all, the monitor spawns a separate task to create the dæmon processes that will represent each of the nodes in the system. In the example in figure 6–3, three NSM nodes are created, with identities *co-ord*, *server* and *d-serv* (a server dedicated to a single service). These nodes will be used in further examples in the remainder of this chapter.

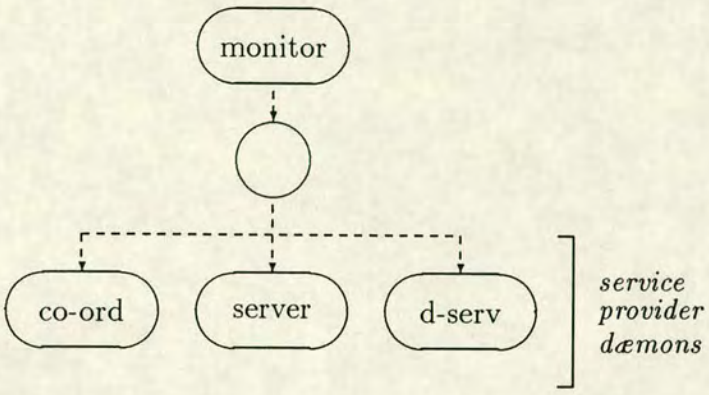

Figure 6–3: System Initiation

The process spawned by the monitor only exists until all of the nodes have been created, whereupon it terminates. During its lifetime, the process scans through the config.nol file, reading each entry in turn. The entries in the file give the names of the nodes and their types, e.g. TNM *basic S-P* module. The temporary process creates a new process running the corresponding program. The resulting process uses the name listed in its entry in the config.nol file when communicating with the monitor, and as the name of its own configuration and log files.

Service Request

As with the initialisation of the system, the monitor creates a temporary process when the user opts to issue a new service request. This new process then adopts the same procedure as would a command shell in issuing a request for a network service : a process is spawned to run the TNM *basic S-R* module, with the details of the required service passed as a parameter list. Also, channels are established

for the input and output of streams, where required. Figure 6–4 illustrates the way in which the monitor makes a simple service request.

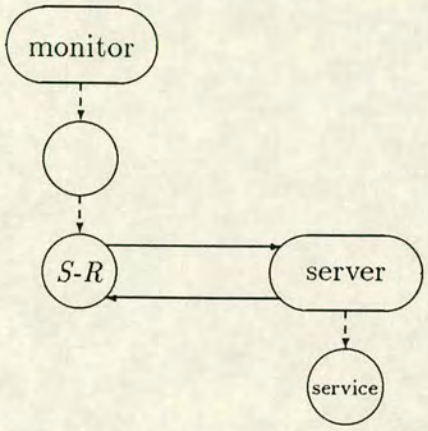

Figure 6–4: Simple Service Request

User Observation of Protocol Primitive Transfers

In its normal mode of operation, the monitor displays all of the node status and record transfer reports on the screen, in the manner described in section 6.1.4. However, it can be difficult to focus on the area of interest because of the extra information on the screen, and the rate at which messages are written to the screen. For this reason, the monitor provides some additional display controls for the user.

First, it is possible to *turn off* the display of messages on the screen. This is of greatest use when the system is being initialised and the user does not wish to see the status reports for the service provider nodes. The monitor does not provide a more sophisticated message filtering scheme since it was felt that this

was unnecessary - the simple on/off toggling is sufficient. Whilst the displaying of messages is under the user's control, all incoming status and transfer reports are logged to a file, regardless of the display mode.

For detailed monitoring of the effect of a sequence of protocol primitives, the monitor supports a *single-step* mode of operation. When the user has opted to run the monitor in this way, record transfers are only made when the user presses a key. This enables the user to make closer observations of the response to a particular protocol primitive. So, a rogue response may be introduced, at the user's request, and the implementation's error handling capabilities can be monitored step by step.

The user may select to run in *single-step* or at *full speed* at any time. The remaining monitor operations proceed unaffected by the current mode. The single-step capability is only possible because of the way in which the monitoring of events is achieved, as described in the next section.

6.1.3 Monitoring

The monitor receives encoded messages from other nodes in the experimental system whenever one of the following events occurs :-

1. the node changes state

2. a new node is created

3. an existing node terminates

4. a node issues a TNM protocol primitve

5. a TNM protocol primitive is received.

The first 3 events are designated as *status reports*, the last 2 as *transfer reports*. All of the messages include the name associated with the sending station and follow the syntax rules of the monitor. Hence, the messages are short and efficient; they are formed quickly by the nodes, and readily interpreted by the monitor.

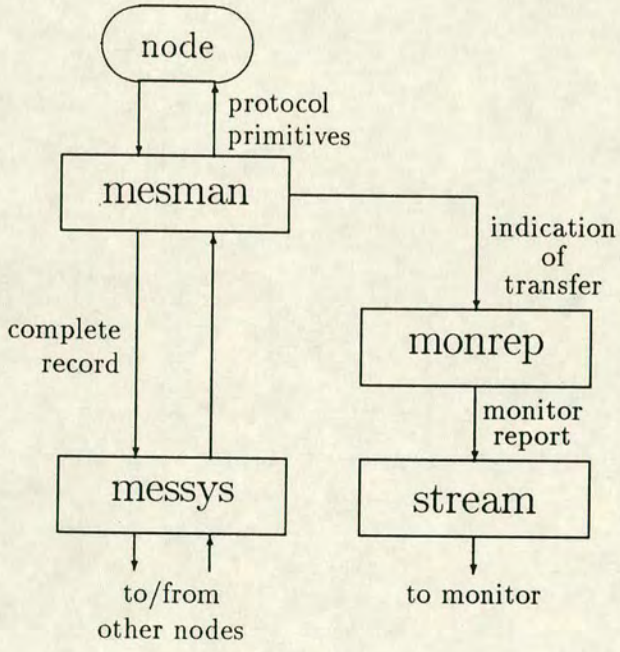

Figure 6–5: Reporting Record Transfers to the Monitor

Figure 6–5 indicates the rôles of the software modules, described in section 5.2.2, in the reporting of record transfers to the monitor. Status reports are handled in a similar way, but there is no need for any **messys** routines because no TNM protocol primitives are actually being transferred. The *node* element in the illustration refers to the higher level software modules that are responsible for the actual TNM protocols.

As described in section 5.2.2, the higher level software uses the recvmx() and sendmx() functions provided by **mesman**. In turn, these functions use recvm() and sendm(), provided by **messys**, to handle the receipt and transmission of messages using the communication system. Where there is to be no notification of protocol primitive exchanges to the monitor, this is all that is involved in the normal communication between nodes. This will be the case when the experimental system has been initialised using the **sim** program, as opposed to being started by the monitor.

When a node is created, it calls initms(), provided by **mesman**, to establish its initial operating environment using the **nollib** and **serlib** modules. Additionally, an attempt is made to notify the monitor of the birth of the node, using the status reporting mechanism of **monrep**. If the node cannot contact the monitor at its well known socket, the node assumes that there is no monitor and sets an internally monitored flag to indicate that no event messages are to be issued.

Where contact is made with the monitor, the node '*born*' message is recorded by the monitor, and results in a new entry being made in its internal list of active nodes. All subsequent messages to the monitor, until after the node '*died*' message, are associated with the initial entry so that all of the corresponding messages appear in the same window on the screen. The operation of the display is described further in section 6.1.4.

If event reporting is enabled, when recvmx() or sendmx() are called, the tf_report() function in **monrep** is used to report the transfer of the record to the monitor. The tf_report() function forms a transfer report using the node's

name and the identity, type and permit for the original message. It then uses sends(), in the **stream** module, to pass the report onto the monitor.

Whilst it may seem more natural to use *datagrams* for making reports to a monitor, the **stream** function sendms() provides the benefit of using a handshaking mechanism. This feature is useful here because it provides the user with some control over the other nodes in the system. Whenever a report is sent to the monitor, the node concerned will be forced into a *wait state* until an appropriate reply is received from the monitor - on the same channel. Thus, by handing control to the user over when this reply is sent, the *single-step* mode of operation is implemented. So, whenever the user presses a key, a reply is issued to the waiting node and a new report is accepted.

6.1.4 Display

Status Windows

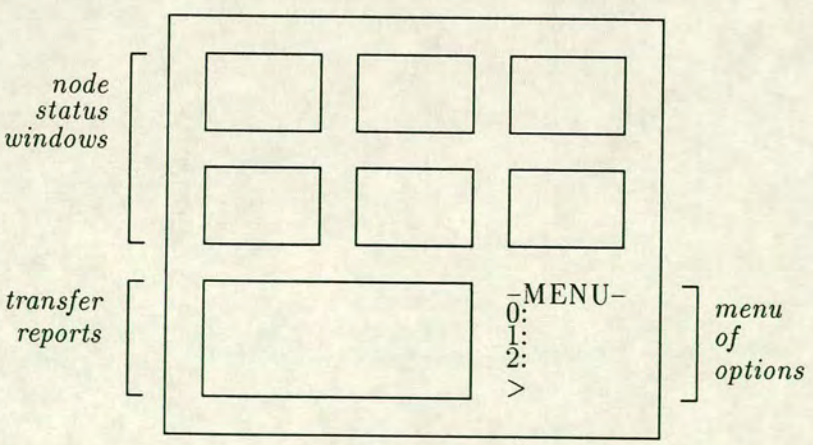

Figure 6–6: Monitor Display Format

The layout of the monitor's windowing display is shown in figure 6–6. The upper portion of the display may have up to six windows active at one time, or none at all, according to the behaviour of the system. Each of these upper windows, when present, is associated with a single node, whose name appears in the upper edge of the window's frame. The windows indicate the changes in the state of the corresponding nodes, with the oldest information scrolling up and out of sight.

The node status windows are created when a node '*born*' report is received by the monitor. On receipt of this message, the monitor records the node's name in a table of active nodes, and then allocates a window, if one is available. Thereafter, all event reports issued by that mode are indicated by a new status message in the corresponding window. The last message to be received from a node, a node '*died*' report, is written to the window and, after a brief pause, the window is cleared and the node's entry in the table is deleted.

Common Window

Since there is a limit of six node status windows, the lower, larger window acts as an overflow. If a new node fails to gain an associated status window, its event reports default to this lower window - the **common** window. A node can only open a status window with its initial '*born*' message, so extra nodes will continue to have their state changes recorded in the common window even when any of the upper windows become available. This feature is exploited in the monitor's *record only* mode of operation, described in section 6.1.2. To avoid having the service provider nodes take over all of the status windows, the monitor can be put

into the *record only* mode during their creation, and then revert to the *display all* mode. Subsequent nodes will be allocated status windows on their creation, but the changes in the states of the service provider nodes will all be logged in the common window.

The common window is used for all messages that are relevant to the operation of the monitor itself, as opposed to being pertinent only to a single node. These messages include confirmations of user directives, such as the issuing of a service request. However, the primary rôle of the common window is for the display of transfer reports. The status windows for the end nodes in a record transfer will indicate the issue and receipt of a message, and the common window gives the type of protocol primitive, the name of the source node, the identity value and the permit for the record. Taken together, these event reports describe the TNM protocol primitive exchange that has just occurred. Since the common window scrolls in the same manner as the node status windows, the most recent series of TNM protocol primitive exchanges is visible at any one time.

Menus

The lower right-hand portion of the display is reserved for a set of option menus. The menus are *context sensitive*, and are redrawn according to the mode of operation of the monitor and recent user directives. For example, if the monitor is running normally and the user opts to issue a new service request, the menu displayed will give a list of available services. This is because the user input control is incapable of allowing the direct entry of service requests. So, a predetermined

list of service request options is made available, and the user merely selects one of these options.

All of the menus provide a list of options, with each entry preceded by a value from 0 to 9 (or less if there are fewer options). The user is prompted for input only when it is required, and must then select a value in the valid range. An invalid entry usually has no effect. The exception to this is with the standard menu when the *single-step* mode of operation is being used. In this case, certain special keys, such as the space bar, are treated as an indication to the monitor to allow a step to the next state.

Display Mechanism

All of the display operations are implemented by calls to the libraries of functions provided by **curses**[3]. These functions support the windowing operations, text drawing and blanking. Furthermore, the functions provided by **curses** are independent of machine and output device, enabling the monitor to be ported quite readily if necessary.

6.1.5 Logging

The dynamic nature of the monitor's display makes it suitable for the interactive operation required by the user for observing the effect of particular events. Furthermore, the display provides a good way in which to gain a clear overview of how the implementation behaves. However, the display is not suitable for more detailed study of, say, how a network service is implemented. Similarly, when

faults occur, it is frequently useful to be able to look over the recent history of node message exchanges.

Every report received by the monitor is logged to a file. The information in the file is partially decoded, and presented in a tabular form to assist inspection. The logfile provides a chronological record of all of the events that occurred in the session. Whilst the information given in each entry is brief, further details may often be derived from the log files kept by the individual nodes. The extra details contained in the node's local logfile is of greatest use in locating the source of problems once the monitor's logfile has indicated a specific transaction. Individually, the nodes' local log files do not provide a clear view of what occurred in the session. This rôle is filled by the monitor's own logfile.

Although the monitor's logfile gives a useful overview of the behaviour of the corresponding sessions, it is not the ideal form for such inspections. To provide greater clarity, the analysis tools **pp1** and **pp2** were developed. The next section explains how these tools are used, how they function and the benefits they provide.

6.2 Operational Analysis

The primary function of the analysis tools is to provide a clearer format for the inspection of the results of using the TNM protocols. In particular, checking for the correct exchange of protocol primitives is a major concern, along with tracking down the cause of faults. Additionally, the diagrams produced by the tools are useful for documentation.

The two analysis tools **pp1** and **pp2** derive their names from their rôle and the sequence in which they must be applied. The *pp* stands for *post-processor*, because both tools are applied once the processing for a particular group of sessions is complete. Further, pp1 must be used before pp2, since the former generates output that the latter requires as input.

As mentioned in the preceding section, pp1 and pp2 take the logfile produced by the monitor and convert the information that it holds into a diagrammatic form. The result of executing these tools in their prescribed sequence is a file containing directives for the LaTeX typesetting package.

The function of translating the monitor's logfile entries into diagrams is divided between two tools to permit some user interaction between the invocation of each one. This gives the user control over the presentation of the information in the resulting diagrams. The user may, of course, directly change the output from pp2, but the output from pp1 is in a clearer and more manageable format. Furthermore, pp2 accepts some user directives in its input file.

6.2.1 Function of the Tools

The two analysis tools have very different rôles, but they are both primarily concerned with sequences of record transfers. The division of responsibility between the two tools is as follows :-

pp1 : is concerned with extracting information from the monitor's logfile and compacting it into a manageable form.

pp2 : uses the details produced by pp1 in the generation of LaTeX primitives for diagrams.

PP1

The logfile produced by the monitor has an entry for every event that was notified by the nodes in the system. A description of these events was given in section 6.1.3. **pp1** works through the logfile, determining the nature of the event for each entry and handling the data accordingly. The entries recording the '*birth*' of new nodes are collected to form a list of all the active nodes in the system. The logfile reports for the '*death*' of nodes, or the state of temporary monitor processes, are ignored.

The most important events are the state changes of the nodes and reports of the transfer of TNM protocol primitives. The details in these entries are gathered by pp1 to create an ordered sequence of record transfers. Once this list is complete, as determined by the end of the logfile, pp1 moves into its second stage and produces the intermediate file required by pp2. This file details the two lists formed by pp1 : the list of active nodes and the sequence of TNM protocol primitive transfers.

The intermediate file produced by pp1 contains entries of two types :

1. control entries, with an asterisk as the first character on the line, to convey information that pp2 will require to manage the generation of diagrams.

2. details of record transfers, with a hyphen as the leading character.

The control entries specify the number of active nodes, the number of record transfers listed in the file and mark the end of the file. Also, the names of all

the nodes are given together with an associated identification number. These identification numbers are used in place of the node names for each record transfer. There is a pair of numbers for each record, giving the source and destination nodes. Additionally, each entry indicates the record identity, the permit value and the type of the primitive. All of this information is used by pp2 in the generation of diagrams.

In addition to producing the intermediate file that pp2 will use in the production of diagrams, pp1 also records a log of how it treated the actual reports in the original logfile and how it formed the set of record transfers. In the event of the diagram containing faults, pp1's log may be the best place for the user to look in the search for the cause of those faults. Since pp1 uses a set of event reports to produce a single transfer entry in the intermediate file, the log may indicate any mistakes made in the derivation of the record transfer entries. Similarly, pp1 has expectations of how the monitor should present its event reports. So, any inconsistencies in the monitor's recording of reports should be apparent from the pp1 log.

PP2

As well as the two types of entry in the intermediary file that have already been identified, pp2 also recognises a third type. These entries are preceded by an exclamation mark and give the user some extra control over how pp2 produces the diagrams. They allow the user to specify a caption and a label for a diagram, and also to mark where diagrams should begin and end. Additionally, these entries can change the parameters that control the sizes of diagrams by fixing the maximum

number of transfers and nodes in a diagram. To appreciate how these values affect the diagram production, it is necessary to consider the algorithm used by pp2.

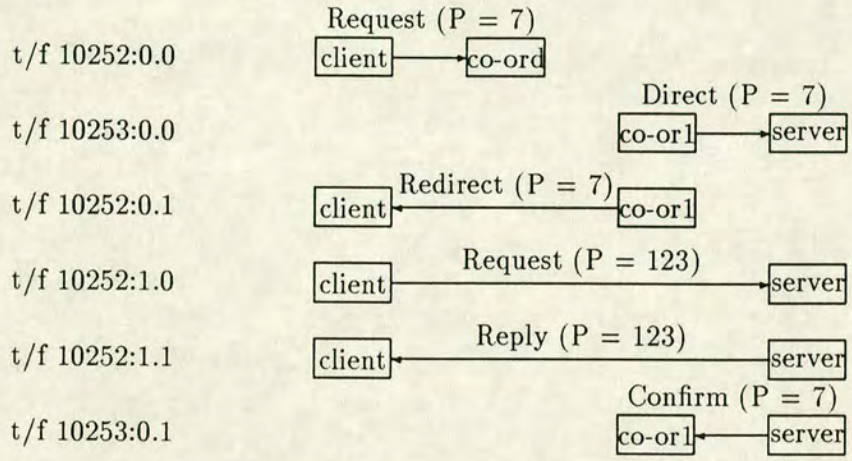

Figure 6–7: Example of Output from Analysis Tools

Figure 6–7 is an example of the diagrams produced by pp2. In the diagram, each transfer appears on a new line that begins with its identity. The two boxes represent the communicating nodes and the arrow indicates the direction of the transfer. Above the arrow is the type of the TNM protocol primitive and the value of the permit. A key feature of the diagram is the tabulation, such that each node always appears in the same column. This helps to make the diagrams clearer to read, and is an important consideration in the generation of diagrams by pp2.

The decision as to which nodes should be placed in each of the columns is determined by the order in which the nodes are read in from the intermediary file by pp2. Hence, the user can directly manage the positioning of nodes merely by reordering the control entries that identify the nodes. So, for example, the control entries in the intermediate file for figure 6–7 are as follows:

⋆ NN 4

⋆ ID 3 client

⋆ ID 0 co-ord

⋆ ID 4 co-or1

⋆ ID 1 server

The identification numbers are set by pp1 and are used to associate the names of the nodes with entries describing record transfers. This system enables the user to change the names of the nodes for more meaningful mnemonics, if desired.

There are physical constraints on the production of diagrams with which pp2 must comply. The most notable are the size of the printed page and the typeface for textual information. For pp2, these limits are immutable, but there are other restrictions that are variable. These include the maximum number of record transfers that may be detailed in a single diagram and the number of nodes that may be involved. Both of these variables may be set by the user, using special directives in the intermediary file, and both values affect pp2's decisions on where to conclude one diagram and begin another.

For each diagram, pp2 reads successive record transfer entries from the intermediary file. These transfers are added to an internal table until the total number of transfers for that diagram has been read. At this stage, pp2 defines which node should appear in which column and then writes the appropriate set of LaTeX primitives to produce the new diagram. This process continues until all of the record transfers have been used.

The total number of record transfers for a single diagram is determined from the following criteria :-

- the limit on the number of nodes that may appear in a single diagram. This limit may be set by the user, but must in any case be less than that accommodated by the width of the page.

- the limit on the number of transfers for a single diagram. Again, this limit has an upper value that is fixed by the size of the page, but may have a lower value as set by the user.

- the presence of a user directive (!P) that forces the diagram to be drawn before the next transfer is read.

- the number of transfers in the intermediary file.

For every diagram, pp2 writes a start and end sequence of LaTeX primitives that includes a caption and label, and also defines the size of the diagram. Each transfer in the diagram requires a set of 5 primitives to draw the following :

1. the identity of the record

2. the source node

3. the destination node

4. an arrow, indicating the direction of the transfer

5. the type of the TNM protocol primitive, together with the value of the permit.

As with pp1, pp2 produces a separate log to record the decisions it made in the production of a new set of diagrams. Where a resulting diagram is not clear, the user may inspect this log to help discover what changes should be made to the intermediate file in order to clarify the appropriate diagram.

6.2.2 Operation

Figure 6–8 illustrates the procedure for using pp1 and pp2 to generate LaTeX diagrams from the monitor's logfile. There are eight stages shown, and the mnemonics in italics and preceded by dots indicate the types of the file that are passed between each stage :

.log : the monitor's logfile

.int : the pp1-pp2 intermediate file

.tex : contains the LaTeX primitives required for the drawing of the diagrams

.dvi : holds device independent information that describes the resulting diagram.

The user's control over the layout of the diagrams is represented by the third stage, *EDIT*. The feedback loop from stage 7 is present for the user to change the intermediate file in the light of how the resulting diagrams appear. The facility that allows the user to inspect LaTeX output on a display before making a hardcopy of the output is listed as stage 6, *PREVIEW*. Where no changes are required, stage 8, *LASER*, is responsible for the final production of the diagrams.

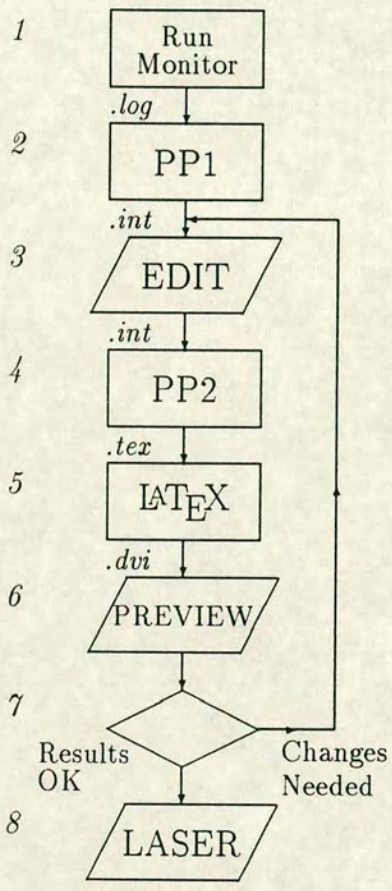

Figure 6–8: Analysis of Simulation Results

6.3 Command Shell

The support tools described in the preceding sections of this chapter are concerned with the *inner workings* of the TNM protocol set implementation. This aspect is of interest to the designer and implementor of the TNM protocols, and, to a lesser extent, to the manager of the system. In contrast, the remaining sections of this

chapter are concerned with how the implementation may be applied by the user of the system.

First, a simple command shell is described. This shell provides a rudimentary means of using network services in a similar way to local commands. In the next section, the environment management utility (EMU) is discussed. Some of the features of EMU have already been incorporated into the simple command shell, and provide more sophisticated capabilities for the system's users.

6.3.1 Purpose

As described in section 6.1.2, the monitor provides a facility for the user to initiate requests for network services. However, only a limited range of service requests is available for selection from the menu. Furthermore, this mechanism is crude and only really suitable for studying the intricacies of the implementation's behaviour.

The simple command shell was developed to provide a more useable means of generating requests for network services. Additionally, this simple shell permitted some investigation into how the user interface should be provided. To this end, the concept of **layered network transparency**, described in section 3.3.1, is used as the basis for the design of the simple shell.

6.3.2 Operation

The command shell allows the user to enter text in the same way as the C-shell [53]. A sequence of commands and parameters, separated by special delimiting characters, is entered at the keyboard, and the shell only acts on the input once

the return key has been pressed. So, the command shell is essentially formed from three stages :-

1. a **command line handler** to read keys from the keyboard and form a buffered line of text.

2. an **interpreter** that parses the current line and identifies the commands that are to be executed.

3. a **command executor** which receives successive commands from the interpreter and spawns processes to execute each of the commands.

Command Line Handler

The command line handler maintains a cyclic buffer of the most recently entered command lines. Currently, no advantage is derived from this cyclic buffer. However, it would be relatively straightforward to expand the handler's functionality to allow more sophisticated line editing. In this case, the cyclic buffer would be a useful asset.

On a single line, the command and its parameters are normally separated by spaces. However, the interpreter also supports the special characters that the C-shell uses to separate a number of commands on a single line :

- a vertical line between two commands means that the output from the first is to be piped into the second.

- an ampersand between commands signifies that the second command should be initiated immediately after the first, without waiting for its completion. Hence the two commands may be executed concurrently.

- a semicolon indicates that the preceding command should be complete before the command that follows it is executed. Thus, the two commands must be executed consecutively.

Interpreter

When the interpreter passes commands to the executor, the types of the delimiting characters are supplied so that the appropriate communication channels are established. Similarly, the executor creates processes for the various commands according to whether their execution is to be sequential or in parallel.

The command interpreter also checks the commands themselves before deciding whether they should be passed on to the executor. This is to detect the presence of commands that are implemented internally by the shell. This includes functions that alter shell variables (including **cd**, the *change directory* command) and allows for special EMU operations. The latter may involve the expansion or substitution of the commands' parameters before the command is passed to the executor.

Command Executor

Where the command is not internal to the shell, it is passed to the executor with its parameters. It is the executor's responsibility to discern whether the command is supported locally or by the network. This is achieved by following the *directory*

search path for local commands and, if the command is not in this path, executing the *S-R* program to request the network service. The search path comprises a list of directories in which commands are expected to reside. The order in which directories appear in this list indicates their relative priority.

A conventional shell will expect all commands to exist in the search path, and failure to locate the command will result in an error message. The simple command shell effectively adds an extra check at the end of the search path to allow for the provision of a suitable service on the network.

Algorithm

The basic operation of the simple command shell may be expressed in an algorithmic form.

```
while TRUE do
        read line from keyboard into buffer
        find tokens in buffer
        repeat
                extract command from buffer
                if EMU command
                        then
                                expand command parameters
                if NOT internal command
                        then
                                execute command
        until empty buffer
```

The step that concerns the search for *tokens* in the buffered line is to speed up the operation of the inner *repeat-until* loop. At this point, a structure is formed with a pointer into the line buffer for every element of the line, i.e. all of the

commands, parameters and delimiters. Subsequently, the extraction of commands, and associated parameters, may be undertaken simply by progressing through the list of tokens.

Example

The features of the simple command shell may be illustrated by an example.

ptime ; demo | wc -l ; ptime

ptime is a network service that prints the current network time.

demo is a network service, printing a number of lines of text.

wc is a standard Unix function that counts the number of characters, words and lines in the specified file. The **-l** option results in the line count being the only one to be printed.

This command line will cause a service provider to send a stream of text to the current station, which will count the number of lines transferred and also print the network time at the start and the end of the textual transfer. First of all, the executor of the simple shell will spawn an *S-R* process to request the **ptime** service. Then it will spawn two processes : one to request the **demo** service and another to run the **wc** program with the **-l** parameter. A pipe will be established between the two processes before they run their respective programs. Finally, the executor will spawn another *S-R* process to request the **ptime** service for the second time. The resulting output will indicate the time taken to transfer a number of lines of text, and the transfer rate may be calculated.

6.3.3 Local and Remote Commands

Response Time

The use of the directory search path was discussed in the previous section. The command executor looks in each directory of the path until it finds the program for the specified command. If the command cannot be reached through the path, the simple command shell then attempts to access an appropriate network service. Only if the network service request is unsuccessful does the command shell indicate to the user that the command does not exist. If the user had made an error when typing in the command, there may be an appreciable delay before he is given a chance to correct the mistake. In fact, the response time in general may be unacceptably long because of the need to search through the local directories in the path before even requesting the network service.

An alternative approach would be to issue the network service request before beginning the search through the local directories, and allow both operations to proceed concurrently. The first successful result would then be accepted, and the other option cancelled. This approach should mean that the response time for network services is improved. However, the response for simple local commands would be impaired by the extra processing overhead. Furthermore, the network would be swamped with unnecessary service requests. Bearing in mind the fact that most of the requested functions will be implemented locally, this solution is not suitable for the general case.

A better approach would be to use a pre-formed hash look-up table for accessing the programs for commands. The most common network services would also

be included in this table, but with a special entry that causes the shell to make a network service request. This solution would result in local and remote functions having similar look-up overheads, but with the network services having the additional loads of the *S-R* process, the TNM protocol overhead and the network's latency.

Multiple Implementations

So far, the discussion of local and remote services access has assumed that the requested command is only provided in one way. However, it is likely that a number of functions will be supplied by both the network and the user's own machine. In this case, the decision as to which instance of the service is to be used should depend on the performance of each implementation.

Best Performance	Choice	Reason
local	local	ideal solution
equal	local	no network overheads
remote slightly better	local	network overhead can be unpredictable
remote much better	remote	performance improvement

Table 6–1: Choice of Local or Remote Service

Table 6–1 gives an indication of an approach for deciding whether to use the local or remote implementation of a command. The assessment of 'how much better' the performance may be for the remote service will depend on a number of factors, such as the amount of input data, network load, service provider performance, etc. The shell should be provided with enough information to be able to make its decision. Ideally, this information should be reached through a table associated with the command look-up table for ease of access.

The simple shell does not accommodate the situation in which a particular function is implemented both locally and remotely. The question of how best to overcome this problem, both within the simple shell and more generally, is beyond the scope of this thesis.

6.4 Environment Management

The **Environment Management Utility** (EMU) is independent of the work on the Triadic Network Model and its related protocols. However, the creation of the utility has assisted the development of the experimental implementation. EMU is described here for this reason and also because there are features of the utility that would be well suited for inclusion in a more sophisticated computing environment using the TNM protocols.

EMU has been developed under the DEC VAX/VMS operating system, but there is also a partial implementation for 4.2 BSD Unix. Some integration of EMU functions in the simple command shell, described in section 6.3, has been achieved. The decision to undertake the EMU development using the VMS control language DCL (DEC Control Language) was based on the power and versatility of the language. Additionally, at that time the greatest need for EMU facilities existed on the VAX because the machine was being used for the early work on the Triadic Network Model.

6.4.1 Purpose of Development

EMU was developed to permit investigation into four key areas :-

1. more extensive use of *system default parameters.*

2. the *preservation* of states at the end of a **session**, and the subsequent restoration of those states on *commencement* of the next session.

3. the identification of different work **activities** underway in a single terminal session.

4. the management of a multitude of computing **environments**, each suited to a particular line of work.

Individually, each of these facilities contributed towards the sophisticated computing environment that the TNM protocol set was designed to support. However, when taken together, they would provide the user with a powerful, yet simple to use, means of managing a multi-task work load. On reflection, the flexibility afforded by EMU during the course of the work on the model and protocols has demonstrated the advancement in task management that these ideas can provide.

6.4.2 Principles

System Default Parameters

The technique of providing default options for commands is widely accepted and is used extensively by a number of operating systems. For example :

MUSS : allows use of a *current file* that acts as the default argument when no other is specified. So, invoking the editor or printer spooler without giving the name of a file will cause the special file called '**0**' to be used. However, if the contents of '0' are to be preserved, an explicit storing operation is required.

Unix : The hierarchical tree structure can result in the full pathname for a file being quite long. So, Unix maintains a shell variable **cwd** which is the **current working directory**. All partial references to files are expanded by the shell to be relative to this default directory.

VMS : also maintains a current working directory, established by using the SET DEFAULT command. Additionally, a default filename *extension* may be specified[1]. Subsequently, file parameters that are given without an extension will automatically have the default extension added. Other commands will require files to have a particular extension, and will add this instead. So, to the user, file name extensions are beyond his concern and this allows the system to manage files almost transparently.

It is apparent that these operating systems are only making limited use of the default options. EMU extends the use of default parameters to further simplify the user's file handling responsibilities.

---

[1]A filename comprises a primary name followed by a dot and an *extension*. Both parts of the filename are formed from a sequence of alphanumeric characters, and the extension is usually restricted to 3 characters.

EMU uses DCL system variables to maintain the following group of default parameters :-

- the name of the current file
- the extension to the current file
- the directory in which the current file resides
- the active printer queue
- the name of the active batch queue.

In addition to these, EMU forms a 3 deep stack of the most recent instances of the current file name, its extension and directory. The current file acts as the top of the stack. So, when a change is made to the current entry, the previous entries are all shuffled down by one position, with the oldest entry being lost. It is possible to extract one of the entries on the stack. This then becomes the current file and the stack is partially shuffled.

For commands that are supported by the EMU system, omission of the file name parameter implies that the current file should be used. The other entries in the stack may be referred to simply by quoting the position within the stack, i.e. 1, 2 or 3. The current file may also be referred to by giving its position in the stack - 0. This is needed where a *binary operation* is requested. For example,

CF 0 temp.dat

will copy the contents of the current file to a new file called **temp.dat**, in the same directory. This new file then becomes the current file and the previous file is shuffled down the stack.

The name, extension and directory for the current file are usually defined by being specified as the filename parameter for an EMU-supported command, as in the previous example. Where any of these values are ommitted, the existing default is maintained and used for that operation. So, in the previous example, the directory was omitted and therefore EMU used the same directory as the current file. If the extension **.dat** had not been specified, the current file's extension would have been used as well.

There is a special EMU command **NEW** that explicitly defines a new current set of defaults. The stack may be cleared by issuing the **FL** (or **FLUSH**) command, and this may be combined with setting the new defaults by using **NEWF**. The current defaults may be listed by requesting **XX** for the current file, or **XS** for the entries on the stack.

Where the user's terminal is of a type supported by EMU's screen handling routines, the current file and the stack will appear on status lines on the screen. A lower intensity is used so that this information is not confusing; the current file name, extension and directory are displayed at the top right-hand of the screen; the file names and extensions for the three stack entries are given at the bottom of the screen. Where any of these entries on the stack have different characteristics to the current default, the entry is displayed in reverse video.

End of Session State Preservation

Frequently, work on a particular topic will take more than one terminal session to complete. Normally, commencement of a new session involves an initialisation period, during which the user re-establishes all of the defaults that were active at the conclusion of the preceding session. The duration of this period may be quite lengthy, especially if there is a large interval since the preceding session.

EMU is particularly concerned with reducing this inefficient period. This is achieved by preserving the current defaults at the end of work on a topic, then restoring these defaults when work recommences. The default parameters that are stored are the same as those maintained by EMU to simplify the entry of commands by the user. The file holding this information is designated a *history file*, and has a name defined by the user with an extension **.his**.

Hence, EMU enables work to resume on the previous topic with a much lower initial overhead, since the user is quickly placed in the same position as he was at the end of the previous session. EMU provides a facility for recording *reminder messages* to assist the user further in his recollection of his current stage of work on that topic.

Activities

During any single terminal session, the user is likely to spend time on a number of different areas of work. Similarly, it would be unusual for a collection of terminal sessions to be concerned purely with a specific task, although this may well be the case. Hence, EMU allows a distinction to be made between different areas of work

by treating each one as a separate **activity**. For each activity, a different *history* file is recorded so that the user may change between activities quite readily. In each case, the user can resume an area of work with the defaults restored to their values at the time of the last instance of usage of that activity.

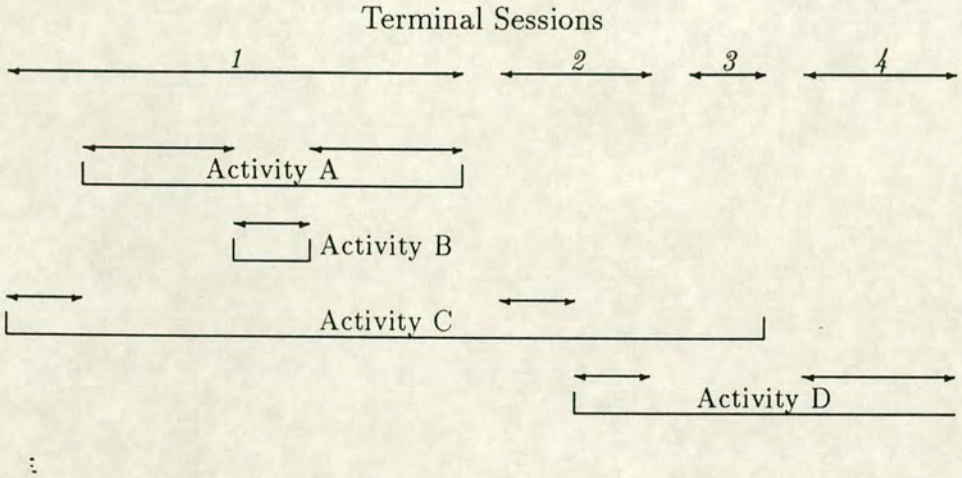

Figure 6–9: Relating Activities to Sessions

An activity *starts* when a new line of work begins, and *ends* when that area of work is complete. Figure 6–9 illustrates how the lifetime of an activity may be smaller than that of the terminal session in which it is begun, or how it may be composed of a number of sessions. In figure 6–9 :

Activity A : is started during terminal session 1. It is briefly suspended, perhaps for work on a different, but related, activity (B). It is completed before the end of session 1, so its lifetime is less than one session.

Activity B : begins during the interval between work on activity A. The activity is completed without interruption.

Activity C : occupies more than one terminal session. It is started in session 1, continued in session 2 and concluded in session 3.

Activity D : is continued indefinitely, and may be resumed in any number of sessions after being created in session 2.

Environments

A group of activities may all be concerned with a similar line of work. For example, the user may have a collection of activities involved with the production of reports, papers and articles. All of the activities may be categorised as *text processing*, and will all involve similar operations. Additional facilities may be provided for distinct such areas of work. These extra facilities serve to enhance the user's working environment, but only for activities in related areas of work.

Since the grouping of activities is a frequent occurrence, EMU associates every activity with a corresponding environment. Where there are additional features provided by that environment, they are made available for work on that activity. For example, the '**report**' environment is for text processing activities. Activities associated with this environment inherit a small set of commands to simplify the processing of text by LaTeX. In particular, there is a function to spawn a background task to run LaTeX with appropriate files, where these are determined from the current defaults. Further commands provide the user with control over how the background task performs.

All activities must be associated with a named environment, and the user is

required to **enter** an environment before an activity may be started or **resumed**[2]. New environments may be created using a special EMU environment that also allows the user to list, modify or delete existing environments. A distinction is made between *system* and *user* environments to protect certain privileged environments from accidental alterations.

It is recognised that some tasks will be short lived, and will therefore be over burdened by the EMU mechanisms. EMU accommodates these tasks through the use of the **null** environment and activity. A null activity will be provided with the full use of EMU and the current environment's facilities, but there will be no preservation of states at the end of that area of work. A null environment provides no additional features to those of EMU, and may only have the null activity underway.

6.4.3 Application

EMU and the Simple Shell

EMU has been designed to provide extra flexibility with assistance for a multi-task workload. The simple shell provides access to the standard Unix command set as well as network services, as supplied via the TNM protocols. Together, EMU and the shell would enable users to enjoy the rich functionality of a networked computing environment but with the structure of EMU to simplify access to these resources. EMU's environments could be tailored to the needs of the users, so that

---

[2]**Enter** and **resume** are special EMU commands that cause the current environment and activity to be replaced by those specified by the user.

procedures for accessing sophisticated network services would become as easy to use as simple local commands. This would help to conceal the complexity of the network from the users, whilst enabling full use of the wealth of services. This would be as intended in the design of the Triadic Network Model.

Chapter 7

Conclusion

The final chapter of this thesis reviews the work described in the preceding chapters, and puts it in the context of what has been achieved overall. The current state of the research and the experimental implementation is given together with some possible areas for improvement. The chapter goes on to consider directions for any further work that may be undertaken as a result of this research. Finally, a statement is made of the results and the achievements of the research embodied in the thesis.

7.1 Review

The thesis began with a statement of the direction of the research, setting an ultimate target of a multiple computer system, comprising a wide range of machines with different architectures and instruction sets, yet all cooperating in the provision of a powerful computing service for the users. Whilst the implementation of this target system is beyond the scope of this thesis, the communication re-

quirements of this environment are of primary concern. In this respect, the stated objective of the thesis is the derivation of network protocols that are oriented towards the needs of the target system.

The thesis continued with a study of existing multiple computer systems. The review of these systems in chapter 2 concentrated on the key ideas and techniques that are fundamental to work in this field. However, none of the systems selected for inclusion in this study embodied all the characteristics of the target system. Hence, chapter 3 began with a description of a network structure that combined many of the features of the reviewed systems. The network devices in this general network structure were then classified into 3 groups, according to their rôles and communication needs.

The three-way classification of network devices forms the basis of the Triadic Network Model, which defines the interactions within and between the three groups. The model's specification of network communication provides the foundations for the development of protocols that are tailored to the needs of the networked computing environment. The principles of the protocols based on the model are described in chapter 4. The implementation of the protocol set is covered in chapter 5, with an explanation of the software tools developed to support the implementation in chapter 6.

7.2 Current State

Chapters 5 and 6 concentrated on key aspects of the implementation, but it is worthwhile to take a brief look at the present state of this work. The three TNM protocols were described in chapters 4 and 5, and more comprehensive definitions of the protocols are given in appendices A, B and C. Only a subset of these protocols has been implemented on the HLH Orion minicomputer. The emphasis in the implementation has been on the core primitives of the TNM *basic* protocol, because of the use of the three-party mechanism, section 4.2.

The experimental implementation has allowed many of the aspects of the protocols to be tried out. In particular, interfacing to the 4.2 BSD Unix socket mechanism illustrated the benefits of the modular structure of the implementation. The socket mechanism is designed to provide network communication implementors with a great deal of flexibility in their use of the Unix network services. However, this is only possible by requiring the higher level software to have a high degree of awareness of the service provided by the lower level networking software. For the TNM protocols, the flexibility of the sockets interface provided unnecessary complications.

The implementation uses a tightly defined interface to the communication system. This is more restrictive than the socket mechanism, but satisifies the needs of the software modules responsible for the handling of TNM protocol set primitives. There is the further advantage that the implementation is more easily ported to another system because only the interface functions need to be rewritten.

The experimental implementation has also enabled some consideration of the problems associated with spawning sub-tasks in order to simplify the operation of a single node. This investigation was aided by the support tools, which have also undergone a degree of development.

The monitor process is constrained by the limits of the windowing interface provided by the 4.2 BSD Unix **curses** functions[3], but was designed to cope with a much larger number of nodes than can be displayed on the screen. The monitor is supplemented by a much more streamlined control process, **sim**. **Sim** uses the same configuration files as **monitor**, but terminates once all of the nodes have been created. This means that **sim** is applicable for use in the start-up sequence of Unix systems.

The two analysis tools, **pp1** and **pp2**, were developed for a specific job, and as such have no path for further enhancement. However, the automatic generation of diagrams from the log files of the monitor has proven invaluable in a number of respects :-

1. the diagnosis of faults is achieved more rapidly than using just the log files.

2. analysis of the effects of refinements in the operation of the protocols is simplified.

3. documentation is assisted by the inclusion of these diagrams.

The remaining support tools, the simple command shell and the environment management utility (EMU), both have clear paths for further development. Currently, EMU is only fully implemented under VAX/VMS, but there is a partial

implementation for 4.2 BSD Unix. Some integration of EMU functions into the simple command shell has been achieved.

7.3 Areas for Improvement

To be of greatest benefit, the TNM protocol set should have a high performance when used in the target networked computing environment. In the experimental implementation, the focus has been on trying out new ideas and obtaining clear feedback on how they performed. Consequently, there is plenty of scope for improvement in the operation of the TNM protocol set implementation.

7.3.1 Heavyweight Processes

One aspect of the implementation that particularly warrants some effort to reduce its inefficiency is the spawning of sub-tasks and new processes. Under 4.2 BSD Unix there is no means of differentiating between these two forms of child process, but the implementation uses each in a different way.

Where a service provider is required, under the three-party mechanism, to act as a service manager, a new process is spawned for the limited duration of the need for this node, as explained in section 5.2.3. The 4.2 BSD Unix creates a new application process for the service manager, with its own area of virtual memory. The operating system performs a full security check on all procedure calls, and the process is subject to the usual scheduling pattern. This is a significant overhead for a node that is infrequently involved in TNM protocol operation. A more effective

approach would permit the sub-task to share the memory structures of the parent process directly[1] and operate with a higher level of privilege than a normal user process. Such tasks are often referred to as *lightweight processes*, as in the LOCUS operating system, for example, as described in section 2.3.2.

The second way in which sub-processes are used by the TNM protocol set implementation is in the provision of the requested services. Since these services will usually take the form of application programs, it is appropriate to treat them as conventional user programs. However, there are services that are much closer to the operating system kernel. These service have much in common with the Unix *server processes* that are implemented by process dæmons, i.e. the processes have already been created, but are dormant, awaiting input. Frequently requested services, such as the translation of names to network addresses, should be provided in this way. This is because the overheads of normal process management would become significant for such services.

7.3.2 Transport Services

As explained in section 5.2.2, for the experimental implementation, the TNM protocols use the transport service provided by TCP and UDP, as supported by the Unix socket mechanism. TCP satisfies the TNM protocol set's requirement for a reliable underlying communication system; UDP is less reliable, but its use permitted some assessment of the advantages of using a number of transport services

[1]The experimental implementation already includes mechanisms to support shared access to the appropriate structures.

- each according to its merits. However, the choice of TCP and UDP was forced on the implementation by the limitations of the development environment, and both protocols are a little cumbersome for a high performance networked computing system.

The TNM protocols would be much more effective were they to be provided above a streamlined transport service that had been designed for operation in a high performance environment. For example, a transport layer running above the Centrenet Burst Protocol, section 1.2.3, would be particularly suitable. The re-working needed to transform the current implementation to use this more efficient service would be minimal because of the communication system interface functions provided by the **messys** module, section 5.2.2.

7.3.3 Monitoring

The **monitor** tool has proven to be extremely useful in observing the behaviour of processes created during the operation of the TNM protocol set implementation. The most important information derived by the **monitor** is the sequence in which TNM protocol primitives are exchanged, although the creation and subsequent termination of sub-processes are of secondary interest. For the experimental implementation, all of the nodes in the system were present on a single processor and the most straightforward method for obtaining the required information was through the use of a separate monitor node. However, the overhead of having to report every protocol primitive exchange to the monitor would be impractical for a working implementation.

The conventional method for observing network conversations is to use a special montoring node that listens to all of the traffic on the network. This method only works where the transmission medium has a broadcast nature, for example ethernet, and the monitor device can be made to filter out unwanted traffic. Some of the network analyzers that are commercially available may be programmed to decode proprietary or experimental network protocols. By collecting a sequence of network transmissions, and interpreting them according to the TNM protocols, analysis of protocol primitive exchanges is still possible with a network analyzer, but with no overhead on the communicating nodes. This contrasts with the **monitor** node that requires the communicating nodes to send event reporting messages in addition to their normal interactions.

7.4 Further Work

During the course of the research described in this thesis, it has become apparent that there are a number of related topics that could warrant further investigation. Additionally, the work described here suggests a number of aspects that could be developed, but would not justify research effort in their own right. There are also new products and techniques that may have some bearing on this work and maybe worthy of further investigation. This section takes an inexhaustive look at these points, with some guidelines on how further work could proceed.

7.4.1 Higher Performance

The aspects of the experimental implementation that were highlighted in section 7.3 could be reworked to provide some improvements in efficiency. However, for the high level of performance appropriate to the target system, even greater measures are necessary. In particular, the protocols would have to be "moved closer to the hardware" on which they were being used. There are two aspects to this :-

1. The TNM protocol set would need to be implemented in the *firmware* of a dedicated network controller. This would relieve the host processor of the burden of communication protocol management. Furthermore, the network controller would be more effective because it would not have the same demands as the host processor's operating system.

2. The hardware of the network controller should be designed to treat the different forms of network communication in the most effective manner.

7.4.2 Specification, Verification and Validation

The TNM protocols are specified in terms of

- the descriptions in the chapters of this thesis
- the state diagrams and tables of appendices A, B and C.

It is difficult to validate experimental protocols when they are specified in this form [95], and verification that an implementation is correct is unreliable. However, were the protocols to be specified in more formal terms, using a description language such as **LOTOS** [89], both validation and verification of the TNM protocols would be simplified.

7.4.3 User Interface

The simple command shell, described in section 6.3, has been enhanced to include some of the basic operations of the environment management utility (EMU). Further evaluation of the experimental implementation would benefit from continued development of the EMU features in the shell. Additionally, there remains some investigation that may be performed on the possibilities for expanding the service routing capabilities of the shell.

The shell provides a subset of the facilities available in the simpler of the Unix command interpreters. For commands that are implemented by user programs

that exist in the search path, the shell merely spawns a process to execute that program, as with other shells. For network services, the shell uses TNM protocols to access this service on a remote machine. However, where the service is available both locally and remotely, the shell requires additional information in order that it can select the most suitable option.

Further research is warranted into the problem of how service requests should be mapped onto available providers in this environment. This problem exists both in this line of research and generally in this field. For the Triadic Network Model, there is an additional question of how the user communication facilities of the TNM *user* protocol could be exploited in a more advanced shell.

7.4.4 Support for Objects

The expanding use of object oriented applications leads to the question of how well the TNM protocols can support the manipulation of objects. The TNM protocol set can readily be used to transfer objects, as described in section 4.3.3, by transporting the name and type of an enclosed object as explicit characteristics. The object's capability is not carried explicitly by the TNM protocols, but could be included as part of the contents of a record.

For an object oriented system, the distribution of the mechanisms of the system over a number of machines may be of interest. In this case, some investigation of how well these mechanisms map onto the TNM protocol set may be justified.

7.4.5 Network Management

Multiple computer installations are becoming increasingly complex. Not only do such systems comprise a number of heterogeneous nodes, but they are often composed of a number of sub-nets, which in turn may involve different media. This requires a large number of network bridges and gateways, in addition to those devices required for terminal and computer connections.

Networks of this complexity can have an unreliable performance because the possibilities for failure or disruption are greatly increased. Furthermore, the detection and remedy of faults is made more difficult by this complexity; the *downtime* for a single failure may be increased and the disruption magnified because of the larger number of users involved.

In an attempt to monitor and control large networked systems, there are moves to introduce international standards for network management protocols. **CMIP** (Common Management Information Protocol), **CMOT** (CMIP over TCP) and **SNMP** (Simple Network Management Protocol) are examples of recently introduced protocols for network management. Each of these protocols has had a limited degree of acceptance by the major computer manufacturers, but none of them has been the subject of a universal commitment. This is primarily because it is still unclear which protocol will have the greatest backing.

All of the network management protocols allow for monitoring and control information to be passed to special nodes, with the data transfer being outwith the normal communication system. The TNM protocols, although derived independently, share these characteristics. In particular, the rôle of a BEM manager

is allied to that of a network management station. Further, the more general purpose service manager (*S-M*) node has a related function.

An investigation of the TNM protocol set's applicability for network management should be investigated along with any work on the transfer of data objects. This is because it it likely that the eventual standards for network management will use **ASN1** (Abstract Syntax Notation 1 - derived from the CCITT X.409[17]), which is used for defining the types and characteristics of embedded data objects.

7.4.6 NFS

It was mentioned in section 5.1.1 that some difficulty was encountered in porting the TNM protocol set implementation onto Sun workstations. This was chiefly because of conflicts with the higher level networking software - the **NFS** (Network File System). However, NFS is now supported by a large number of computer suppliers. So, any attempts to expand the use of the TNM protocols would benefit from some initial investigation into how aspects of the NFS implementation could be exploited.

7.5 Concluding Remarks

The underlying theme, thoughout all of the chapters of this thesis, has been the problem of how best to unify a set of computers, with different characteristics, to form a single system. In particular, the target system for this work is a heterogeneous multiple computer system, incorporating machines with different ar-

chitectures, instruction sets and levels of performance. Whilst there has been a considerable amount of research in the field of distributed computing, resulting in both experimental and commercial solutions, it is apparent that the target system still poses difficulties.

The Triadic Network Model is central to the research described in this thesis. Although the model may appear cumbersome for simpler configurations, such as a single network of intelligent homogeneous machines, it is well suited to the added complexity of the target system. The model's three types of logical module may be used to represent the elements of a distributed system, and the model can then describe the interactions between these modules. Consequently, the protocols derived from the model are well suited to supporting the operation of a distributed system in the manner defined by the model

The experimental implementation of the TNM protocols provided a platform on which ideas could be tried out. In particular, the problems of process spawning and heavyweight transport services have become more apparent. Additionally, the merits of transaction monitoring, with diagrammatic analysis, have been demonstrated. The simple command shell permitted some investigation into how the users of the target system may access the services of both the local and remote computers. The environment management utility (EMU), proving invaluable during the course of this work, provided some pointers as to how the users could eliminate some of the difficulties associated with a sophisticated networked computing system.

7.5.1 Summary

The Triadic Network Model, and its related protocols, provide a basis for the development of a high performance networked computing system. This thesis has defined both the model and the corresponding protocols, described an experimental implementation of the protocol set and presented a study of multiple computer systems. The research has enabled a fuller understanding of the problems of multiple computer connections to be realised, and illustrated how a high performance processor may be efficiently utilized on a general purpose network.

Bibliography

[1] **AM2900 Data Book**, Advanced Micro Devices.

[2] **APM Working Documents**, University of Edinburgh, May 1983.

[3] Arnold, K.C.R.C., "Screen Updating and Cursor Movement Optimization : A Library Package", Dept. of Electrical Engineering and Computer Science, University of California, Berkeley.

[4] **Apollo DOMAIN Station Reference Manual**, Cray Computer Systems technical note SN-0229, Cray Research, Inc., March 1985.

[5] **Apollo DOMAIN Station Internal Reference Manual**, Cray Computer Systems technical note SN-0230, Cray Research, Inc., March 1985.

[6] Arms, W., "The Carnegie-Mellon Andrew Project", personal notes from **Open Lectures in Computing : Campus Networks**, University of Kent at Canterbury, April 3-4, 1986.

[7] Arms, W., "The Dartmouth Personal Computer Project", personal notes from **Open Lectures in Computing : Campus Networks**, University of Kent at Canterbury, April 3-4, 1986.

[8] Ashenhurst, R.L. and Vonderohe, R.H., "A Hierarchical Network", *Datamation*, vol. 21, no. 2, Feb. 1975, pp. 40-44.

[9] Barbacci, M.R., "Personal Computing : The SPICE Project", **New Computer Architectures**, ed. J. Tiberghien, Academic Press, London, 1984, pp. 208-220.

[10] Beach, B., Pender, A. and Szabados, M., "A Multi-OS LAN", *Computer Systems*, 5, 4, April 1985, pp. 51-55.

[11] Binns, S.E., Dallas, I.N. and Spratt, E.B., "Further Developments on the Cambridge Ring Network at the University of Kent", Proc. IFIP TC-6 Int. Symp., Florence, Italy, 1982.

[12] Bird, R.P., "A Compiler Server Node in a Local Area Network", *Proc. Int. Comp. Symp. on Application Systems Development*, BG Teubner, Stuttgart, March 22-24, 1983, p. 182.

[13] Brebner, G., "Some Thoughts on a Future Computing Environment", internal notes, University of Edinburgh, October 1985.

[14] Brebner, G., "Future Communications Research", internal notes, University of Edinburgh, 13 January 1986.

[15] Bochmann, G.V. (ed.), **Architecture of Distributed Computer Systems**, Lecture Notes in Computer Science, vol. 77, 1979.

[16] Bundy, A., "Intelligent Front-Ends", *Infotech State of the Art Report on Expert Systems*, vol. 12:7, 1984, pp. 15-24.

[17] CCITT, **Message Handling Systems : Presentation Transfer and Syntax**, draft recommendation X.409.

[18] Control Data Corporation, **Loosely Coupled Network System Description**, 6 Sept. 1980.

[19] Champine, G.A., "Back-End Technology Trends", *IEEE Computer*, 13, 2, Feb. 1980, pp. 50-58.

[20] Cheriton, D.R. and Zwaenepool, W., "The Distributed V-Kernel and its Performance for Diskless Workstations", *Proc. Ninth ACM Symposium on Operating Systems Principles*, 10-13 Oct. 1983; printed in *Operating Systems Review*, 17, 5, pp. 129-140.

[21] Cheong, V.E. and Hirschheim, R.A, **Local Area Networks : Issues, Products and Developments**, John Wiley & Sons, Chichester, 1983.

[22] **Centrenet Manual**, University of Edinburgh, internal document, 5 Sept. 1986.

[23] Corley, C.J. and Statz, J.A., "LISP Workstation brings AI Power to a Users Desk", *Computer Design*, 24, 1, Jan. 1985, pp. 155-162.

[24] Daly, T., "Centrenet Burst Protocol Version 1.1", internal notes, University of Manchester, 27 June 1985.

[25] DCS, **The Co-ordinated Programme of Research in Distributed Computing Systems, 1977-1984, Final Report**, produced by the Science and Engineering Research Council, 1984.

[26] Deitel, H.M., "VM : A Virtual Machine Operating System", **An Introduction to Operating Systems**, Addison-Wesley, London, 1984, pp. 601-630.

[27] Dunsmuir, M.R.M. and Davies, G.J., **Programming the UNIX System**, Macmillan Publishers Ltd., London, 1985.

[28] Enslow, P.H., "What is a Distributed Data Processing System", *IEEE Computer*, 11, 1, Jan. 1978.

[29] Edwards, D.G.B., Knowles, A.E. and Woods, J.V., "MU6G - A New Design to Achieve Mainframe Performance from a Mini-Sized Computer", *Proc. 7th Annual Int. Symp. Comp. Arch.*, 1978, pp. 161-167.

[30] Feldman, S.I., "Make - A Program for Maintaining Computer Programs", Bell Laboratories, August 1978.

[31] Frank, G.R. and Theaker, C.J., "The Design of the MUSS Operating System", *Software - Practice and Experience*, vol. 9, pp. 599-620, 1979.

[32] ORION Microarchitecture Reference Manual, 3rd edition, High Level Hardware Ltd., Oxford 1986.

[33] Hopkins, T.M., "Provision of a Shareable Processing Resource for Sparse Vector Computation on a Local Area Network", proposal for research, University of Edinburgh, 1985.

[34] Hopkins, T.M., "The Centrenet-Ethernet Gateway", internal report, University of Edinburgh, 1986.

[35] Hopkins, T.P. and Wilson, I.R., "Distributed Network Services", internal document, University of Manchester.

[36] Hopkins, T.P., "The Design of a Local Area Network", Ph.D. thesis, University of Manchester, Sept. 1984.

[37] Hopkins, T.P., "Centrenet Digital Speech System", internal document, University of Manchester, August 25, 1984.

[38] Hopkins, T.P., "Image Transfer by Packet Switched Network", technical report UMCS-85-9-2, University of Manchester, 1985.

[39] Hutchison, D. and Walpole, J., "Eclipse : A Distributed Software Development Environment", internal report, University of Lancaster, August 1985.

[40] Huxley. R.S., "Remote Host Facility, Permanent File Transfer - Introduction", University of Manchester Regional Computer Centre, Internal Information, ref. 13 12 01 01, 16 August 1983.

[41] Ibbett, R.N., Edwards, D.A., Hopkins, T.P., Cadogan, C.K. and Train, D.A., "Centrenet - A High Performance Local Area Network", *Computer Journal*, vol. 28, no. 3, 1985, pp. 231-242.

[42] Ibbett, R.N., Capon, P.C. and Topham, N.P., "MU6V: A Parallel Vector Processing System", *Proc. 12th Int. Symp. on Computer Architecture*, June 17-19, 1985, pp. 136-144.

[43] ISO, **Connection Oriented Transport Protocol Specification**, ISO/DIS 8073, 1983.

[44] ISO, **Transport Service Definition**, ISO/DIS 8072, 1983.

[45] ISO, **Basic Connection Oriented Session Service Definition**, ISO/DIS 8326, 1983.

[46] ISO, **Basic Connection Oriented Session Protocol Specification**, ISO/DIS 8327, September 1983.

[47] Iverson, W.V., "CDC Couples CPU's in Fast Network", *Electronics*, 30 June 1981.

[48] Iyer, V. and Joshi, S.P., "Hardware Considerations in LANs", *Electronic Product Design*, October/November, 1983.

[49] Johnson, S.C., "Lint - A C Program Checker", Bell Laboratories, December 1977.

[50] Joshi, S. and Iyer, V., "New Standards for Local Networks push Upper Limits for Lightwave Data", *Data Communications*, July 1984.

[51] Joshi, S. and Iyer, V., "Protocols and Network-Control Chips : a Symbiotic Relationship", *Electronics*, 12 January, 1984.

[52] Joy, B., "Computer Workstation Architecture : 1982-1992", *IFIP Congress 1986*, Dublin, (H.-J. Kugler, ed.), pp. 1163-1168.

[53] Joy, W., "An Introduction to the C shell", Dept. of Electrical Engineering and Computer Science, University of California, Berkeley, Jan. 1986.

[54] **A Network Independent Job Transfer and Manipulation Protocol**, prepared by the JTP Working Party of the Data Communication Protocols Unit, Sept. 1981.

[55] Kernighan, B.W. and Ritchie, D.M., "The C Programming Language", Prentice Hall Software Series, London, 1978.

[56] King, F., Dewar, H., Hansen, I. and Thonnes, R., "The Edinburgh Advanced Personal Machine or LEGO Bricks and Computing", internal document, University of Edinburgh.

[57] Kravitz, J.K., Lieber, D., Robbins, F.H. and Palermo, J.M., "Workstations and Mainframe Computers Working Together", *IBM Systems Journal*, vol. 25, no. 1, 1986, pp. 116-128.

[58] Kung, H.T., "Special Purpose Supercomputers", *IFIP Congress 1986*, Dublin, (H.-J. Kugler, ed.), pp. 565-570.

[59] Lampson, B.W., Paul, M. and Siegert, H.J. (ed.s), **Distributed Systems - Architecture and Implementation**, Lecture Notes in Computer Science, vol. 105, 1981.

[60] Leach, P.J., Levine, P.H., Duros, B.P., Hamilton, J.A., Nelson, D.L. and Stumpf, B.L., "The Architecture of an Integrated Local Network", *IEEE Trans. on Comm.*, Local Area Networks Special, Nov. 1983.

[61] Lee, J. and Lee, T., "Microway Monoputer : Transputer Transplant", *Practical Computing*, vol. 10, issue 11, November 1987, pp. 65-66.

[62] Leffler, S.J., Fabry, R.S. and Joy, W.N., "4.2 BSD Networking Implementation Notes", Dept. of Electrical Engineering and Computer Science, University of California, Berkeley, 1983.

[63] Leffler, S.J., Joy, W.N. and Fabry, R.S., "4.2 BSD Networking Implementation Notes", Dept. of Electrical Engineering and Computer Science, University of California, Berkeley, 1986.

[64] Marsden, B.W., **Communication Network Protocols**, Chartwell-Bratt, 1985.

[65] Michel, M., "The Scholar's Workstation Project", personal notes from **Open Lectures in Computing : Campus Networks**, University of Kent at Canterbury, April 3-4, 1986.

[66] McCarthy, J., "Recursive Functions of Symbolic Expressions and their Computation by Machine, Part 1", *CACM*, 3, April 1960, pp. 184-195.

[67] McCarthy, J., Abrahams, P.W., Edwards, D.J., Hart, T.P. and Levin, M.I., **LISP 1.5 Programmer's Manual**, MIT Press, Cambridge, Mass., 1962.

[68] Metcalfe, R.M. and Boggs, D.R., "Ethernet: Distributed Packet Switching for Local Computer Networks", *CACM*, vol. 19, no. 7, July 1976, pp. 395-404.

[69] Moody, G., "The Mainframe Connection - Introduction", *Practical Computing*, vol. 10, issue 5, May 1987, pp. 87-88.

[70] Morris, J.H., Satyanarayanan, M., Conner, M.H., Howard, J.H., Rosenthal, D.S.H. and Smith, F.D., "Andrew: A Distributed Personal Computing Environment", *CACM*, vol. 29, no. 3, March 1986, pp. 184-201.

[71] **MU6-G Instruction Set Manual**, University of Manchester Computer Science Department, May 1982.

[72] Mukherjee, A., Kramer, J. and Magee, J., "A Distributed File Server for Embedded Applications", *IEE Software Engineering Journal*, vol. 3 no. 5, September 1988, pp. 142-148.

[73] Mullender, S.J. and Tanenbaum, A.S., "The Design of a Capability-Based Distributed Operating System", *Computer Journal*, vol. 29, no. 4, 1986, pp. 289-299.

[74] **MUSS User Manual** (Version 12), University of Manchester Computer Science Department, 1984.

[75] Naffah, N., "Workstations in the Next 10 Years", *IFIP Congress 1986*, Dublin, (H.-J. Kugler, ed.), pp. 1169-1170.

[76] Needham, R.M. and Herbert, A.J., **The Cambridge Distributed Operating System**, Addison-Wesley, Reading, Mass., 1982.

[77] Nelson, B.J., "Remote Procedure Call", Department of Computer Science, Carnegie-Mellon University tech. report no. CMU-CS-91-119, May 1981.

[78] Nelson, D., "Network Protocols for Apollo's DOMAIN System", **Local Networks, Strategy and Systems**, Proc. Localnet '83 (Europe), Online Publications Ltd., Northwood UK., 1983.

[79] Perry, A., "Some Aspects of the Design of the MU6 Operating System", M.Sc. thesis, University of Manchester, 1983.

[80] Phillips, S., "Distributed Systems and their Protocols", **Local Networks, Distributed Office and Factory Systems**, Proc. of Localnet '83, New York, London Online Inc., 1983, pp. 357-369.

[81] Popek, G.J. and Walker, B., **The LOCUS Distributed System Architecture**, the MIT press, Cambridge, Mass., 1985.

[82] Price, E., "SLIM - Serial Line Multiplexer", internal notes, University of Manchester, April 2, 1985.

[83] Randell, B.(ed.), **Network Protocols**, proc. joint IBM and University of Newcastle upon Tyne Seminar, 3-6 September 1985, University of Newcastle upon Tyne Computing Laboratory.

[84] Ritchie, D.M. and Thompson, K., "The UNIX Time-sharing System", *CACM*, vol. 17, no. 7, July 1974, pp. 365-375.

[85] Robinson, D.C. and Sloman, M.S., "Domain-Based Access Control for Distributed Computing Systems", *IEE Software Engineering Journal*, vol. 3 no. 5, September 1988, pp. 161-170.

[86] Rushby, J.M. and Randell, B., "A Distributed Secure System", Newcastle-upon-Tyne University, tech. report no. 182, May 1983.

[87] Schwartz, M., **Telecommunications Networks - Protocols, Modelling and Analysis**, Addison Wesley, Reading, Massachussetts, 1987.

[88] Schragl, R. and Lauber, D., "A Protocol for the Communication between Objects", *EUUG Spring 1988*, London, 13-15 April 1988.

[89] Scollo, G., Vissers, C.A. and Di Stefano, A., "LOTOS in Practice", *IFIP Congress 1986*, Dublin, (H.-J. Kugler, ed.), pp. 869-875.

[90] Spector, A.Z., "Performing Remote Operations Efficiently on a Local Computer Network", Computer Science Department, Stanford University, tech. report no. STAN-CS-80-831, Dec. 1980.

[91] Spector, A.Z. and Schwarz, P.M., "Transactions : A Construct for Reliable Distributed Computing", Carnegie-Mellon University, tech. report no. CMU-CS-82-143, 4 Jan. 1983.

[92] Stevens, J.S., "Design of a Front-End Processor for MU6", M.Sc. Thesis, University of Manchester, Oct. 1983.

[93] Stobie, I., "The Software Link", *Practical Computing*, vol. 10, issue 5, May 1987, pp. 94-97.

[94] Stobie, I., "How Bright is the Future", *Practical Computing*, vol. 10, issue 7, July 1987, pp. 81-84.

[95] Sunshine, C., "Formal Techniques for Protocol Specification and Verification", *IEEE Computer*, 12, 9, September 1979.

[96] Sun, G., internal report, University of Manchester, 1986.

[97] Swinehart, D., McDaniel, G., Boggs, D., "WFS : A Simple Stored File System for a Distributed Environment", XEROX Palo Alto Research Center, CSL-79-13, Oct. 1979; printed in *Operating Systems Review*, 13, 5, Nov. 1979.

[98] Tanenbaum, A.S., **Computer Networks**, Prentice/Hall International Inc., London, 1981.

[99] Tanenbaum, A.S. and Van Renesse, R., "Distributed Operating Systems", Vrije Universiteit, Amsterdam.

[100] **Texas Instruments Explorer Technical Summary**, Texas Instruments Incorporated, 1984.

[101] Thacker, C.P., McCreight, E.M., Lampson, B.W., Sproull, R.F. and Boggs, D.R., "Alto : A Personal Computer", **Computer Structures : Principles and Examples"**, (Siewiorek, D.P., Bell, C.G. and Newell, A.), McGraw-Hill, Tokyo, Japan, 1984, pp. 549-572.

[102] Thornton, J.E., "Back-End Network Approaches", *IEEE Computer*, 13, 2, Feb. 1980, pp. 10-17.

[103] Topham, N.P., "A Parallel Vector Processing System", Ph.D. thesis, University of Manchester, Sept. 1985.

[104] Topham, N.P., "Performance Analysis of a Data-Driven Multiple Vector Processing System", *Proc. IFIP Conference on Parallel Computers for Numerical and Signal Processing*, March 1986.

[105] VanHoulweling, D., "Workstations for all at Carnegie-Mellon", *Local Networks, Distributed Office and Factory Systems*, Proc. of Localnet '83, New York, London Online Inc., 1983, pp. 513-524.

[106] Vaughn, T.C., "Protocols and Services for a High Speed Network", M.Sc. thesis, University of Manchester, Sept. 1983.

[107] Waldron, M., "The System Comes out of the Network", *Computing*, 11 Sept. 1980, pp. 24, 25.

[108] Walker, B., Popek, G., English, R., Kline, C. and Thiel, G., "The LOCUS Distributed Operating System", *Proc. Ninth ACM Symposium on Operating System Principles*, 10-13 Oct. 1983; printed in *Operating Systems Review*, 17, 5, pp. 49-70.

[109] Wecker, S., "DNA : The Digital Network Architecture", *IEEE Trans. on Comm.*, vol. COM-28, 4, April 1980, pp. 510-526.

[110] Windsor II, W.A., "IEEE Floating Point Chips implement DSP Architecture", Computer Design, Jan. 1985, vol. 24, no. 1, pp. 165-170.

[111] Zwaenepoel, W., "Implementation and Performance of Pipes in the V-System", *IEEE Trans. on Computers*, vol. C-34, no. 12, December 1985, pp. 1174-1178.

Appendix A

The Inter-User Communication Protocol

A.1 Table of Primitives

Primitive	From	To	Purpose
MAKE_CALL	*Cr*	*Ce*	attempt to establish communication channel
ANSWER_CALL	*Ce*	*Cr*	called node is willing and able to accept call
END_CALL	either	either	conclude interaction
CALL_COMPLETE	either	either	confirm end of interaction
ABORT_CALL	either	either	abandon call
SUPPORT_CALL	*Cr*	*Ce*	communication requires extra features
ROUTE_CALL	either	*Ce*	network address of called node is unknown
CALL_FAILED	*Ce*	*Cr*	called node is unable or unwilling to accept call
SUSPEND_CALL	either	either	temporary interruption of call
RESUME_CALL	either	either	continue the call following the temporary break
CALL_CHARGE	*Ce*	*Cr*	accounting information

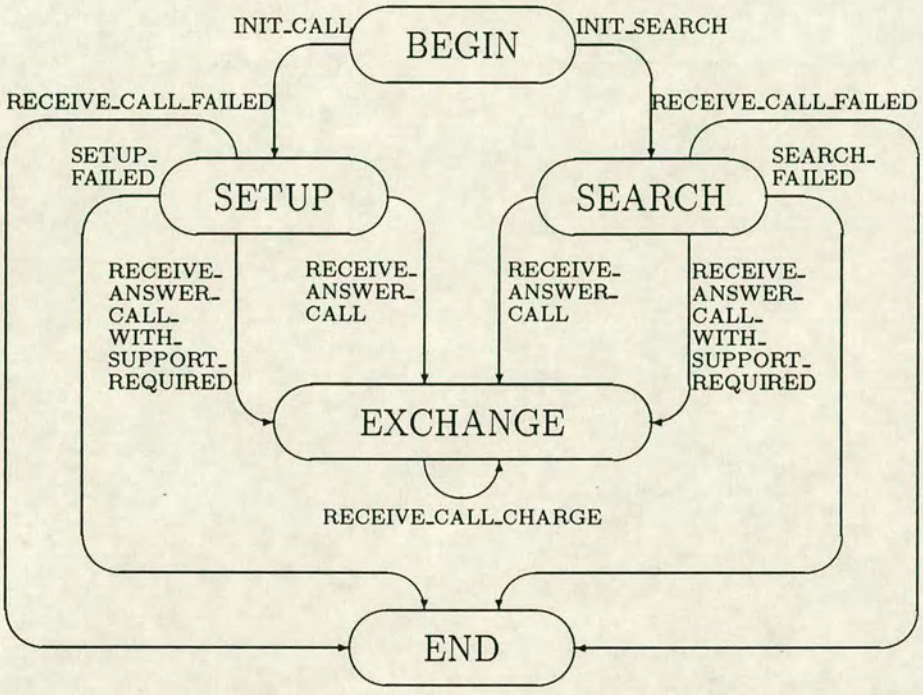

Figure A–1: User protocol : Caller (Cr)

A.2 Protocol Definition

A.2.1 Caller (Cr)

Description of States

The **Caller** (*Cr*) has 8 possible states :

1. BEGIN: *Cr* has not yet initiated a call.

2. SETUP: *Cr* attempts to establish a connection with a *Ce*.

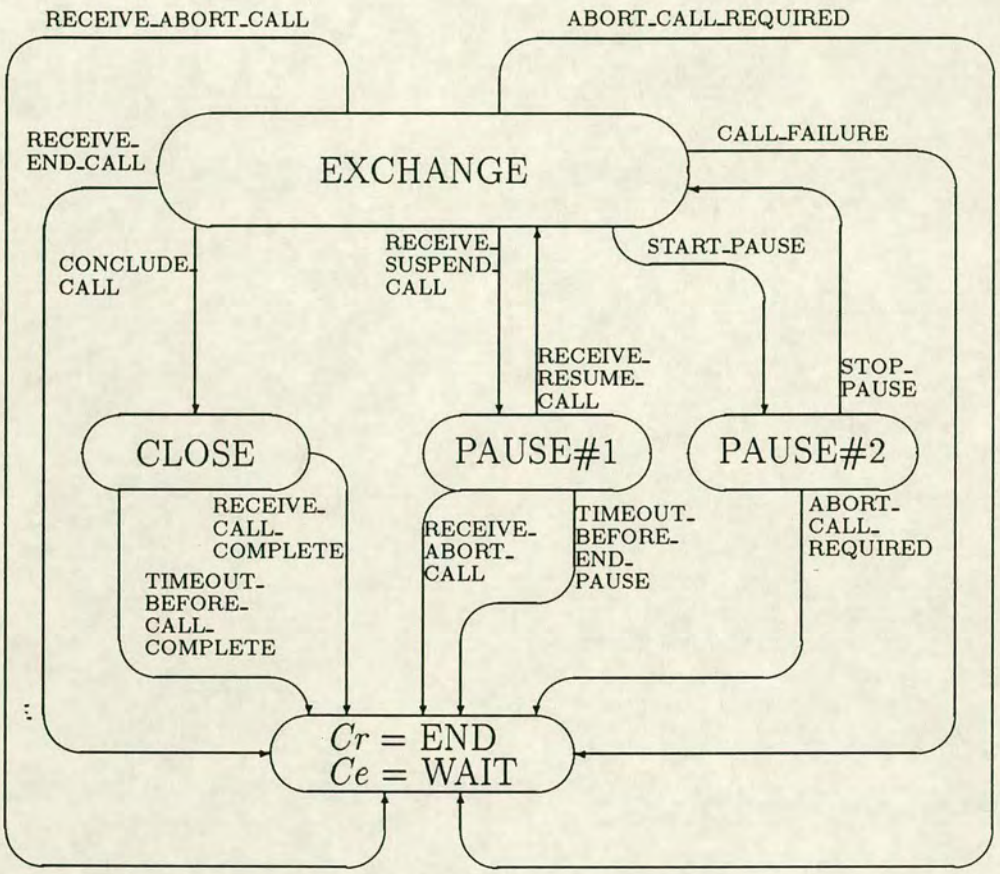

Figure A–2: User protocol : Common

3. SEARCH: *Cr* is attempting to establish a call, but does not have the network address for *Ce*.

4. EXCHANGE: *Cr* and *Ce* are actively communicating.

5. CLOSE: *Cr* concludes the call.

6. PAUSE#1: the call has been suspended at the request of *Ce*.

7. PAUSE#2: the call has been suspended at the request of *Cr*.

8. END: the call has been completed.

Description of Events

The **Caller** (*Cr*) identifies the following events :

1. INIT_CALL: the *Cr* process has just been created, and the network address of *Ce* is known.

2. INIT_SEARCH: the *Cr* process has just been created, and the network address of *Ce* is unknown.

3. RECEIVE_ANSWER_CALL: *Cr* receives **ANSWER_CALL** from *Ce*, and the call requires no special facilities.

4. RECEIVE_ANSWER_CALL_WITH_SUPPORT_REQUIRED: *Cr* receives **ANSWER_CALL** from *Ce*, but the call has special needs.

5. RECEIVE_CALL_FAILED: *Cr* receives **CALL_FAILED** from *Ce*.

6. SETUP_FAILED: *Cr* fails to receive either **ANSWER_CALL** or **CALL_FAILED**, when trying to establish a call with a *Ce* with a known address, within the timeout period (T_{user1}).

7. SEARCH_FAILED: *Cr* has received neither **ANSWER_CALL** nor **CALL_FAILED**, when searching for a *Ce* with an unknown network address, within the timeout period (T_{user2}).

8. RECEIVE_CALL_CHARGE: *Cr* receives **CALL_CHARGE** from *Ce*, to indicate accounting information.

9. CALL_FAILURE: *Cr* fails to communicate with *Ce* within the timeout period (T_{user3}).

10. RECEIVE_END_CALL: *Cr* receives **END_CALL** from *Ce*, indicating that *Ce* wishes to conclude the call.

11. CONCLUDE_CALL: *Cr* requires that the call be concluded normally.

12. ABORT_CALL_REQUIRED: *Cr* requires that the call be aborted.

13. RECEIVE_ABORT_CALL: *Cr* receives **ABORT_CALL** from *Ce*.

14. START_PAUSE: *Cr* requires that the call be suspended temporarily.

15. RECEIVE_SUSPEND_CALL: *Cr* receives **SUSPEND_CALL** from *Ce*, indicating that *Ce* wishes to suspend the call temporarily.

16. RECEIVE_CALL_COMPLETE: *Cr* receives **CALL_COMPLETE** from *Ce*, indicating that the call has been concluded normally.

17. TIMEOUT_BEFORE_CALL_COMPLETE: *Cr* fails to receive **CALL_COMPLETE** within the timeout period (T_{user4}).

18. RECEIVE_RESUME_CALL: *Cr* receives **RESUME_CALL** from *Ce*, indicating that *Ce* wishes to continue the call.

19. TIMEOUT_BEFORE_END_PAUSE:

    Cr fails to receive either **RESUME_CALL** or **ABORT_CALL** within the timeout period (T_{user5}).

20. STOP_PAUSE: *Cr* requires that the suspended call be resumed.

Description of Actions

The **Caller** (*Cr*) may initiate the following actions :

1. ESTABLISH_CALL: *Cr* sends **MAKE_CALL** to the *Ce*.

2. LOOK_FOR_Ce: *Cr* sends **ROUTE_CALL** to a *Ce* that may know the network address of the intended *Ce*.

3. BEGIN_EXCHANGE: *Cr* starts communication with *Ce* using lower level primitives.

4. SEND_SUPPORT_INFO: *Cr* sends **SUPPORT_CALL** to the *Ce*, indicating details of the extra facilities required for the call.

5. ABANDON_CALL: *Cr* discontinues communication with *Ce*, and the process terminates.

6. REGISTER_CHARGE: *Cr* records the accounting information supplied by *Ce*.

7. CONTINUE_EXCHANGE: *Cr* continues the communication with the *Ce*.

8. FINISH_CALL: *Cr* sends **CALL_COMPLETE** to the *Ce*.

9. COMPLETE_CALL_NORMALLY: *Cr* completes the call and the process terminates.

10. REQUEST_CALL_COMPLETION: *Cr* sends **END_CALL** to the *Ce*, to initiate normal completion of the call.

11. REQUEST_CALL_ABORT: *Cr* sends **ABORT_CALL** to the *Ce*, to terminate the call abnormally.

12. REQUEST_CALL_SUSPENSION: *Cr* sends **SUSPEND_CALL** to the *Ce*, to suspend communication with the *Ce* temporarily.

13. SUSPEND_EXCHANGE: *Cr* temporarily suspends the call.

14. REQUEST_CALL_RESUME: *Cr* sends **RESUME_CALL** to the *Ce*, to resume the communication with the *Ce*.

State Transition Tables

This State	Event	Action	Next State
BEGIN	INIT_CALL	ESTABLISH_CALL	SETUP
	INIT_SEARCH	LOOK_FOR_Ce	SEARCH
SETUP	RECEIVE_ANSWER_CALL	BEGIN_EXCHANGE	EXCHANGE
	RECEIVE_ANSWER_CALL_WITH_SUPPORT_REQUIRED	SEND_SUPPORT_INFO; BEGIN_EXCHANGE	EXCHANGE
	RECEIVE_CALL_FAILED	ABANDON_CALL	END
	SETUP_FAILED	ABANDON_CALL	END
SEARCH	RECEIVE_ANSWER_CALL	BEGIN_EXCHANGE	EXCHANGE
	RECEIVE_ANSWER_CALL_WITH_SUPPORT_REQUIRED	SEND_SUPPORT_INFO; BEGIN_EXCHANGE	EXCHANGE
	RECEIVE_CALL_FAILED	ABANDON_CALL	END
	SEARCH_FAILED	ABANDON_CALL	END

This State	Event	Action	Next State
EXCHANGE	RECEIVE_CALL_CHARGE	REGISTER_CHARGE; CONTINUE_EXCHANGE	EXCHANGE
	RECEIVE_END_CALL	FINISH_CALL; COMPLETE_CALL_NORMALLY	END
	CONCLUDE_CALL	REQUEST_CALL_COMPLETION	CLOSE
	ABORT_CALL_REQUIRED	REQUEST_CALL_ABORT; ABANDON_CALL	END
	RECEIVE_ABORT_CALL	ABANDON_CALL	END
	START_PAUSE	REQUEST_CALL_SUSPENSION; SUSPEND_EXCHANGE	PAUSE#2
	RECEIVE_SUSPEND_CALL	SUSPEND_EXCHANGE	PAUSE#1
	CALL_FAILURE	REQUEST_CALL_ABORT; ABANDON_CALL	END

This State	Event	Action	Next State
CLOSE	RECEIVE_CALL_COMPLETE	COMPLETE_CALL_NORMALLY	END
	TIMEOUT_BEFORE_CALL_COMPLETE	ABANDON_CALL	END
PAUSE#1	RECEIVE_RESUME_CALL	CONTINUE_EXCHANGE	EXCHANGE
	RECEIVE_ABORT_CALL	ABANDON_CALL	END
	TIMEOUT_BEFORE_END_PAUSE	ABANDON_CALL	END
PAUSE#2	STOP_PAUSE	REQUEST_CALL_RESUME; CONTINUE_EXCHANGE	EXCHANGE
	ABORT_CALL_REQUIRED	REQUEST_CALL_ABORT; ABANDON_CALL	

A.2.2 Callee (Ce)

Description of States

The **Callee** (*Ce*) has 8 possible states :

1. WAIT: *Ce* is inactive, waiting for a call to be initiated by a *Cr*.

2. SETUP: *Ce* is establishing a call with a *Cr*, but the call requires extra facilities.

3. CLOSE: *Ce* concludes the call.

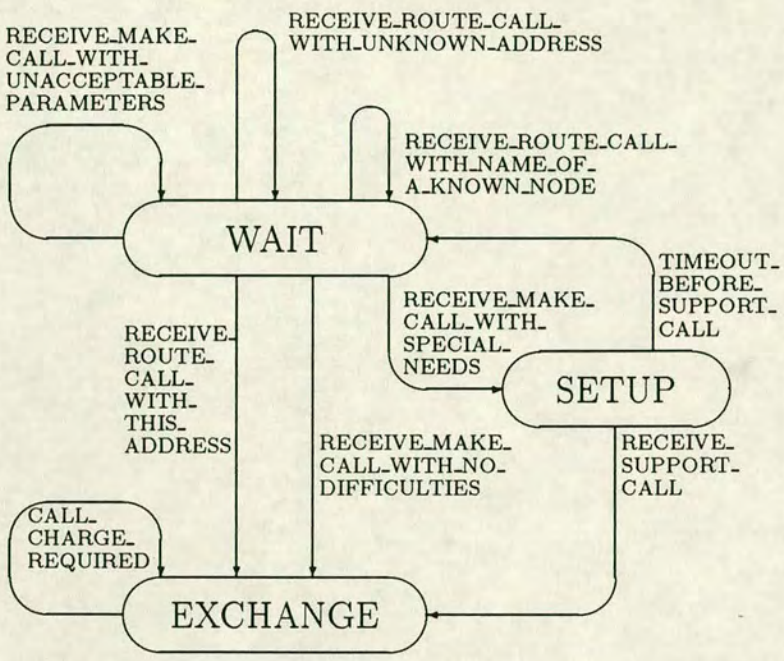

Figure A–3: User protocol : Callee (Ce)

4. EXCHANGE: *Ce* and the *Cr* are actively communicating.

5. PAUSE#1: the call has been suspended at the request of *Cr*.

6. PAUSE#2: the call has been suspended at the request of *Ce*.

Description of Events

The **Callee** (*Ce*) identifies the following events :

1. RECEIVE_MAKE_CALL_WITH_NO_DIFFICULTIES:

   Ce receives **MAKE_CALL** from a *Cr* in order to establish the call. *Ce* is willing and able to accept the call; the call requires no special features.

2. RECEIVE_ROUTE_CALL_WITH_THIS_ADDRESS: *Ce* receives **ROUTE_CALL** from a *Cr*, or an intermediary *Ce*, indicating that a *Cr* wishes to establish a call but does not know the network address of the *Ce*. The specified name is that of this *Ce*.

3. RECEIVE_MAKE_CALL_WITH_SPECIAL_NEEDS: *Ce* receives **MAKE_CALL** from a *Cr*. *Ce* is willing to accept the call, but the call requires special features.

4. RECEIVE_MAKE_CALL_WITH_UNACCEPTABLE_PARAMETERS: *Ce* receives **MAKE_CALL** from a *Cr*, but is unwilling or unable to accept the call.

5. RECEIVE_ROUTE_CALL_WITH_NAME_OF_A_KNOWN_NODE: *Ce* receives **ROUTE_CALL** from a *Cr*, or an intermediary *Ce*, and this *Ce* can route the request towards the intended *Ce*.

6. RECEIVE_ROUTE_CALL_WITH_UNKNOWN_ADDRESS: *Ce* receives **ROUTE_CALL** from a *Cr*, or an intermediary *Ce*, but cannot route the request towards the intended *Ce*.

7. RECEIVE_SUPPORT_CALL: *Ce* receives **SUPPORT_CALL** from the *Cr*, giving details of how the extra facilities required by the call may be accessed or provided.

8. TIMEOUT_BEFORE_SUPPORT_CALL: *Ce* fails to receive **SUPPORT_CALL** from the *Cr* before the end of the timeout period (T_{user6}).

9. CONCLUDE_CALL: *Ce* requires that the call be concluded normally.

10. RECEIVE_END_CALL: *Ce* receives **END_CALL** from the *Cr*, indicating that *Cr* wishes to conclude the call.

11. ABORT_CALL_REQUIRED: *Ce* requires that the call be concluded abnormally.

12. RECEIVE_ABORT_CALL: *Ce* receives **ABORT_CALL** from the *Cr*.

13. CALL_CHARGE_REQUIRED: *Ce* requires that accounting information be passed to the *Cr*.

14. CALL_FAILURE: *Ce* fails to communicate with *Cr* within the timeout period (T_{user7}).

15. START_PAUSE: *Ce* requires that the call be suspended temporarily.

16. RECEIVE_SUSPEND_CALL: *Ce* receives **SUSPEND_CALL** from *Cr*, indicating that the *Cr* wishes to suspend the call temporarily.

17. RECEIVE_CALL_COMPLETE: *Ce* receives **CALL_COMPLETE** from the *Cr*, indicating that the call as been concluded normally.

18. TIMEOUT_BEFORE_CALL_COMPLETE: *Cr* fails to receive **CALL_COMPLETE** within the timeout period (T_{user8}).

19. RECEIVE_RESUME_CALL: *Ce* receives **RESUME_CALL** from the *Cr*, indicating that *Cr* wishes to continue the call.

20. STOP_PAUSE: *Ce* requires that the suspended call be resumed.

21. TIMEOUT_BEFORE_END_PAUSE:

    Cr fails to receive either **RESUME_CALL** or **ABORT_CALL** within the timeout period (T_{user9}).

Description of Actions

The **Callee** (*Ce*) identifies the following events :

1. PREPARE_FOR_CALL: *Ce* sets up communication channels for the call from *Cr*.

2. BEGIN_EXCHANGE: *Ce* starts communicating with *Cr* using lower level primitives.

3. INDICATE_CALL_FAILURE: *Ce* sends **CALL_FAILED** to *Cr*, indicating that it is unable or unwilling to satisfy the request.

4. ROUTE_CALL_TO_REQUIRED_NODE: *Ce* forwards **ROUTE_CALL** towards the intended *Ce*.

5. ABANDON_CALL: *Ce* discontinues communication with the *Cr*, and the call is abandoned.

6. REQUEST_CALL_COMPLETION: *Ce* sends **END_CALL** to the *Cr*, to initiate normal completion of the call.

7. FINISH_CALL: *Ce* sends **CALL_COMPLETE** to the *Cr*.

8. COMPLETE_CALL_NORMALLY: *Ce* completes the call and closes down the communication channels.

9. REQUEST_CALL_ABORT: *Ce* sends **ABORT_CALL** to the *Cr*, to abnormally terminate the call.

10. MAKE_CALL_CHARGE: *Ce* sends **CALL_CHARGE** to the *Cr*, supplying accounting information.

11. REQUEST_CALL_SUSPENSION: *Ce* sends **SUSPEND_CALL**, to suspend communication with the *Cr* temporarily.

12. SUSPEND_EXCHANGE: *Ce* temporarily suspends the call.

13. CONTINUE_EXCHANGE: *Ce* continues the communication with the *Cr*.

14. REQUEST_CALL_RESUME: *Ce* sends **RESUME_CALL** to the *Cr*, to resume the communication with the *Cr*.

State Transition Tables

This State	Event	Action	Next State
WAIT	RECEIVE_MAKE_CALL_WITH_NO_DIFFICULTIES	PREPARE_FOR_CALL; BEGIN_EXCHANGE	EXCHANGE
	RECEIVE_ROUTE_CALL_WITH_THIS_ADDRESS	PREPARE_FOR_CALL; BEGIN_EXCHANGE	EXCHANGE
	RECEIVE_MAKE_CALL_WITH_SPECIAL_NEEDS	PREPARE_FOR_CALL	SETUP
	RECEIVE_MAKE_CALL_WITH_UNACCEPTABLE_PARAMETERS	INDICATE_CALL_FAILURE	WAIT
	RECEIVE_ROUTE_CALL_WITH_NAME_OF_A_KNOWN_NODE	ROUTE_CALL_TO_REQUIRED_NODE	WAIT
	RECEIVE_ROUTE_CALL_WITH_UNKNOWN_ADDRESS	INDICATE_CALL_FAILURE	WAIT
SETUP	RECEIVE_SUPPORT_CALL	BEGIN_EXCHANGE	EXCHANGE
	TIMEOUT_BEFORE_SUPPORT_CALL	ABANDON_CALL	WAIT

This State	Event	Action	Next State
EXCHANGE	CONCLUDE_CALL	REQUEST_CALL_ COMPLETION	CLOSE
	RECEIVE_ END_CALL	FINISH_CALL; COMPLETE_CALL_ NORMALLY	WAIT
	ABORT_CALL_ REQUIRED	REQUEST_CALL_ ABORT; ABANDON_CALL	WAIT
	RECEIVE_ ABORT_CALL	ABANDON_CALL	WAIT
	CALL_CHARGE_ REQUIRED	MAKE_CALL_ CHARGE	EXCHANGE
	START_PAUSE	REQUEST_CALL_ SUSPENSION; SUSPEND_EXCHANGE	PAUSE#2
	RECEIVE_ SUSPEND_CALL	SUSPEND_EXCHANGE	PAUSE#1
	CALL_ FAILURE	REQUEST_ CALL_ABORT; ABANDON_CALL	END

This State	Event	Action	Next State
CLOSE	RECEIVE_CALL_ COMPLETE	COMPLETE_CALL_ NORMALLY	WAIT
	TIMEOUT_BEFORE_ CALL_COMPLETE	ABANDON_CALL	END
PAUSE#1	RECEIVE_RESUME_ CALL	CONTINUE_EXCHANGE	EXCHANGE
	RECEIVE_ ABORT_ CALL	ABANDON_CALL	WAIT
	TIMEOUT_BEFORE_ END_PAUSE	ABANDON_CALL	END
PAUSE#2	STOP_PAUSE	REQUEST_CALL_ RESUME; CONTINUE_EXCHANGE	EXCHANGE
	ABORT_CALL_ REQUIRED	REQUEST_CALL_ ABORT; ABANDON_CALL	WAIT

Appendix B

The Basic Service Provision Protocol

B.1 Tables of Primitives

Primitive	From	To	Purpose
REQUEST	*S-R*	*S-P*	request network service
REPLY	*S-P*	*S-R*	response to network service request
REDIRECT	*S-M*	*S-R*	re-route request to another node
DIRECT	*S-M*	*S-P*	prepare for redirected request
CONFIRM	*S-P*	*S-M*	redirected request completed
REDIR_M	*S-P*	*S-M*	redirected service request has been re-routed to another node
ENQ_PROVISION	*S-M*	*S-P*	poll for state of service provider; requester not receiving response
ENQ_SERVICE	*S-R*	*S-M*	poll for state of service provision; provider has not replied, and has not responded to enquiry
ENQ_SUPPLY	*S-R*	*S-P*	poll for state of service supply; provider has not replied
PROVISION_FAIL	*S-P*	*S-M*	service provision has failed; provider failed in its provision of service

SERVICE_FAIL	*S-R*	*S-M*	service provision has failed; requester bids for recovery of service
SUPPLY_FAIL	*S-P*	*S-R*	service provision has failed; provider failed in its provision of service
NOTIFY_S_FAIL	*S-M*	*S-R*	service provision has failed; no recovery possible
CONTINUE	*S-M*	*S-R*	no apparent problem with *S-P*'s provision of the service
PROV_ACTIVE	*S-P*	*S-M*	provider is active in its provision of the service
EXTEND_TIME	*S-P*	*S-R*	provider requires more time in which to complete the service
REQ_MORE_DATA	*S-P*	*S-R*	provider requires more data for the service provision
GIVE_MORE_DATA	*S-R*	*S-P*	requester supplies extra data for the service
ABORT_REQUEST	*S-R*	*S-P*	request for service rescinded
ABORT_PROVISION	*S-M*	*S-P*	abandon provision of service
REJECT_VALIDITY	any	any	received message has invalid context or permit

B.2 Protocol Definition

B.2.1 Service Requester (S-R)

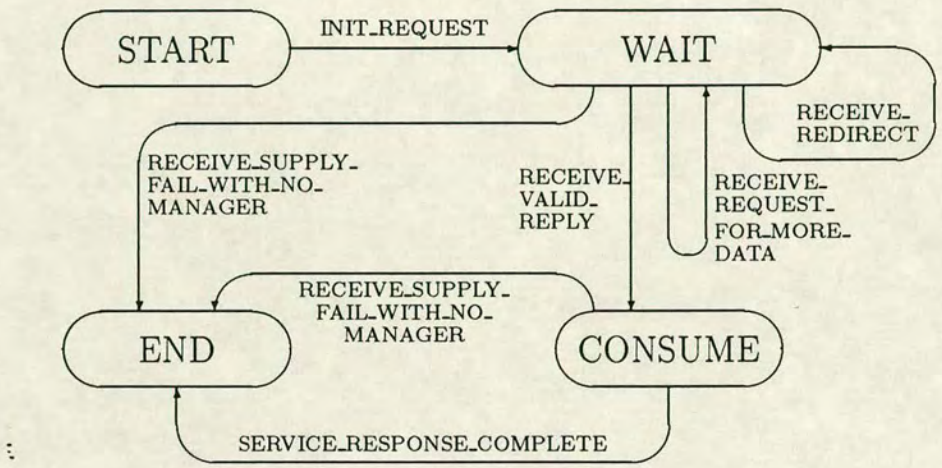

Figure B–1: Basic Protocol : Service Requester (S-R) - 'WAIT'

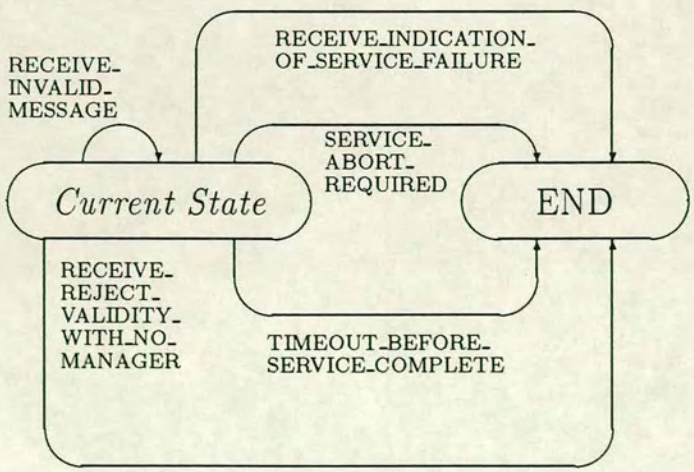

Figure B–2: Basic Protocol : Service Requester (S-R) - All States

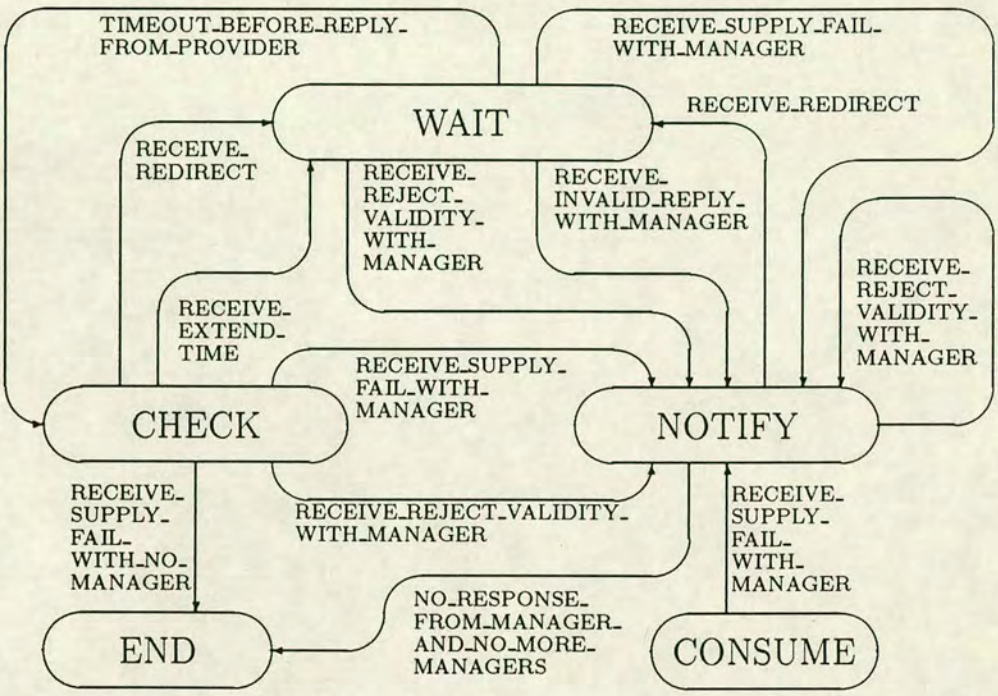

Figure B–3: Basic Protocol : Service Requester (S-R) - 'CHECK' + 'NOTIFY'

Description of States

The **Service Requester** (*S-R*) has 7 possible states :

1. START: *S-R* is not enabled - the process has not been created.

2. WAIT: *S-R* has submitted a request for a service and is waiting for the reply.

3. CONSUME: the reply has begun, and the data is being received.

4. END: the service request is concluding, and the process terminates.

5. CHECK: *S-R* is checking the state of the *S-P* because no reply has been received.

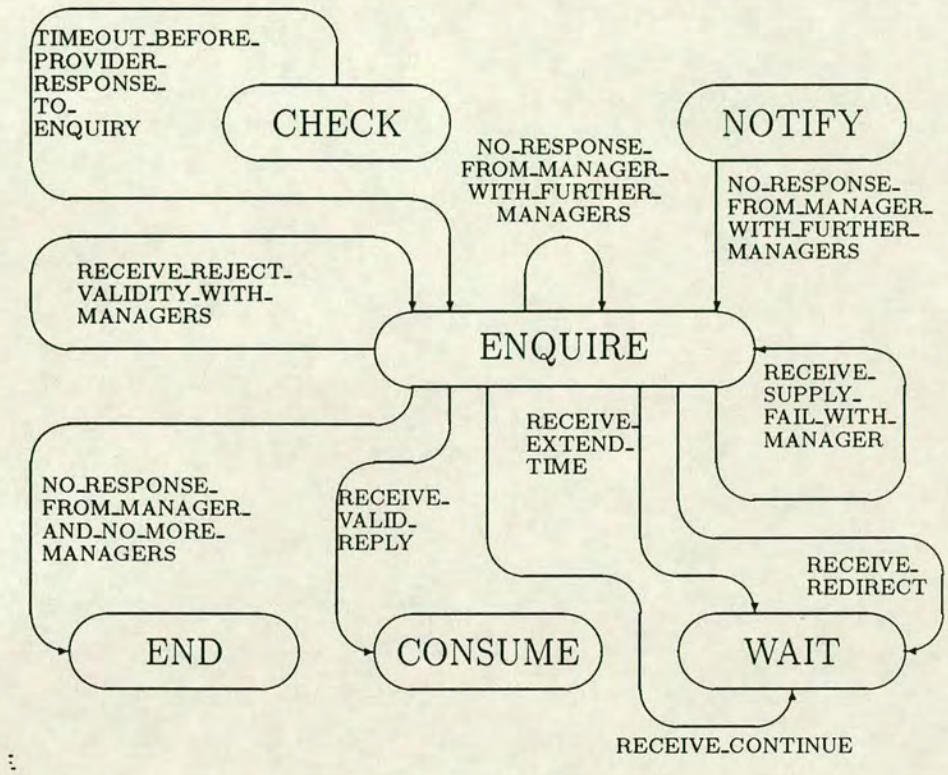

Figure B–4: Basic Protocol : Service Requester (S-R) - 'ENQUIRE'

6. NOTIFY: *S-P* has indicated to *S-R* that its supply of the required service has failed. So, *S-R* is notifying the *S-M* of the problem, with the objective of being re-routed to a new *S-P*.

7. ENQUIRE: *S-R* is enquiring about the state of the service to *S-M*. The nature of the problem is unknown, but the original request has not been satisfied. Where the current *S-M* has failed to respond to a failure notification or service enquiry, *S-R* has sent a further enquiry to an *S-M* higher in the chain.

Description of Events

The **Service Requester** (*S-R*) identifies the following events :

1. INIT_REQUEST: the *S-R* process has just been created.

2. RECEIVE_VALID_REPLY: *S-R* receives **REPLY** from the *S-P* with the correct permit and context.

3. RECEIVE_REDIRECT: *S-R* receives **REDIRECT** from *S-M* because the *S-P* is incapable of providing the required service, but is aware of another node that is capable.

   ⋮

4. RECEIVE_REQUEST_FOR_MORE_DATA: *S-R* receives **REQ_MORE_DATA** from *S-P*, indicating that insufficient information was provided in the original request.

5. RECEIVE_SUPPLY_FAIL_WITH_NO_MANAGER: *S-R* receives **SUPPLY_FAIL** from *S-P*, indicating that the provision of the service has failed, and there is no current *S-M* to recover the service.

6. SERVICE_RESPONSE_COMPLETE: *S-P* has returned all of the results in the **REPLY**, and the provision of the service is complete.

7. TIMEOUT_BEFORE_SERVICE_COMPLETE: the provision of the service has not been completed within the timeout period (T_{basic1}).

8. RECEIVE_REJECT_VALIDITY_WITH_NO_MANAGER: *S-R* receives **REJECT_VALIDITY** in response to its last message, indicating that the permit or context is wrong, but there is no current manager.

9. RECEIVE_INDICATION_OF_SERVICE_FAILURE: *S-R* receives **NOTIFY_S_FAIL** from *S-M*, indicating that the service request hasd failed and should be abandoned.

10. SERVICE_ABORT_REQUIRED: *S-R* is required to abort the request for the service.

11. RECEIVE_INVALID_MESSAGE: *S-R* receives a message with an invalid permit or incorrect context.

12. TIMEOUT_BEFORE_REPLY_FROM_PROVIDER: *S-P* has failed to respond to the initial service request within the timeout period (T_{basic2}).

13. RECEIVE_INVALID_REPLY_WITH_MANAGER: *S-R* receives **REPLY** from *S-P*, but the permit is invalid or the context is incorrect, and there is a current manager.

14. RECEIVE_REJECT_VALIDITY_WITH_MANAGER: *S-R* receives **REJECT_VALIDITY** in response to its last message, indicating that the permit or context is wrong, and there is a current *S-M*.

15. RECEIVE_SUPPLY_FAIL_WITH_MANAGER: *S-R* receives **SUPPLY_FAIL** from *S-P*, indicating that provision of the service has failed, and there is a current *S-M* to recover the service.

16. RECEIVE_EXTEND_TIME: *S-R* receives **EXTEND_TIME** from *S-P*, indicating that provision of the service is active, but extra time is required for its completion.

17. NO_RESPONSE_FROM_MANAGER_AND_NO_MORE_MANAGERS: *S-M* has failed to respond within the timeout period (T_{basic3}), and there are no more managers higher in the chain.

18. TIMEOUT_BEFORE_PROVIDER_RESPONSE_TO_ENQUIRY: *S-P* has failed to respond to the service supply enquiry within the timeout period (T_{basic4}).

19. NO_RESPONSE_FROM_MANAGER_WITH_FURTHER_MANAGERS: *S-M* has failed to respond within the timeout period (T_{basic3}), but there are more managers higher in the chain.

20. RECEIVE_CONTINUE: *S-R* receives **CONTINUE** from *S-M*, indicating that it should proceed with the service request since the *S-P* appears to be active.

Description of Actions

The **Service Requester** (*S-R*) may initiate the following actions :

1. REJECT_INVALID_MESSAGE: *S-R* returns **REJECT_VALIDITY** to the node that sent the invalid message.

2. TERMINATE_REQUEST_ABNORMALLY: *S-R* sends **ABORT_REQUEST** to *S-P* and the process terminates with an error code.

3. ABANDON_REQUEST: *S-R* process terminates with an error code.

4. MAKE_REQUEST: *S-R* sends **REQUEST** to the nominated *S-P*, recording it as the 'current *S-P*'.

5. RECORD_REDIRECT: *S-R* makes the responding node the 'current *S-M*', and updates the 'current *S-P*', changing the permit and context accordingly.

6. BEGIN_CONSUMING_DATA: *S-R* starts to pass the contents of the reply through to the application.

7. PROVIDE_DATA: *S-R* sends **GIVE_MORE_DATA** to *S-P* to forward input data from the application.

8. RECORD_PROVIDER_FAIL: *S-R* marks the current *S-P* as having failed in its supply of the service.

9. NOTIFY_MANAGER_OF_SERVICE_FAILURE: *S-R* sends **SERVICE_FAIL** to *S-M*, with details of how the failure was determined.

10. CHECK_STATUS_OF_PROVIDER: *S-R* sends **ENQ_SUPPLY** to *S-P*.

11. CONCLUDE_REQUEST_NORMALLY: *S-R* completes the return of the output data to the application and the process terminates.

12. ADJUST_TIMEOUT: *S-R* updates the timeout (T_{basic2}) to reflect the extra processing time required by *S-P*.

13. MAKE_SERVICE_ENQUIRY_TO_MANAGER: *S-R* sends **ENQ_SERVICE** to *S-M*.

14. MARK_CURRENT_MANAGER_AS_INVALID: *S-R* marks the 'current *S-M*' as having failed to recover the service.

15. REINSTATE_OLD_MANAGER: *S-R* takes the previous *S-M* and makes it the 'current *S-M*'.

16. RESET_TIMEOUT: *S-R* resets the timeout (T_{basic2}), as directed by *S-M*.

State Transition Tables

This State	Event	Action	Next State
START	RECEIVE_INVALID MESSAGE	REJECT_INVALID MESSAGE	START
	TIMEOUT_BEFORE_ SERVICE_COMPLETE	TERMINATE_ REQUEST_ ABNORMALLY	END
	RECEIVE_REJECT_ VALIDITY_WITH_ NO_MANAGER	ABANDON_ REQUEST	END
	RECEIVE_ INDICATION_ OF_SERVICE_ FAILURE	ABANDON_ REQUEST	END
	SERVICE_ABORT_ REQUIRED	TERMINATE_ REQUEST_ ABNORMALLY	END
	INIT_REQUEST	MAKE_REQUEST	WAIT

This State	Event	Action	Next State
WAIT	RECEIVE_INVALID MESSAGE	REJECT_INVALID MESSAGE	WAIT
	TIMEOUT_BEFORE_ SERVICE_COMPLETE	TERMINATE_ REQUEST_ ABNORMALLY	END
	RECEIVE_REJECT_ VALIDITY_WITH_ NO_MANAGER	ABANDON_ REQUEST	END
	RECEIVE_ INDICATION_ OF_SERVICE_ FAILURE	ABANDON_ REQUEST	END
	SERVICE_ABORT_ REQUIRED	TERMINATE_ REQUEST_ ABNORMALLY	END
	RECEIVE_REDIRECT	RECORD_REDIRECT; MAKE_REQUEST	WAIT
	RECEIVE_VALID_ REPLY	BEGIN_CONSUMING_ REPLY	CONSUME
	RECEIVE_REQUEST_ FOR_MORE_DATA	PROVIDE_DATA	WAIT
	RECEIVE_SUPPLY_ FAIL_WITH_NO_ MANAGER	ABANDON_ REQUEST	END

This State	Event	Action	Next State
WAIT	RECEIVE_SUPPLY_FAIL_WITH_MANAGER	RECORD_PROVIDER_FAIL; NOTIFY_MANAGER_OF_SERVICE_FAILURE	NOTIFY
	RECEIVE_REJECT_VALIDITY_WITH_MANAGER	RECORD_PROVIDER_FAIL; NOTIFY_MANAGER_OF_SERVICE_FAILURE	NOTIFY
	RECEIVE_INVALID_REPLY_WITH_MANAGER	RECORD_PROVIDER_FAIL; NOTIFY_MANAGER_OF_SERVICE_FAILURE	NOTIFY
	TIMEOUT_BEFORE_REPLY_FROM_PROVIDER	CHECK_STATUS_OF_PROVIDER	CHECK

This State	Event	Action	Next State
CONSUME	RECEIVE_INVALID MESSAGE	REJECT_INVALID MESSAGE	CONSUME
	TIMEOUT_BEFORE_ SERVICE_COMPLETE	TERMINATE_ REQUEST_ ABNORMALLY	END
	RECEIVE_REJECT_ VALIDITY_WITH_ NO_MANAGER	ABANDON_ REQUEST	END
	RECEIVE_ INDICATION_ OF_SERVICE_ FAILURE	ABANDON_ REQUEST	END
	SERVICE_ABORT_ REQUIRED	TERMINATE_ REQUEST_ ABNORMALLY	END
	SERVICE_RESPONSE_ COMPLETE	CONCLUDE_ REQUEST_ NORMALLY	END
	RECEIVE_SUPPLY_ FAIL_WITH_NO_ MANAGER	ABANDON_ REQUEST	END
	RECEIVE_SUPPLY_ FAIL_WITH_ MANAGER	RECORD_PROVIDER_ FAIL; NOTIFY_ MANAGER_OF_ SERVICE_FAILURE	NOTIFY

This State	Event	Action	Next State
CHECK	RECEIVE_INVALID MESSAGE	REJECT_INVALID MESSAGE	CHECK
	TIMEOUT_BEFORE_ SERVICE_COMPLETE	TERMINATE_ REQUEST_ ABNORMALLY	END
	RECEIVE_REJECT_ VALIDITY_WITH_ NO_MANAGER	ABANDON_ REQUEST	END
	RECEIVE_ INDICATION_ OF_SERVICE_ FAILURE	ABANDON_ REQUEST	END
	SERVICE_ABORT_ REQUIRED	TERMINATE_ REQUEST_ ABNORMALLY	END
	RECEIVE_ EXTEND_TIME	ADJUST_TIMEOUT	WAIT
	RECEIVE_ REDIRECT	RECORD_REDIRECT; MAKE_REQUEST	WAIT
	TIMEOUT_BEFORE_ PROVIDER_RESPONSE_ TO_ENQUIRY	MAKE_SERVICE_ ENQUIRY_TO_ MANAGER	ENQUIRE
	RECEIVE_REJECT_ VALIDITY_WITH_ MANAGER	RECORD_PROVIDER_ FAIL; NOTIFY_ MANAGER_OF_ SERVICE_FAILURE	NOTIFY

This State	Event	Action	Next State
CHECK	RECEIVE_SUPPLY_FAIL_WITH_MANAGER	RECORD_PROVIDER_FAIL; NOTIFY_MANAGER_OF_SERVICE_FAILURE	NOTIFY
	RECEIVE_SUPPLY_FAIL_WITH_NO_MANAGER	ABANDON_REQUEST	END

This State	Event	Action	Next State
NOTIFY	RECEIVE_INVALID MESSAGE	REJECT_INVALID MESSAGE	NOTIFY
	TIMEOUT_BEFORE_ SERVICE_COMPLETE	TERMINATE_ REQUEST_ ABNORMALLY	END
	RECEIVE_REJECT_ VALIDITY_WITH_ NO_MANAGER	ABANDON_ REQUEST	END
	RECEIVE_ INDICATION_ OF_SERVICE_ FAILURE	ABANDON_ REQUEST	END
	SERVICE_ABORT_ REQUIRED	TERMINATE_ REQUEST_ ABNORMALLY	END
	RECEIVE_REJECT_ VALIDITY_WITH_ MANAGER	MARK_CURRENT_ MANAGER_AS_ INVALID; REINSTATE_OLD_ MANAGER; NOTIFY_ MANAGER_OF_ SERVICE_FAILURE	NOTIFY
	RECEIVE_ REDIRECT	RECORD_REDIRECT; MAKE_REQUEST	WAIT

This State	Event	Action	Next State
NOTIFY	NO_RESPONSE_FROM_MANAGER_AND_NO_MORE_MANAGERS	ABANDON_REQUEST	END
	NO_RESPONSE_FROM_MANAGER_WITH_FURTHER_MANAGERS	MARK_CURRENT_MANAGER_AS_INVALID; REINSTATE_OLD_MANAGER; MAKE_SERVICE_ENQUIRY_TO_MANAGER	ENQUIRE

This State	Event	Action	Next State
ENQUIRE	RECEIVE_INVALID MESSAGE	REJECT_INVALID MESSAGE	ENQUIRE
	TIMEOUT_BEFORE_ SERVICE_COMPLETE	TERMINATE_ REQUEST_ ABNORMALLY	END
	RECEIVE_REJECT_ VALIDITY_WITH_ NO_MANAGER	ABANDON_ REQUEST	END
	RECEIVE_ INDICATION_ OF_SERVICE_ FAILURE	ABANDON_ REQUEST	END
	SERVICE_ABORT_ REQUIRED	TERMINATE_ REQUEST_ ABNORMALLY	END
	RECEIVE_ REDIRECT	RECORD_REDIRECT; MAKE_REQUEST	WAIT
	RECEIVE_ EXTEND_TIME	ADJUST_TIMEOUT	WAIT
	RECEIVE_CONTINUE	RESET_TIMEOUT	WAIT
	RECEIVE_ VALID_REPLY	BEGIN_ CONSUMING_ REPLY	CONSUME

This State	Event	Action	Next State
ENQUIRE	NO_RESPONSE_FROM_MANAGER_AND_NO_MORE_MANAGERS	ABANDON_REQUEST	END
	RECEIVE_SUPPLY_FAIL_WITH_MANAGER	RECORD_PROVIDER_FAIL	ENQUIRE
	NO_RESPONSE_FROM_MANAGER_WITH_FURTHER_MANAGERS	MARK_CURRENT_MANAGER_AS_INVALID; REINSTATE_OLD_MANAGER; MAKE_SERVICE_ENQUIRY_TO_MANAGER	ENQUIRE
	RECEIVE_REJECT_VALIDITY_WITH_MANAGER	MARK_CURRENT_MANAGER_AS_INVALID; REINSTATE_OLD_MANAGER; MAKE_SERVICE_ENQUIRY_TO_MANAGER	ENQUIRE

B.2.2 Service Manager (S-M)

Description of States

The **Service Manager** (*S-M*) has 6 possible states :

1. BEGIN: *S-M* process has not been created.

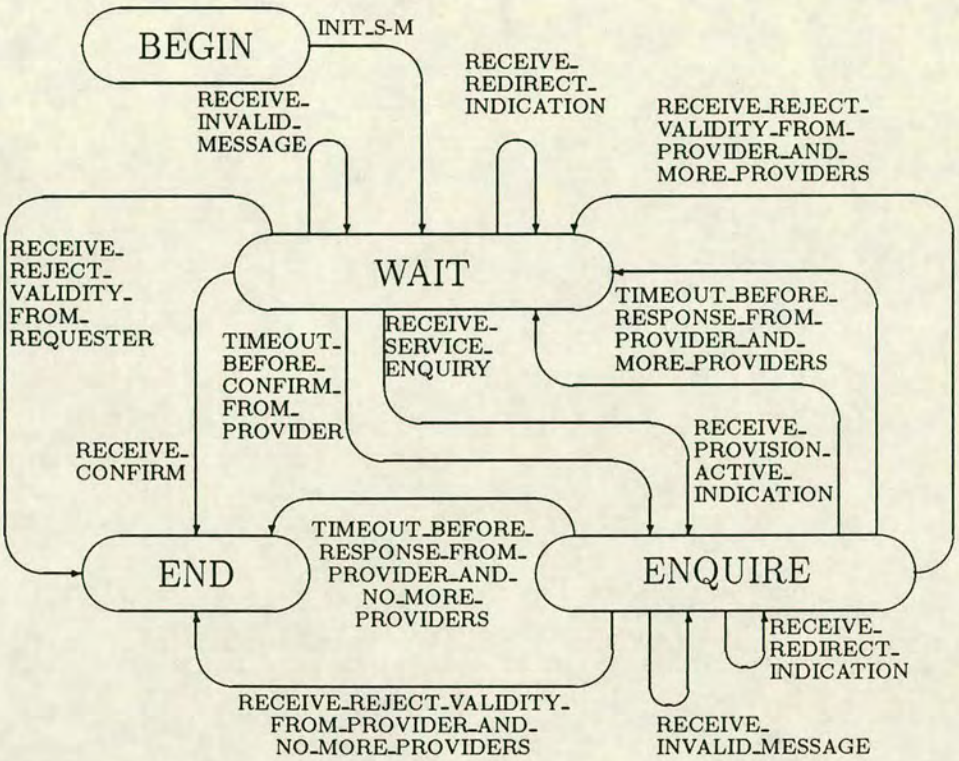

Figure B–5: Basic Protocol : Service Manager (S-M) - 'ENQUIRE'

2. WAIT: *S-M* has redirected a service request to a new *S-P* and is waiting for the confirmation of completion.

3. END: the rôle of *S-M* in the provision of the service is complete, and the process is terminated.

4. ENQUIRE: *S-M* is checking the state of the *S-P* because the provision of the service appears to have failed.

5. VERIFY: *S-M* is checking that *S-P* is aware of the failure of the service; *S-R* has already notified the *S-M* that this is the case.

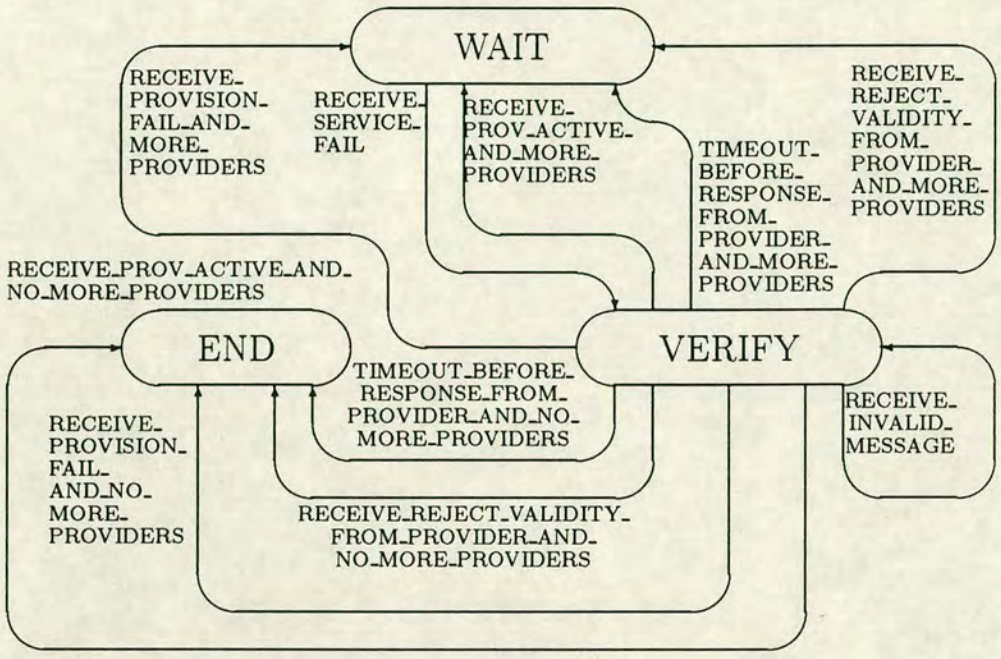

Figure B–6: Basic Protocol : Service Manager (S-M) - 'VERIFY'

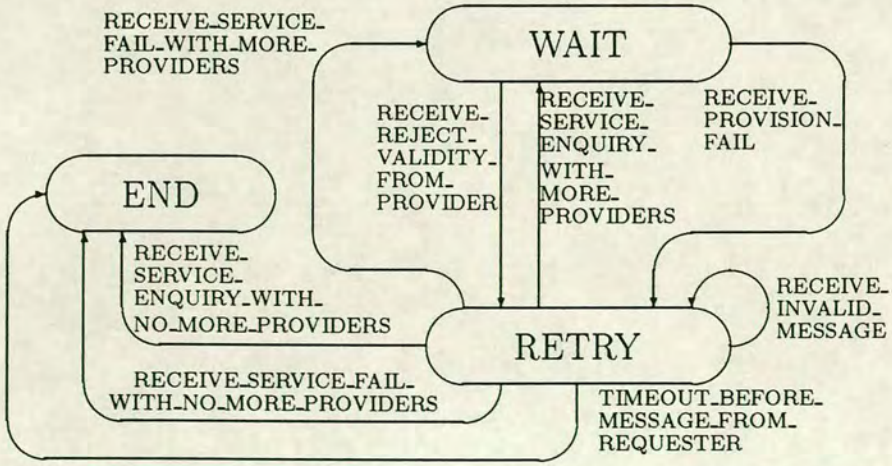

Figure B–7: Basic Protocol : Service Manager (S-M) - 'RETRY'

6. RETRY: *S-M* waits for a message from *S-R* before attempting to redirect the request to a new *S-P*. The previous *S-P* has indicated a failure in the provision of the service.

Description of Events

The **Service Manager** (*S-M*) identifies the following events :

1. INIT_S-M: the *S-M* process has just been created.

2. RECEIVE_CONFIRM: *S-M* receives **CONFIRM** from *S-P*, indicating that *S-P* has completed its provision of the service.

3. RECEIVE_REJECT_VALIDITY_FROM_REQUESTER: *S-M* receives **REJECT_VALIDITY** from *S-R* because the permit is invalid or the context is incorrect.

4. RECEIVE_INVALID_MESSAGE: *S-M* receives a message with an invalid permit or incorrect context.

5. RECEIVE_REDIRECT_INDICATION: *S-M* receives **REDIR_M** from*S-P*, indicating that the service request has been further redirected.

6. RECEIVE_SERVICE_ENQUIRY: *S-M* receives **ENQ_SERVICE** from *S-R*.

7. TIMEOUT_BEFORE_CONFIRM_FROM_PROVIDER: *S-P* has failed to confirm completion of the service within the timeout period (T_{basic5}).

8. RECEIVE_PROVISION_ACTIVE_INDICATION: *S-M* receives **PROV_ACTIVE** from *S-P*, indicating that *S-P* is actively providing the service.

9. TIMEOUT_BEFORE_RESPONSE_FROM_PROVIDER_AND_MORE_PROVIDERS: *S-P* has failed to respond to the enquiry from *S-M* within the timeout period (T_{basic6}), and there are more nodes capable of providing the service.

10. TIMEOUT_BEFORE_RESPONSE_FROM_PROVIDER_AND_NO_MORE_PROVIDERS: *S-P* has failed to respond to the enquiry from *S-M* within the timeout period (T_{basic6}), but there are no more nodes capable of providing the service.

11. RECEIVE_REJECT_VALIDITY_FROM_PROVIDER_AND_MORE_PROVIDERS: *S-M* receives **REJECT_VALIDITY** from *S-P*, and there are more nodes capable of providing the service.

12. RECEIVE_REJECT_VALIDITY_FROM_PROVIDER_AND_NO_MORE_PROVIDERS: *S-M* receives **REJECT_VALIDITY** from *S-P*, but there are no more nodes capable of providing the service.

13. RECEIVE_SERVICE_FAIL: *S-M* receives **SERVICE_FAIL** from *S-R*.

14. RECEIVE_PROV_ACTIVE_AND_MORE_PROVIDERS: *S-M* receives **PROV_ACTIVE** from *S-P*, and there are more nodes capable of providing the service.

15. RECEIVE_PROV_ACTIVE_AND_NO_MORE_PROVIDERS: *S-M* receives **PROV_ACTIVE** from *S-P*, but there are no more nodes capable of providing the service.

16. RECEIVE_PROVISION_FAIL_AND_MORE_PROVIDERS: *S-M* receives **PROVISION_FAIL** from *S-P*, and there are more nodes capable of providing the service.

17. RECEIVE_PROVISION_FAIL_AND_NO_MORE_PROVIDERS: *S-M* receives **PROVISION_FAIL** from *S-P*, but there are no more nodes capable of providing the service.

18. RECEIVE_PROVISION_FAIL: *S-M* receives **PROVISION_FAIL** from *S-P*.

19. RECEIVE_REJECT_VALIDITY_FROM_PROVIDER: *S-M* receives **REJECT_VALIDITY** from *S-P*.

20. RECEIVE_SERVICE_ENQUIRY_WITH_MORE_PROVIDERS: *S-M* receives **ENQ_SERVICE** from *S-R*, and there are more nodes capable of providing the service.

21. RECEIVE_SERVICE_ENQUIRY_WITH_NO_MORE_PROVIDERS: *S-M* receives **ENQ_SERVICE** from *S-R*, but there are no more nodes capable of providing the service.

22. RECEIVE_SERVICE_FAIL_WITH_MORE_PROVIDERS: *S-M* receives **SERVICE_FAIL** from *S-R*, and there are more nodes capable of providing the service.

23. RECEIVE_SERVICE_FAIL_WITH_NO_MORE_PROVIDERS: *S-M* receives **SERVICE_FAIL** from *S-R*, but there are no more nodes capable of providing the service.

24. TIMEOUT_BEFORE_MESSAGE_FROM_REQUESTER: *S-R* fails to alert *S-M* of a problem with the service within the timeout period (T_{basic7}).

Description of Actions

The **Service Manager** (*S-M*) may initiate the following actions :

1. DETERMINE_NEW_PROVIDER: *S-M* records details of the selected *S-P*.

2. DIRECT_PROVIDER: *S-M* sends **DIRECT** to *S-P*, with details of the permit and context.

3. REDIRECT_REQUESTER: *S-M* sends **REDIRECT** to *S-R*, with the new permit.

4. CONCLUDE_SERVICE_NORMALLY: *S-M* returns an indication of success and the process terminates.

5. ABORT_SERVICE_PROVISION: *S-M* sends **ABORT_PROVISION** to *S-P*.

6. ABANDON_SERVICE: *S-M* process terminates with an error code.

7. REJECT_INVALID_MESSAGE: *S-M* sends **REJECT_VALIDITY** to the node that sent the invalid message.

8. RECORD_DETAILS_OF_REDIRECTION: *S-M* records that the selected *S-P* has further redirected the service request.

9. MAKE_SERVICE_PROVISION_ENQUIRY: *S-M* sends **ENQ_PROVISION** to *S-P*.

10. RECORD_SERVICE_FAILURE: *S-M* records that *S-R* perceived a failure in the service.

11. RECORD_PROVISION_FAILURE: *S-M* notes that the selected *S-P* has failed.

12. INSTRUCT_REQUESTER_TO_CONTINUE: *S-M* sends **CONTINUE** to *S-R*.

13. NOTIFY_REQUESTER_OF_SERVICE_FAILURE: *S-M* sends **NOTIFY_S_FAIL** to *S-R*.

State Transition Tables

This State	Event	Action	Next State
BEGIN	INIT_S-M	DETERMINE_NEW_PROVIDER; DIRECT_PROVIDER; REDIRECT_REQUESTER	WAIT
WAIT	RECEIVE_CONFIRM	CONCLUDE_SERVICE_NORMALLY	END
	RECEIVE_REJECT_VALIDITY_FROM_REQUESTER	ABORT_SERVICE_PROVISION; ABANDON_SERVICE	END
	RECEIVE_INVALID_MESSAGE	REJECT_INVALID_MESSAGE	WAIT
	RECEIVE_REDIRECT_INDICATION	RECORD_DETAILS_OF_REDIRECTION	WAIT
	RECEIVE_SERVICE_ENQUIRY	MAKE_SERVICE_PROVISION_ENQUIRY	ENQUIRE
	TIMEOUT_BEFORE_CONFIRM_FROM_PROVIDER	MAKE_SERVICE_PROVISION_ENQUIRY	ENQUIRE
	RECEIVE_SERVICE_FAIL	RECORD_SERVICE_FAILURE	VERIFY
	RECEIVE_PROVISION_FAIL	RECORD_PROVISION_FAILURE	RETRY
	RECEIVE_REJECT_VALIDITY_FROM_PROVIDER	RECORD_PROVISION_FAILURE	RETRY

This State	Event	Action	Next State
ENQUIRE	RECEIVE_PROVISION_ACTIVE_INDICATION	INSTRUCT_REQUESTER TO_CONTINUE	WAIT
	TIMEOUT_BEFORE_RESPONSE_FROM_PROVIDER_AND_MORE_PROVIDERS	DETERMINE_NEW_PROVIDER; DIRECT_PROVIDER; REDIRECT_REQUESTER	WAIT
	RECEIVE_REJECT_VALIDITY_FROM_PROVIDER_AND_MORE_PROVIDERS	DETERMINE_NEW_PROVIDER; DIRECT_PROVIDER; REDIRECT_REQUESTER	WAIT
	RECEIVE_REDIRECT_INDICATION	RECORD_DETAILS_OF_REDIRECTION; MAKE_SERVICE_PROVISION_ENQUIRY	ENQUIRE
	RECEIVE_INVALID_MESSAGE	REJECT_INVALID_MESSAGE	ENQUIRE
	RECEIVE_REJECT_VALIDITY_FROM_PROVIDER_AND_NO_MORE_PROVIDERS	NOTIFY_REQUESTER_OF_SERVICE_FAILURE; ABANDON_SERVICE	END
	TIMEOUT_BEFORE_RESPONSE_FROM_PROVIDER_AND_NO_MORE_PROVIDERS	NOTIFY_REQUESTER_OF_SERVICE_FAILURE; ABANDON_SERVICE	END

This State	Event	Action	Next State
VERIFY	RECEIVE_PROVISION_FAIL_AND_MORE_PROVIDERS	RECORD_PROVISION_FAILURE; DETERMINE_NEW_PROVIDER; DIRECT_PROVIDER; REDIRECT_REQUESTER	WAIT
	RECEIVE_PROV_ACTIVE_AND_MORE_PROVIDERS	ABORT_SERVICE_PROVISION; DETERMINE_NEW_PROVIDER; DIRECT_PROVIDER; REDIRECT_REQUESTER	WAIT
	TIMEOUT_BEFORE_RESPONSE_FROM_PROVIDER_AND_MORE_PROVIDERS	DETERMINE_NEW_PROVIDER; DIRECT_PROVIDER; REDIRECT_REQUESTER	WAIT
	RECEIVE_REJECT_VALIDITY_FROM_PROVIDER_AND_MORE_PROVIDERS	RECORD_PROVISION_FAILURE; DETERMINE_NEW_PROVIDER; DIRECT_PROVIDER; REDIRECT_REQUESTER	WAIT

This State	Event	Action	Next State
VERIFY	RECEIVE_INVALID_ MESSAGE	REJECT_INVALID_ MESSAGE	VERIFY
	RECEIVE_ PROVISION_ FAIL_AND_ NO_MORE_ PROVIDERS	NOTIFY_REQUESTER_ OF_SERVICE_ FAILURE; ABANDON_SERVICE	END
	RECEIVE_REJECT_ VALIDITY_FROM_ PROVIDER_AND_ NO_MORE_ PROVIDERS	NOTIFY_REQUESTER_ OF_SERVICE_ FAILURE; ABANDON_SERVICE	END
	TIMEOUT_BEFORE_ RESPONSE_FROM_ PROVIDER_AND_ NO_MORE_	NOTIFY_REQUESTER_ OF_SERVICE_ FAILURE; ABANDON_SERVICE	END
	RECEIVE_PROV_ ACTIVE_AND_ NO_MORE_ PROVIDERS	ABORT_SERVICE_ PROVISION; NOTIFY_REQUESTER_ OF_SERVICE_ FAILURE; ABANDON_SERVICE	END

This State	Event	Action	Next State
RETRY	RECEIVE_SERVICE_ENQUIRY_WITH_MORE_PROVIDERS	DETERMINE_NEW_PROVIDER; DIRECT_PROVIDER; REDIRECT_REQUESTER	WAIT
	RECEIVE_SERVICE_FAIL_WITH_MORE_PROVIDERS	DETERMINE_NEW_PROVIDER; DIRECT_PROVIDER; REDIRECT_REQUESTER	WAIT
	RECEIVE_INVALID_MESSAGE	REJECT_INVALID_MESSAGE	RETRY
	RECEIVE_SERVICE_FAIL_WITH_NO_MORE_PROVIDERS	NOTIFY_REQUESTER_OF_SERVICE_FAILURE; ABANDON_SERVICE	END
	RECEIVE_SERVICE_ENQUIRY_WITH_NO_MORE_PROVIDERS	NOTIFY_REQUESTER_OF_SERVICE_FAILURE; ABANDON_SERVICE	END
	TIMEOUT_BEFORE_MESSAGE_FROM_REQUESTER	ABANDON_SERVICE	END

B.2.3 Service Provider (S-P)

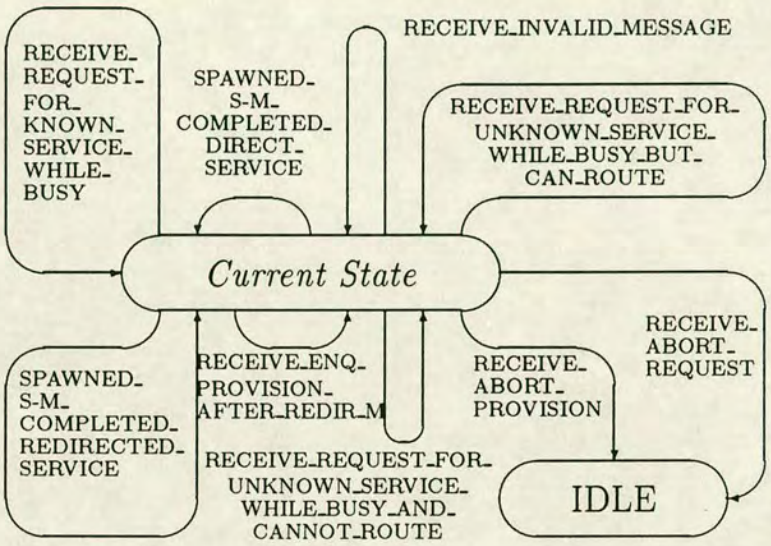

Figure B–8: Basic Protocol : Service Provider (S-P) - All States

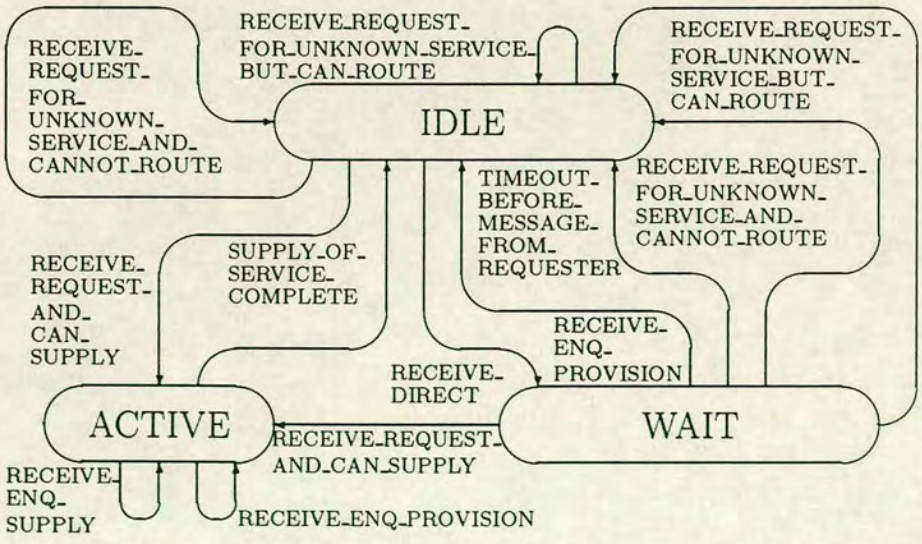

Figure B–9: Basic Protocol : Service Provider (S-P) - 'WAIT'

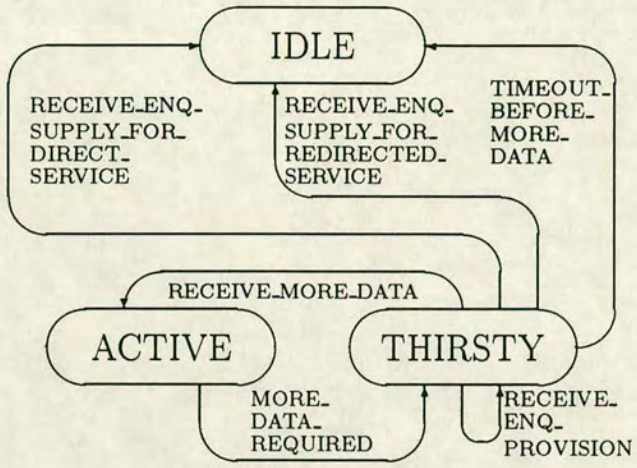

Figure B–10: Basic Protocol : Service Provider (S-P) - 'THIRSTY'

Description of States

The **Service Provider** (*S-P*) has 4 possible states :

1. IDLE: *S-P* is not currently providing a service, and is awaiting a new request.

2. ACTIVE: *S-P* is actively providing a service.

3. WAIT: *S-P* has been directed to provide a redirected service, and is awaiting the request.

4. THIRSTY: *S-P* has received a service request that provided insufficient data for the service to be completed. *S-P* is waiting for more data to be supplied by *S-R*.

Description of Events

The **Service Provider** (*S-P*) identifies the following events :

1. RECEIVE_INVALID_MESSAGE: *S-P* receives a message with an invalid permit or an incorrect context.

2. RECEIVE_ABORT_REQUEST: *S-P* receives **ABORT_REQUEST** from *S-R*.

3. RECEIVE_ABORT_PROVISION: *S-P* receives **ABORT_PROVISION** from *S-M*.

4. SPAWNED_S-M_COMPLETED_DIRECT_SERVICE: a service request, redirected by an *S-M* created by *S-P*, has been completed and the *S-M* process has terminated.

5. SPAWNED_S-M_COMPLETED_REDIRECTED_SERVICE: a service request, already redirected to *S-P*, has been further redirected by an *S-M* created by *S-P*. The service has been completed and the *S-M* process has terminated.

6. RECEIVE_REQUEST_FOR_KNOWN_SERVICE_WHILE_BUSY: *S-P* receives **REQUEST** from an *S-R* whilst it is already servicing a request from another *S-R*. The *S-P* is able to provide the required service.

7. RECEIVE_REQUEST_FOR_UNKNOWN_SERVICE_WHILE_BUSY_BUT_CAN_ROUTE: *S-P* receives **REQUEST** from an *S-R* for a service that it cannot provide whilst it is already servicing a request from another *S-R*. The *S-P* knows another *S-P* that is able to provide the required service.

8. RECEIVE_REQUEST_FOR_UNKNOWN_SERVICE_WHILE_BUSY_AND_CANNOT_ROUTE: *S-P* receives **REQUEST** from an *S-R* for a service that it cannot provide whilst it is already servicing a request from another *S-R*.

The *S-P* does not know of another *S-P* that is able to provide the required service.

9. RECEIVE_REQUEST_AND_CAN_SUPPLY: *S-P* receives **REQUEST** from an *S-R* and can supply the required service.

10. RECEIVE_DIRECT: *S-P* receives **DIRECT** from *S-M*, indicating that it is to expect a redirected service request.

11. RECEIVE_REQUEST_FOR_UNKNOWN_SERVICE_BUT_CAN_ROUTE: *S-P* receives **REQUEST** from *S-R*, is unable to supply the required service, but knows another node that can.

12. RECEIVE_REQUEST_FOR_UNKNOWN_SERVICE_AND_CANNOT_ROUTE: *S-P* receives **REQUEST** from *S-R*, is unable to supply the required service, and does not know of another node that can.

13. RECEIVE_ENQ_PROVISION_AFTER_REDIR_M:
S-P receives **ENQ_PROVISION** from *S-M* for a service that has already been redirected further by an *S-M* created by *S-P*.

14. SUPPLY_OF_SERVICE_COMPLETE: *S-P* has completed provision of the required service.

15. TIMEOUT_BEFORE_MESSAGE_FROM_REQUESTER: *S-R* has failed to send a redirected **REQUEST** to *S-P* within the timeout period (T_{basic8}).

16. RECEIVE_ENQ_PROVISION: *S-P* receives **ENQ_PROVISION** from *S-M*.

17. RECEIVE_ENQ_SUPPLY: *S-P* receives **ENQ_SUPPLY** from *S-R*.

18. MORE_DATA_REQUIRED: *S-P* requires more information from *S-R* before the service may be provided.

19. RECEIVE_MORE_DATA: *S-P* receives **GIVE_MORE_DATA** from *S-R*, providing additional information that *S-P* requires for provision of the service.

20. TIMEOUT_BEFORE_MORE_DATA: *S-R* has failed to supply additional information to *S-P* within the timeout period (T_{basic9}).

21. RECEIVE_ENQ_SUPPLY_FOR_DIRECT_SERVICE:
    S-P receives **ENQ_SUPPLY** from *S-R*, where no previous **DIRECT** had been sent from *S-M*.

22. RECEIVE_ENQ_SUPPLY_FOR_REDIRECTED_SERVICE: *S-P* receives **ENQ_SUPPLY** from *S-R* for a service that has been redirected by *S-M*.

Description of Actions

The **Service Provider** (*S-P*) may initiate the following actions :

1. REJECT_INVALID_MESSAGE: *S-P* sends **REJECT_VALIDITY** to the node that sent the invalid message.

2. RECORD_S-M_DEATH: *S-P* marks the redirected service as complete, and removes the entry from the table of active spawned managers.

3. CONFIRM_SERVICE_COMPLETION: *S-P* sends **CONFIRM** to the *S-M* that sent the **DIRECT**.

4. INFORM_MANAGER_OF_PROVISION_FAIL: *S-P* sends **PROVISION_FAIL** to the *S-M*.

5. TERMINATE_SERVICE: *S-P* terminates the process executing the service application.

6. NOTIFY_MANAGER_OF_SERVICE_REDIRECTION: *S-P* sends **REDIR_M** to the *S-M*, indicating that it has further redirected the service.

7. INDICATE_SUPPLY_FAIL: *S-P* sends **SUPPLY_FAIL** to the *S-R*.

8. CREATE_NEW_S-M: *S-P* spawns a process to redirect the service request.

9. RECORD_SERVICE_REDIRECTION: *S-P* updates the table of active spawned managers and marks the service as being further redirected.

10. INITIATE_SERVICE_PROVISION: *S-P* spawns a process to run the service application.

11. RECORD_SERVICE_DETAILS: *S-P* notes the expected source of the request, the permit and the context.

12. RETURN_RESULTS_TO_REQUESTER: *S-P* sends **REPLY** to the *S-R*, together with the output data and an indication of the success of the service.

13. INSTRUCT_REQUESTER_TO_EXTEND_TIME: *S-P* sends **EXTEND_TIME** to *S-R*, indicating that extra time is required in which to complete the service.

14. INDICATE_PROVISION_ACTIVE: *S-P* sends **PROV_ACTIVE** to *S-M*.

15. REQUEST_MORE_DATA: *S-P* sends **REQ_MORE_DATA** to *S-R*, with details of stream channels, if needed.

16. CONSUME_EXTRA_DATA: *S-P* passes the extra data to the service application.

State Transition Tables

This State	Event	Action	Next State
IDLE	RECEIVE_INVALID MESSAGE	REJECT_INVALID MESSAGE	IDLE
	SPAWNED_S-M_ COMPLETED_ DIRECT_SERVICE	RECORD_S-M_ DEATH	IDLE
	SPAWNED_S-M_ COMPLETED_ REDIRECTED_ SERVICE	RECORD_S-M_ DEATH; CONFIRM_ SERVICE_ COMPLETION	IDLE
	RECEIVE_ ABORT_ REQUEST	INFORM_ MANAGER_ OF_PROVISION_ FAIL; TERMINATE_ SERVICE	IDLE
	RECEIVE_ABORT_ PROVISION	TERMINATE_ SERVICE	IDLE
	RECEIVE_ENQ_ PROVISION_ AFTER_REDIR_M	NOTIFY_ MANAGER_ OF_SERVICE_ REDIRECTION	IDLE
	RECEIVE_REQUEST_ FOR_KNOWN_ SERVICE_WHILE_ BUSY	INDICATE_ SUPPLY_ FAIL	IDLE

This State	Event	Action	Next State
IDLE	RECEIVE_REQUEST_FOR_UNKNOWN_SERVICE_WHILE_BUSY_AND_CANNOT_ROUTE	INDICATE_SUPPLY_FAIL	IDLE
	RECEIVE_REQUEST_FOR_UNKNOWN_SERVICE_WHILE_BUSY_BUT_CAN_ROUTE	CREATE_NEW_S-M; RECORD_SERVICE_REDIRECTION	IDLE
	RECEIVE_REQUEST_AND_CAN_SUPPLY	INITIATE_SERVICE_PROVISION	ACTIVE
	RECEIVE_DIRECT	RECORD_SERVICE_DETAILS	WAIT
	RECEIVE_REQUEST_FOR_UNKNOWN_SERVICE_BUT_CAN_ROUTE	CREATE_NEW_S-M; RECORD_SERVICE_REDIRECTION	IDLE
	RECEIVE_REQUEST_FOR_UNKNOWN_SERVICE_AND_CANNOT_ROUTE	INDICATE_SUPPLY_FAIL	IDLE

This State	Event	Action	Next State
ACTIVE	RECEIVE_INVALID MESSAGE	REJECT_INVALID MESSAGE	ACTIVE
	SPAWNED_S-M_ COMPLETED_ DIRECT_SERVICE	RECORD_S-M_ DEATH	ACTIVE
	SPAWNED_S-M_ COMPLETED_ REDIRECTED_ SERVICE	RECORD_S-M_ DEATH; CONFIRM_ SERVICE_ COMPLETION	ACTIVE
	RECEIVE_ ABORT_ REQUEST	INFORM_ MANAGER_ OF_PROVISION_ FAIL; TERMINATE_ SERVICE	IDLE
	RECEIVE_ABORT_ PROVISION	TERMINATE_ SERVICE	IDLE
	RECEIVE_ENQ_ PROVISION_ AFTER_REDIR_M	NOTIFY_MANAGER_ OF_SERVICE_ REDIRECTION	ACTIVE
	RECEIVE_REQUEST_ FOR_KNOWN_ SERVICE_WHILE_ BUSY	INDICATE_ SUPPLY_ FAIL	ACTIVE

This State	Event	Action	Next State
ACTIVE	RECEIVE_REQUEST_FOR_UNKNOWN_SERVICE_WHILE_BUSY_AND_CANNOT_ROUTE	INDICATE_SUPPLY_FAIL	ACTIVE
	RECEIVE_REQUEST_FOR_UNKNOWN_SERVICE_WHILE_BUSY_BUT_CAN_ROUTE	CREATE_NEW_S-M; RECORD_SERVICE_REDIRECTION	ACTIVE
	SUPPLY_OF_SERVICE_COMPLETE	RETURN_RESULTS_TO_REQUESTER	IDLE
	RECEIVE_ENQ_SUPPLY	INSTRUCT_REQUESTER_TO_EXTEND_TIME	ACTIVE
	RECEIVE_ENQ_PROVISION	INDICATE_PROVISION_ACTIVE	ACTIVE
	MORE_DATA_REQUIRED	REQUEST_MORE_DATA	THIRSTY

This State	Event	Action	Next State
WAIT	RECEIVE_INVALID MESSAGE	REJECT_INVALID MESSAGE	WAIT
	SPAWNED_S-M_ COMPLETED_ DIRECT_SERVICE	RECORD_S-M_ DEATH	WAIT
	SPAWNED_S-M_ COMPLETED_ REDIRECTED_ SERVICE	RECORD_S-M_ DEATH; CONFIRM_ SERVICE_ COMPLETION	WAIT
	RECEIVE_ ABORT_ REQUEST	INFORM_ MANAGER_ OF_PROVISION_ FAIL; TERMINATE_ SERVICE	IDLE
	RECEIVE_ABORT_ PROVISION	TERMINATE_ SERVICE	IDLE
	RECEIVE_ENQ_ PROVISION_ AFTER_REDIR_M	NOTIFY_MANAGER_ OF_SERVICE_ REDIRECTION	WAIT
	RECEIVE_REQUEST_ FOR_KNOWN_ SERVICE_WHILE_ BUSY	INDICATE_ SUPPLY_ FAIL	WAIT

This State	Event	Action	Next State
WAIT	RECEIVE_REQUEST_FOR_UNKNOWN_SERVICE_WHILE_BUSY_AND_CANNOT_ROUTE	INDICATE_SUPPLY_FAIL	WAIT
	RECEIVE_REQUEST_FOR_UNKNOWN_SERVICE_WHILE_BUSY_BUT_CAN_ROUTE	CREATE_NEW_S-M; RECORD_SERVICE_REDIRECTION	WAIT
	RECEIVE_REQUEST_AND_CAN_SUPPLY	INITIATE_SERVICE_PROVISION	ACTIVE
	RECEIVE_ENQ_PROVISION	INFORM_MANAGER_OF_PROVISION_FAIL	IDLE
	TIMEOUT_BEFORE_MESSAGE_FROM_REQUESTER	INFORM_MANAGER_OF_PROVISION_FAIL	IDLE
	RECEIVE_REQUEST_FOR_UNKNOWN_SERVICE_AND_CANNOT_ROUTE	INDICATE_SUPPLY_FAIL; INFORM_MANAGER_OF_PROVISION_FAIL	IDLE
	RECEIVE_REQUEST_FOR_UNKNOWN_SERVICE_BUT_CAN_ROUTE	CREATE_NEW_S-M; RECORD_SERVICE_REDIRECTION; NOTIFY_MANAGER_OF_SERVICE_REDIRECTION	IDLE

This State	Event	Action	Next State
THIRSTY	RECEIVE_INVALID MESSAGE	REJECT_INVALID MESSAGE	THIRSTY
	SPAWNED_S-M_ COMPLETED_ DIRECT_SERVICE	RECORD_S-M_ DEATH	THIRSTY
	SPAWNED_S-M_ COMPLETED_ REDIRECTED_ SERVICE	RECORD_S-M_ DEATH; CONFIRM_ SERVICE_ COMPLETION	THIRSTY
	RECEIVE_ ABORT_ REQUEST	INFORM_ MANAGER_ OF_PROVISION_ FAIL; TERMINATE_ SERVICE	IDLE
	RECEIVE_ABORT_ PROVISION	TERMINATE_ SERVICE	IDLE
	RECEIVE_ENQ_ PROVISION_ AFTER_REDIR_M	NOTIFY_MANAGER_ OF_SERVICE_ REDIRECTION	THIRSTY
	RECEIVE_REQUEST_ FOR_KNOWN_SERVICE_ WHILE_BUSY	INDICATE_ SUPPLY_ FAIL	THIRSTY

This State	Event	Action	Next State
THIRSTY	RECEIVE_REQUEST_FOR_UNKNOWN_SERVICE_WHILE_BUSY_AND_CANNOT_ROUTE	INDICATE_SUPPLY_FAIL	THIRSTY
	RECEIVE_REQUEST_FOR_UNKNOWN_SERVICE_WHILE_BUSY_BUT_CAN_ROUTE	CREATE_NEW_S-M; RECORD_SERVICE_REDIRECTION	THIRSTY
	RECEIVE_MORE_DATA	CONSUME_EXTRA_DATA	ACTIVE
	RECEIVE_ENQ_PROVISION	INDICATE_PROVISION_ACTIVE	THIRSTY
	TIMEOUT_BEFORE_MORE_DATA	INFORM_MANAGER_OF_PROVISION_FAIL; TERMINATE_SERVICE	IDLE
	RECEIVE_ENQ_SUPPLY_FOR_DIRECT_SERVICE	INDICATE_SUPPLY_FAIL; TERMINATE_SERVICE	IDLE
	RECEIVE_ENQ_SUPPLY_FOR_REDIRECTED_SERVICE	INDICATE_SUPPLY_FAIL; INFORM_MANAGER_OF_PROVISION_FAIL; TERMINATE_SERVICE	IDLE

Appendix C

The Special Service Provision Protocol

C.1 Table of Primitives

Primitive	From	To	Purpose
RUN_JOB	*S-M**	*S-P**	initiate provision of a specialised service; transfers description of job required
JOB_COMPLETE	*S-P**	*S-M**	indicate completion of the specified task; returns details of the behaviour of the job
STATUS_REQUEST	*S-M**	*S-P**	poll for current state of BEM and job
STATUS_REPORT	*S-P**	*S-M**	details of the current state
KILL_JOB	*S-M**	*S-P**	premature termination of service provision
JOB_STOPPED	*S-P**	*S-M**	notice/confirmation of premature end
SEEK_MANAGER	*S-P**	*S-M**	determine association between BEM and its manager
BECOME_MANAGER	*S-M**	*S-P**	establish bond between BEM and manager

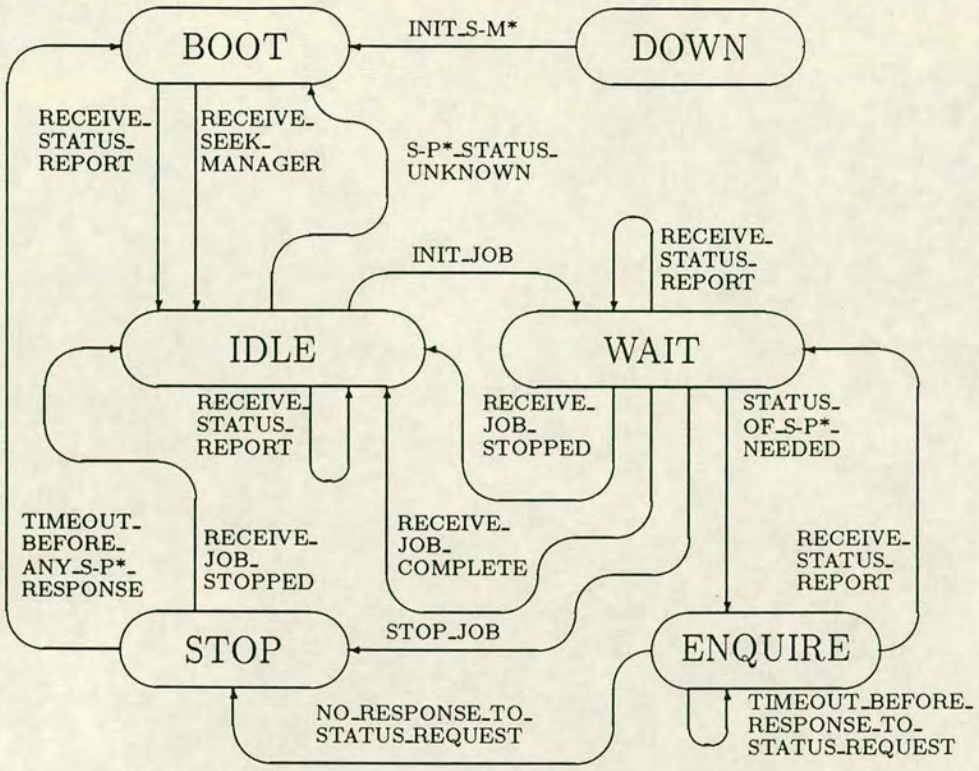

Figure C–1: Special protocol : Special Service Manager (S-M*)

C.2 Protocol Definition

C.2.1 Special Service Manager (S-M*)

Description of States

The **Special Service Manager** (*S-M**) has 6 possible states :

1. DOWN: *S-M** is not enabled - the process has not been created.

2. BOOT: *S-M** attempts to form a relationship with its *S-P**.

3. IDLE: *S-P** returns regular **STATUS_REPORT**s to *S-M**, but there is no job active.

4. WAIT: *S-P** is actively processing a job started by *S-M**. *S-P** returns **STATUS_REPORT**s to *S-M** in one of two forms:

   (a) periodic, unsolicited messages.

   (b) in response to a **STATUS_REQUEST** from *S-M**.

5. ENQUIRE: *S-M** waits for a **STATUS_REPORT** from *S-P**, having sent an explicit **STATUS_REQUEST**.

6. STOP: The job is being terminated prematurely.

Description of Events

The **Special Service Manager** (*S-M**) identifies the following events :

1. INIT_S-M*: the *S-M** process has just been created.

2. RECEIVE_STATUS_REPORT: *S-M** receives **STATUS_REPORT** from the *S-P**.

3. RECEIVE_SEEK_MANAGER: *S-M** receives **SEEK_MANAGER** from the *S-P**; the *S-P** will have been created or enabled after the *S-M**.

4. INIT_JOB: *S-M** is instructed to provide a special service.

5. S-P*_STATUS_UNKNOWN: *S-P** has not sent a **STATUS_REPORT** within the timeout period ($T_{special1}$).

6. RECEIVE_JOB_COMPLETE: *S-P** has completed processing the job and has sent **JOB_COMPLETE** to *S-M**.

7. RECEIVE_JOB_STOPPED: processing of the job failed to complete normally and *S-P** has sent **JOB_STOPPED** to *S-M**.

8. STOP_JOB: *S-M** is instructed to terminate provision of the service, or a constraint of the job processing has been violated.

9. STATUS_OF_S-P*_NEEDED: *S-M** requires an update to the status of the *S-P**. The *S-P** may have failed to provide a periodic message.

10. TIMEOUT_BEFORE_RESPONSE_TO_STATUS_REQUEST: *S-P** has failed to provide a **STATUS_REPORT** within the timeout period ($T_{special2}$).

11. NO_RESPONSE_TO_STATUS_REQUEST: *S-P** has not sent a **STATUS_REPORT** within the timeout period ($T_{special3}$).

12. TIMEOUT_BEFORE_ANY_S-P*_RESPONSE: *S-P** has not sent a valid message within the timeout period ($T_{special4}$).

Description of Actions

The **Special Service Manager** (*S-M**) may initiate the following actions :

1. LOOK_FOR_S-P*: *S-M** sends **BECOME_MANAGER** to the *S-P**.

2. LOG_S-P*_STATUS: *S-M** records the information provided in the **STATUS_REPORT** from *S-P**.

3. RECORD_S-P*_DETAILS: *S-M** records details about *S-P** from the **SEEK_MANAGER**.

4. FORM_BOND_WITH_NEW_S-P*: *S-M** sends **BECOME_MANAGER** to the *S-P**.

5. SEND_JOB_DESCRIPTION: *S-M** sends **RUN_JOB** to the *S-P**.

6. CONCLUDE_JOB_WITH_NORMAL_END: *S-M** returns the results of the job with an indication of success.

7. CONCLUDE_JOB_WITH_ABNORMAL_END: *S-M** returns an indication of failure with details.

8. TERMINATE_JOB: *S-M** sends **KILL_JOB** to the *S-P**.

9. REQUEST_STATUS_OF_S-P*: *S-M** sends **STATUS_REQUEST** to the *S-P**.

State Transition Tables

This State	Event	Action	Next State
DOWN	INIT_S-M*	LOOK_FOR_S-P*	BOOT
BOOT	RECEIVE_ STATUS_REPORT	LOG_S-P*_ STATUS	IDLE
	RECEIVE_SEEK_ MANAGER	RECORD_S-P*_ DETAILS; FORM_ BOND_WITH_ NEW_S-P*	IDLE
IDLE	INIT_JOB	SEND_JOB_ DESCRIPTION	WAIT
	RECEIVE_ STATUS_REPORT	LOG_S-P*_ STATUS	IDLE
	S-P*_STATUS_ UNKNOWN	LOOK_FOR_S-P*	BOOT
WAIT	RECEIVE_JOB_ COMPLETE	CONCLUDE_JOB_ WITH_NORMAL_ END	IDLE
	RECEIVE_JOB_ STOPPED	CONCLUDE_JOB_ WITH_ABNORMAL_ END	IDLE
	RECEIVE_ STATUS_REPORT	LOG_S-P*_ STATUS	WAIT
	STOP_JOB	TERMINATE_JOB	STOP
	STATUS_OF_S-P*_ NEEDED	REQUEST_STATUS_ OF_S-P*	ENQUIRE

This State	Event	Action	Next State
ENQUIRE	RECEIVE_STATUS_REPORT	LOG_S-P*_STATUS	WAIT
	TIMEOUT_BEFORE_RESPONSE_TO_STATUS_REQUEST	REQUEST_STATUS_OF_S-P*	ENQUIRE
	NO_RESPONSE_TO_STATUS_REQUEST	TERMINATE_JOB	STOP
STOP	RECEIVE_JOB_STOPPED	CONCLUDE_JOB_WITH_ABNORMAL_END	IDLE
	TIMEOUT_BEFORE_ANY_S-M*_RESPONSE	LOOK_FOR_S-P*	BOOT

C.2.2 Special Service Provider (S-P*)

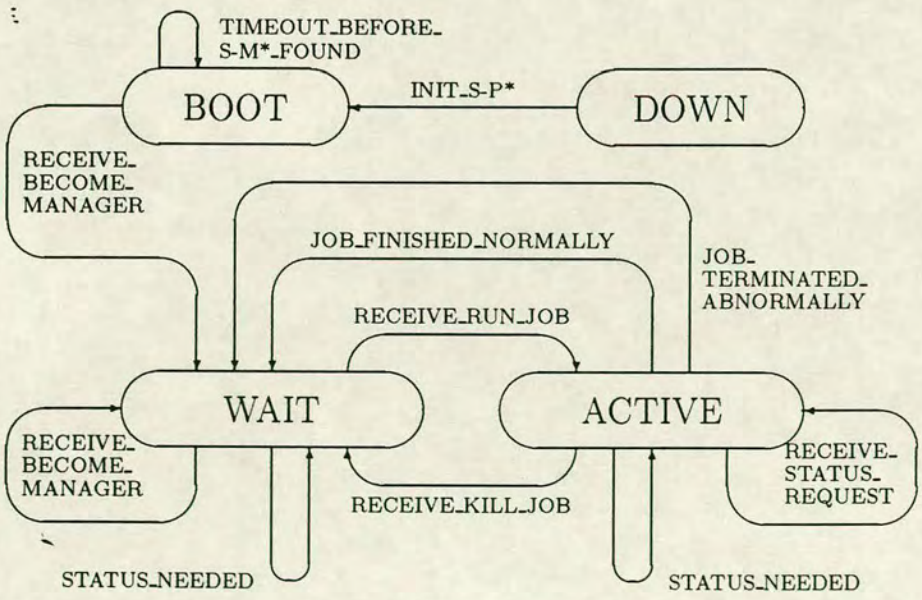

Figure C–2: Special protocol : Special Service Provider (S-P*)

Description of States

The **Special Service Provider** (*S-P**) has 4 possible states :

1. DOWN: *S-P** is not enabled or the process has not been created.

2. BOOT: *S-P** attempts to form a relationship with its *S-M**.

3. WAIT: there is no job processing active, but *S-P** will still send regular **STATUS_REPORT**s to the *S-M**.

4. ACTIVE: *S-P** is processing a job started by *S-M**. It may return **STATUS_REPORT**s to *S-M** in one of two forms :

   (a) periodic, unsolicited messages.

   (b) in response to an explicit **STATUS_REQUEST** from *S-M**.

Description of Events

The **Special Service Provider** (*S-P**) identifies the following events :

1. INIT_S-P*: the *S-P** process has just been created.

2. RECEIVE_BECOME_MANAGER: *S-P** receives **BECOME_MANAGER** from the *S-M**.

3. TIMEOUT_BEFORE_S-M*_FOUND: *S-M** failed to send **BECOME_MANAGER** within the timeout period ($T_{special5}$).

4. RECEIVE_RUN_JOB: *S-P** receives **RUN_JOB** from *S-M**; this is the description of the job, and a directive to commence processing.

5. STATUS_NEEDED: the interval between the regular **STATUS_REPORT**s ($T_{special6}$) has passed.

6. JOB_FINISHED_NORMALLY: processing of the job has been completed successfully.

7. RECEIVE_KILL_JOB: *S-P** receives **KILL_JOB** from *S-M**; this is a directive to terminate processing of the job.

8. JOB_TERMINATED_ABNORMALLY: processing of the job has been terminated because of an abnormal condition.

9. RECEIVE_STATUS_REQUEST: *S-P** receives an explicit **STATUS_REQUEST** from *S-M** for a **STATUS_REPORT**.

Description of Actions

The **Special Service Provider** (*S-P**) may initiate the following actions :

1. LOOK_FOR_S-M*: *S-P** sends **SEEK_MANAGER** to the *S-M**.

2. FORM_BOND_WITH_S-M*: *S-P** records details about *S-M**, as supplied in **BECOME_MANAGER**.

3. BEGIN_JOB: *S-P** begins processing the job, as described in the **RUN_JOB**.

4. SUPPLY_STATUS: *S-P** sends a **STATUS_REPORT** to *S-M**.

5. CONCLUDE_JOB_NORMALLY: *S-P** records statistics and accounting information for the completed job.

6. INDICATE_JOB_COMPLETION: *S-P** sends the results of the job to *S-M** in **JOB_COMPLETE**.

7. TERMINATE_JOB_ABNORMALLY: *S-P** stops processing of the job.

8. SUPPLY_DETAILS_OF_JOB_FAILURE: *S-P** returns information about the final state of the terminated job in **JOB_STOPPED** to *S-M**.

State Transition Tables

This State	Event	Action	Next State
DOWN	INIT_S-P*	LOOK_FOR_S-M*	BOOT
BOOT	RECEIVE_BECOME_ MANAGER	FORM_BOND_ WITH_S-M*	WAIT
	TIMEOUT_BEFORE_ S-M*_FOUND	LOOK_FOR_S-M*	BOOT
WAIT	RECEIVE_RUN_JOB	BEGIN_JOB	ACTIVE
	STATUS_NEEDED	SUPPLY_STATUS	WAIT
	RECEIVE_BECOME_ MANAGER	SUPPLY_STATUS	WAIT
ACTIVE	JOB_FINISHED_ NORMALLY	CONCLUDE_JOB_ NORMALLY; INDICATE_JOB_ COMPLETION	WAIT
	RECEIVE_ KILL_JOB	TERMINATE_JOB_ ABNORMALLY; SUPPLY_DETAILS_ OF_JOB_FAILURE	WAIT
	JOB_TERMINATED_ ABNORMALLY	SUPPLY_DETAILS_ OF_JOB_FAILURE	WAIT
	STATUS_NEEDED	SUPPLY_STATUS	ACTIVE
	RECEIVE_STATUS_ REQUEST	SUPPLY_STATUS	ACTIVE

Appendix D Published Paper

A Model for the Design of High Performance Protocols for a Networked Computing Environment

Gary D. Law
Spider Systems Limited, 65 Bonnington Road, Edinburgh EH6 5JQ, U.K.

Abstract

The advent of high bandwidth local area networks means that it is now possible to interconnect large numbers of devices with widely differing processing capabilities in such a manner that the various devices may closely interact. Before such a system may be realised, it is necessary to define a set of conventions to govern the way in which the network nodes interact with each other.

The Triadic Network Model is introduced to describe the interactions between personal computers and high performance computers over a general purpose network by using a third classification of devices, whose purpose is to support the operation of the network as a distributed system. The model is not restricted to any particular network or operating system, but may form the interface between the two. Hence, the use of a protocol set based on this model enables the provision of a uniform interface to the applications software. Some of the principles of a protocol set developed from the Triadic Network Model are presented in the paper.

Introduction

High performance computers are extremely expensive. Usually they provide specialised services that are only occasionally required by individual users but which may be considered desirable by a large number of users. Consequently installations housing such high performance computers attempt to make these machines accessible to as many users as possible. In the past this has been achieved by attaching the high performance computer as a 'back-end' to a general purpose multi-user computer, which has the capability to handle a large number of terminals or batch jobs.

The advent of high bandwidth local area networks, of which **Centrenet** [3] is one example, means that it is now feasible to interconnect large numbers of devices with widely differing processing capabilities in such a manner that the various levels may closely interact.

General Network Structure

To give some indication of the wide range of devices that may be attached to a network to form a distributed system, consider the network structure illustrated in figure 1. This structure is considered to be representative of configurations that will soon be in widespread existence.

The range of machines to which the users of the system have direct access is wide and includes 'dumb' terminals with no inherent computational capabilities. There are personal computers with somewhat basic facilities and personal workstations that have fairly substantial resources. There are also medium performance multi-user computers, to which terminals may be connected directly or via the network.

Of the machines that are connected to the network which the user cannot directly access, some are high performance special purpose processors that have been designed for particular types of computation, e.g. vector processing. The architectures of these processors are optimised for their respective applications. Hence they are not suitable for direct interaction with the users of the system.

Services are provided by dedicated servers for use by all of the devices attached to the network. These

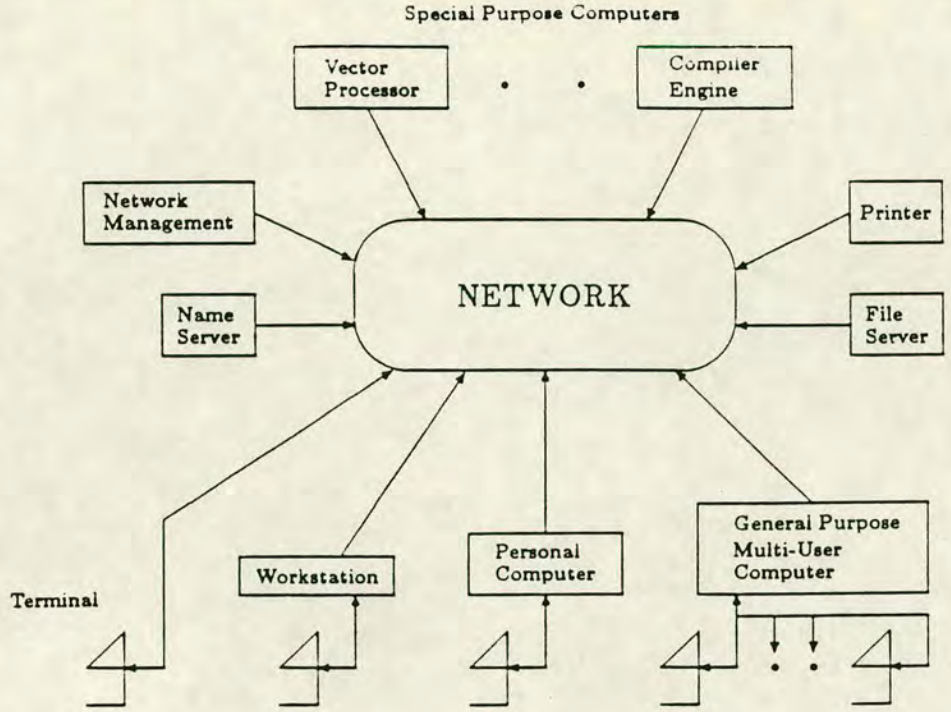

Figure 1: A General Network Structure

services include file storage and access to peripherals such as printers and graph plotters. The remaining devices are purely concerned with the management and control of the system. The facilities that they may provide and the degree of control that they exert are determined by particular installations and no generalisation is possible.

The network could have any one of a number of topologies and use many different types of media; the implementation adopted for a specific system will be selected according to its requirements [6][8].

Before such a system may be realised, it is necessary to define a set of conventions to govern the way in which the network nodes interact with each other. A well structured set of conventions will enable the individual devices to work as a unified system; without any government at all the devices operate disjointly, contesting for use of the network.

Triadic Classification of Network Devices

By dividing the wide variety of network devices of figure 1 into a limited number of classes, it is possible to isolate the requirements of communication within and between these differing classes, with the ultimate objective of providing protocols oriented towards efficiently satisfying these needs. Three classes are distinguished that give an indication of areas of current development, and which represent those machines that may be expected to exist in future network environments.

User Workstations : Suitable examples are the Apollo [5], SUN and TORCH XXX workstations, the IBM Personal System/2, and the Apple Macintosh II.

Network Servers : This includes most peripherals, but the management functions of the network operating system would have to be supplied by additional types of node. The **Centrenet NIM** (Network Intelligence Module) is an example of this particular type of network server. Centrenet [3] is a packet-switched local area network with a hierarchical tree structure. The network incorporates dedicated network devices (NIMs) that perform name-serving and virtual circuit management functions.

Back-End Processors : The sort of computers that would be classified as back-end processors include fast numerical processors such as MU6V [4], specialist mathematical processors, compiler engines [1], simulators, image processors, language translators and inference processors. MU6V is a good example of a Back-End Processor. The processor incorporates mechanisms to achieve a high performance in vector

calculations. It is designed to operate as a special purpose processor attached to a local area network and hence it has a minimum of operating system software.

These three classes of device provide a focus for further considerations of the way in which the devices will require use of the network. However, this simple three-way classification is inadequate because of the degree of variation that may occur in the nature of individual members of one division. So, to strengthen this framework, the model defined in the next section takes these three classes of device and isolates those features that are of primary concern.

A Triadic Network Model

The **Triadic Network Model** is an idealised representation of a network structure, such as that shown in figure 1. The model defines three types of module that correspond to these classes of device: **Personal Workstation Modules**(PWMs), **Back-End modules** (BEMs) and **Network Service Modules** (NSMs). The model defines the form of the modules, but primarily it exists to describe the nature of the interactions between them, from which protocols may be developed. In figure 2, the domain of each type of module is shown as a triangle and where the triangles touch it indicates that the domains may interact.

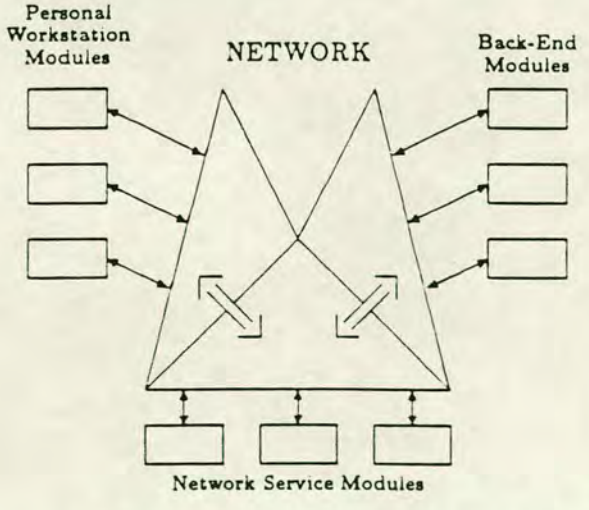

Figure 2: Triadic Network Model

Personal Workstation Modules (PWMs)

The **Personal Workstation Modules** are notionally general purpose single-user workstations. Each PWM is considered to have good computational capabilities and a fairly substantial amount of memory. They possess powerful input and output facilities, possibly including colour graphics and sound for output, and using a keyboard, mouse, digitiser or speech for input.

The PWMs are very much user oriented and contain a good deal of resources, both hardware and software, that are specifically intended to support a good interface between the user and the system, i.e. a good *"Human-Computer Interface"*. These resources allow the development and use of *"Intelligent Front-Ends"* [2] within the PWMs themselves, although this is not their solitary purpose. The PWMs have sufficient processing power for many of the user's computational requirements, allowing use as a 'stand-alone' computer that acts completely independently of the rest of the system.

The connection of the PWMs to the system provides the advantages to the user of allowing the use of shared network resources such as printers and file storage on discs, and access to specialised services supplied by high performance processors [7]. However, the most important use of the network by the PWMs is for interaction with each other, both individually and in groups. This interaction may take the form of textual messages (passed by a *mailbox* utility or directly), sound or vision.

Back End Modules (BEMs)

The **Back-End Modules** are special purpose computers. They are designed to achieve a high performance with great efficiency in particular applications. Each BEM is intended for use in a specific type of application. Consequently the BEMs bear little or no resemblance to one other and will probably have quite strict requirements with regards to the form of their input and output. These requirements may be quite complex and the users are unlikely to be able to comply with them without significant effort. Similarly, because of their dissimilarity, the BEMs may find great difficulty in direct communication with each other but BEM intercommunication is not likely to be a frequent occurrence.

The BEMs may be considered to be individual resources, where each BEM corresponds to a single resource. The resource may only be capable of utilisation by a single process at any one time. This requires quite advanced resource management, which may be

provided by the BEMs, but only at the expense of their efficiency. The BEMs make use of the resources supplied by the network, such as the filestore, to obviate unnecessary duplication of such facilities.

Network Service Modules (NSMs)

This is the third classification of devices used by the model. As illustrated in figure 2, the **Network Service Modules** serve as an *'interface'* between the other two domains and also supply resources for use by the PWMs and BEMs. The presence of the NSM domain is desirable from the viewpoint of both the PWM and BEM domains since it permits economical sharing of frequently used resources, such as large disc space, printers etc., and it enables the BEMs to operate at their optimum efficiency, by removing all operating system concerns.

The NSMs are dedicated network servers whose purpose is to assist the general operation of the distributed system. The services which they might provide include file storage, data bases, peripheral access, virtual circuit establishment, background resource management (e.g. compilation) and special purpose resource management. Collectively the NSMs provide a **Network Operation Support Service** (NOSS) required for the implementation of a distributed system.

The NSMs allow even more advanced types of service to be provided than the individual BEMs are capable of supplying. An advanced network service may be created by utilising a number of BEMs and mapping all requests for this new service onto a set of interactions with the group of BEMs. For example, an individual image processor BEM may not be capable of operating at sufficient speed to cope with real time data. So, by using a number of image processor BEMs, a NSM may take 'live' video pictures and share the job of image enhancement between the group of BEMs. The NSM can then combine the outputs and direct them to the required destination.

Notice that an alternative approach would be for an NSM to multiplex a number of service requests from PWMs onto a single BEM. The resulting service would be of a lower quality than the BEM could provide for a single process, but the level of service desired by the PWMs may actually be quite low, in which case the BEM is being used more effectively by the system through the use of this simple mapping technique.

The organisational part of a distributed operating system comprises the functions that it is inappropriate to locate on either PWMs or BEMs, and therefore these functions are located on the NSMs. Consequently, the NSMs are involved in accounting, scheduling, protection (e.g. store management), synchronization and communication, and the allocation and management of resources. Most of these functions are provided by those NSMs that are dedicated to the management of the system, but some accounting may be performed by the individual resources and many of the NSMs will be concerned with protection. Additionally, the presence of communication functions is obviously a prerequisite of all NSMs because of their interconnection by the network.

The key position occupied by the NSM domain means that there are three important requirements of the NSMs:-

reliability : since these devices are critical to the operation of the full system, the domain should not be susceptible to total failure through the failure of a single node.

security : the NSMs control access to expensive resources and specialised services, as well as managing intercommunication between PWMs, so they must be capable of protecting the other devices from illicit intrusion by unauthorised users.

integrity : in the case of the use of a BEM service, the NSMs manage the entire transaction so they are responsible for ensuring successful total completion of each request.

Relating Devices to Modules

The model is an idealised representation of the computer system. So, the PWM, NSM and BEM, as defined by the Triadic Network Model, are merely logical entities. As illustrated in figure 3, the individual machines that will approximate to these modules are most likely to have capabilities in excess of those defined by the model.

CDC Cyber 205 : The specification of fileserver and scheduler services, resident with the processor, enables other nodes to exploit these functions. This may permit network file transfer and processor job scheduling to be achieved without the involvement of other NSMs.

Meiko Computing Surface : The disc storage units attached to the main processor are not specified here, so they are only for the private use of the processor. This is appropriate for holding local code modules and providing virtual memory paging facilities.

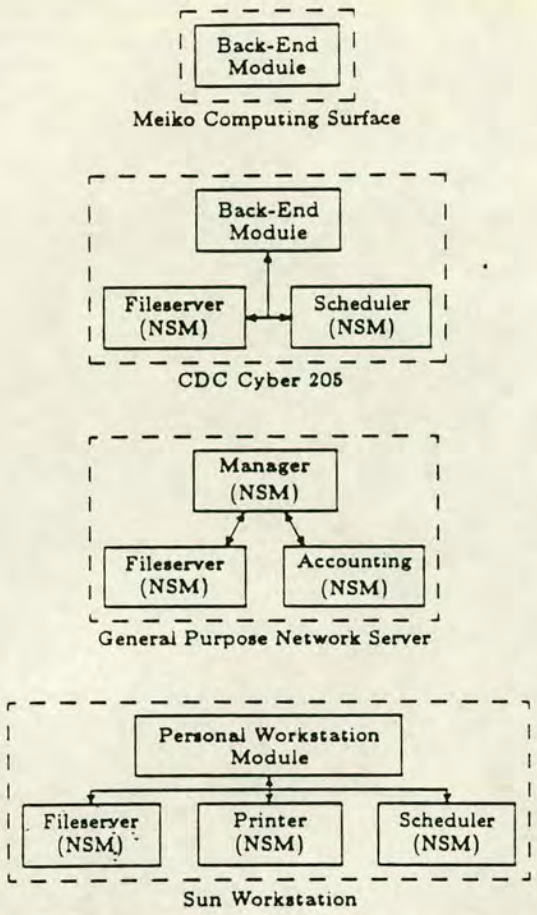

Figure 3: Examples of Network Devices

General purpose network server : A general purpose minicomputer may be used to provide a number of network services. In this manner, the cost of implementing some of the standard, but infrequently used, network services may be reduced. For services that will be in greater demand, it is cost effective to provide a dedicated server node.

Sun workstation : The workstation is shown as having local storage and scheduling facilities that may be accessed by other network nodes, as well as having a printer directly attached.

Interactions in the Model

The fundamental restrictions on interactions imposed by the model are :-

1. **Back-End Modules** never interact with each other
2. **Back-End Modules** do not communicate directly with **Personal Workstation Modules**

Interactions within and between the domains of the PWMs and NSMs are allowed, but interactions with the BEMs are only allowed with the NSMs.

Workstation Intercommunication

The communication between the PWMs is of the following forms:-

- interactive transfer of small amounts of text (c.f. VAX/VMS 'phone' utility)
- digitised speech
- digitised pictures, whether 'live' video or computer generated graphics.

For efficiency, it is desirable that there is minimal involvement of NSMs in this intercommunication. However, a more fundamental network requirement, that the topology of the network and physical locations of the network devices remain invisible to individual nodes, seems to conflict with this aim. This is because only the NOSS is considered to be aware of the actual state of the network. Therefore, before any two devices on the network may interact directly they must first determine their respective physical addresses by communication with the NOSS.

Hence, the rôle of the NSMs in this type of interaction should be restricted to that of *'name server'*, i.e. the PWM wishing to make contact with another PWM initially interacts with the NOSS **Name Server** module to determine the physical address of the other node; the subsequent establishment, utilisation and termination of the actual connection are controlled by the PWMs. An alternative situation, where multiple PWM connections are required (for teleconferencing, for example) would justify the use of an NSM to assist the 'chairman'. Similarly, PWM intercommunication between users speaking in different languages may be assisted by the use of a translator BEM. In this case the involvement of an NSM is essential.

Hence, the initial enquiry sent from the PWM to the NOSS should state the type of intercommunication to be used. For basic communication the NOSS Name Server module would just return the address of the destination PWM. If a more advanced type of communication is required the NOSS returns an acknowledgement to the source PWM and then endeavours to provide the desired service.

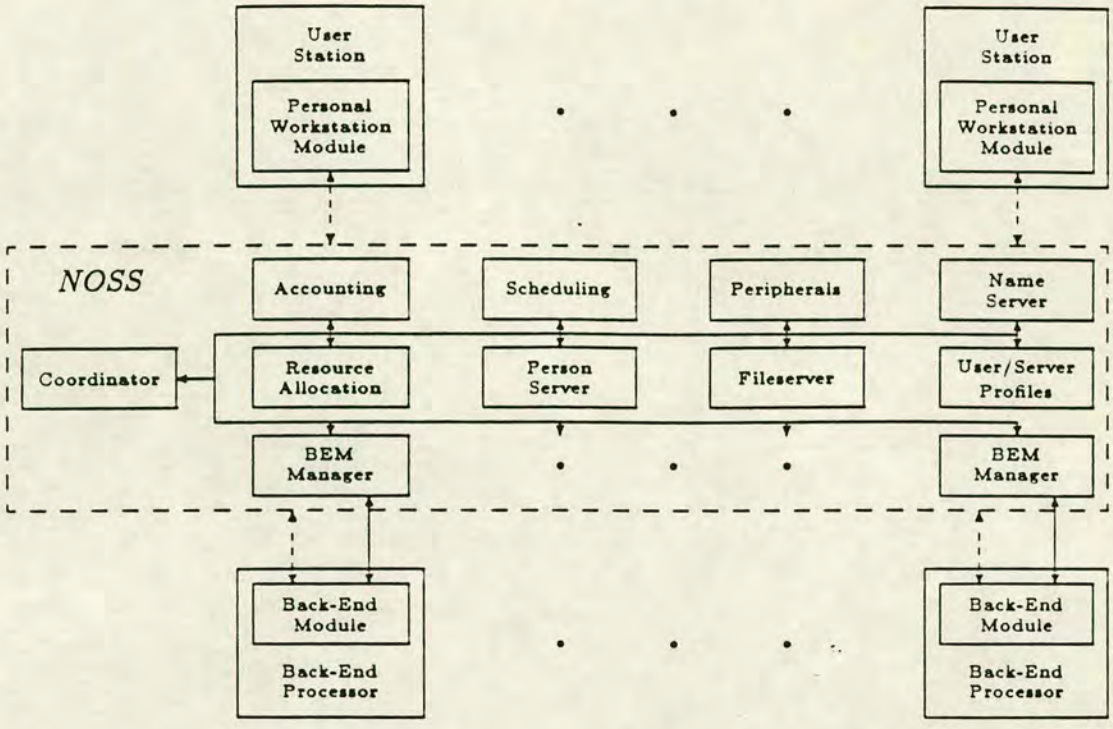

Figure 4: A Networked Computing Environment

Use of Standard Network Services

The standard network services are those functions that the NSMs provide for use by both the PWMs and the BEMs. Included in this category are the network filestore and peripheral access. Access to any of these services is initiated by the sending of a request to the NOSS **Coordinator** module of figure 4. This request is validated and then, possibly after having been queued temporarily, it is forwarded to a suitable NSM. Any subsequent interactions between the customer and the supplier pass directly without any involvement of the NOSS Coordinator module. On completion of the service request, the NSM that has supplied the service indicates its availability to the Coordinator and it may then be used by another customer.

For most of the transaction, the nature of the interactions is largely as described by the **Client-Server** model. The differences lie in the initiation and the termination procedures. Here, the protocols are of a three-party form, with the NOSS Coordinator module acting as mediator in negotiations between the requesting PWM and the supplier NSM.

Use of Special Network Services

As far as the PWMs are concerned, the mechanism for accessing specialised network services is the same as that for use of the standard services. The PWM sends the service request to the NOSS Coordinator module, as before. The Coordinator treats the service request largely as for a standard service and, after validation and scheduling, the request is passed to another NOSS module. This module acts as the BEM's **Manager** and any subsequent interactions with the customer PWM, needed for the effecting of the function, are made by the BEM's Manager rather than by the BEM itself. So the PWM is never made aware of which BEM is acting as the supplier.

Since the BEM's Manager is responsible for all high level interactions with the PWM, the only network protocols that need to be implemented on the BEM are of quite a basic nature. Additionally the BEM need only be concerned with the sort of computation for which it was designed; there is no requirement for a sophisticated user interface or for advanced operating system functions. Hence the BEM may operate with a high degree of efficiency.

Layered Network Interface

It is generally agreed that it is desirable to make a heterogeneous multicomputer system appear as if it were a single entity because this greatly simplifies the applications software and provides a more uniform view of the system to the users. It is also considered preferable to make the protocols governing interaction between the different nodes on the system quite basic so that it is straightforward to implement them on a wide range of devices.

The **Triadic Network Model** would appear to violate both of these aims because of the widely differing levels of processing capability of the different network devices, and the distinction between standard and special services. The idea of the various parts of the system having different degrees of awareness of the nature of the network is introduced in an attempt to overcome these difficulties. This use of a layered interface to the network allows the sophisticated nature of the network structure to be concealed from the user by relatively basic pieces of software.

User's View

From the uppermost layer, the user is aware of having access to a large and comprehensive range of services, which enable him to communicate with fellow users, interrogate a large information base and perform extensive processing of information. Access to these services is provided by the user's personal workstation, and to the user it may appear that this machine is performing functions that are in fact being executed elsewhere on the network. Similarly, the application software is only aware of the functions that are available. The software communicates with all of the processes supplying the functions in the same manner, regardless of whether they are internal or external to the workstation.

Workstation's view

The workstation divides the global set of functions into two categories, local and non-local. It may be possible for some functions to reside in both categories, if different levels of service are recognised and the network is able to offer the same functions but with a higher level of service.

For local functions, communication from the application software is passed directly to the appropriate internal process. For non-local functions, the workstation will first send a service request to the network and then pass the information to the network server. The workstation is only aware of the division between local and non-local functions; it has no knowledge of the difference in the types of non-local functions.

Network's view

The network distinguishes between standard and specialised services. For standard services, the service request is passed to the corresponding NSM, but for specialised services the request is passed to another NOSS module to manage the transaction.

When the NOSS is regarded as composed of a number of modules, it is evident that the NOSS Coordinator module itself draws no distinction at all. The distinction arises out of the way in which the services are actually implemented: for a standard service, the Coordinator passes the service request directly to the NOSS module that will provide the service, whereas for a specialised service the destination NOSS module uses another machine, a BEM, to implement the service.

Application of the Model

The objective of the work described in this document is to enable a distributed operating system to be implemented on a computer system composed of a multiplicity of different types of processors, all of which communicate over a common network, which itself may comprise a variety of sub-nets. The aim is to allow the operation of high-level applications software executing on the processors that collectively form the computing system, with all the software using well-defined and uniform interfaces to the remainder of the system.

The applications software may view the system as a supplier of a broad range of services. Use of any service is gained through the issuing of a *request*, with the system, in turn, providing a *response* on completion of the service. The system is responsible for implementing the required service by utilising one or more logical modules. Each module has a specific purpose and exists locally or is available via the network. Through the use of the same *request-response* mechanism for all services, whether they are provided locally or remotely, the details of the implementation may be concealed from the user or application.

The principles of the Triadic Network Model (TNM) may be applied in the design of a protocol set for a heterogeneous multi-computer system. The following sections descibe some of the principles of one such protocol set.

Aspects of the Protocol Set

The system towards which this work is directed aims to achieve a high performance by moving each process to the processor that has an architecture and instruction set most suited to the computational requirements of the process. However, if the mechanism that effects this process migration is very inefficient, then its usefulness is greatly diminished - possibly to the extent that it becomes less efficient to move the process than to run it locally in the initiating processor.

So, the process should not be moved between processors unless doing so will improve the overall performance of the execution of the process, including the overheads incurred by the transfer. With this objective in mind, the TNM protocols strive to minimise these overheads as much as possible.

Three-Party Mechanism

To support the provision of a uniform interface to the applications software, a mechanism is required that enables any single request for use of a network service to be re-routed through the system to a node capable of providing the service. The **Three-Party mechanism** is fundamental to the correct operation of all the sub-sets of the TNM protocol set. Its principles of operation are:

1. When a network-supplied service is required by an application, but cannot be satisfied locally by the network node, a **service request** is made to the system. The network node then becomes the **Service Requester** *(S-R)*.

2. If the recipient of this request is capable of providing the service, it satisfies the request and then sends a response back to the *S-R*. If it cannot provide the required service then it will either return an indication of its failure to the *S-R* or it may re-direct the request to a node which, it believes, to be capable of supporting the desired service. If this node re-directs the original request, it becomes the **Service Manager** *(S-M)* for the duration of this session.

3. To re-direct the request, the *S-M* must send an indication to the intended node, together with some means of identifying the session to ensure that only the appropriate *S-R* is served. The node to which the service request is re-directed is the **Service Provider** *(S-P)*, and a **Service Permit** is the means of identifying the session.

4. The *S-M* must also return an indication to the *S-R* that the service request should be re-directed, and again the service permit should be provided.

5. If the *S-R* receives, in reply to a service request, an indication that the request is to be re-directed, then it should re-submit the service request, together with the corresponding service permit, to the substitute server, *S-P*.

6. The *S-P*, upon receipt of the request, should first ensure that the service permit is correct and then satisfy the request. The response should then be returned to the *S-R* and finally confirmation provided to the *S-M* that the service request has been satisfied.

So, to summarise, *S-R* is the node that issues the original service request, *S-P* is the node that actually satisfies the service request and *S-M* is the node that re-directs the request to *S-P*. The principle of operation may be clearer from figure 5.

Service Permits

The use of **Service Permits** enables some necessary flexibility to be introduced into the use of network services, whilst retaining security and control over resources. A service permit is, in essence, a 'capability' or 'token' enabling a process to make use of a particular resource. The permit only has a limited period of validity and the interval between the issuing and re-issue of a permit is much greater than this validity period. This aids the security of the system by making it difficult for 'rogue' nodes to break security keys embedded within the service permit. For a system where security is not a major concern, it is possible for 'block allocations' of service permits to be made, such that a given node may make direct requests for use of a service without resort to obtaining a permit from an *S-M* first.

The use of the service permit mechanism means that the requester of a service retains control over the use of the service and may also help analysis of service provision failures, as in table 1. Further exchanges of information between the various nodes are needed to determine the exact nature of the failure.

Service Managers

The three-party mechanism only requires nodes to assume the rôle of **Service Manager** on a temporary basis; it is 'accountable' for the provision of the service until the service request has been satisfied although this responsibility is only really essential in

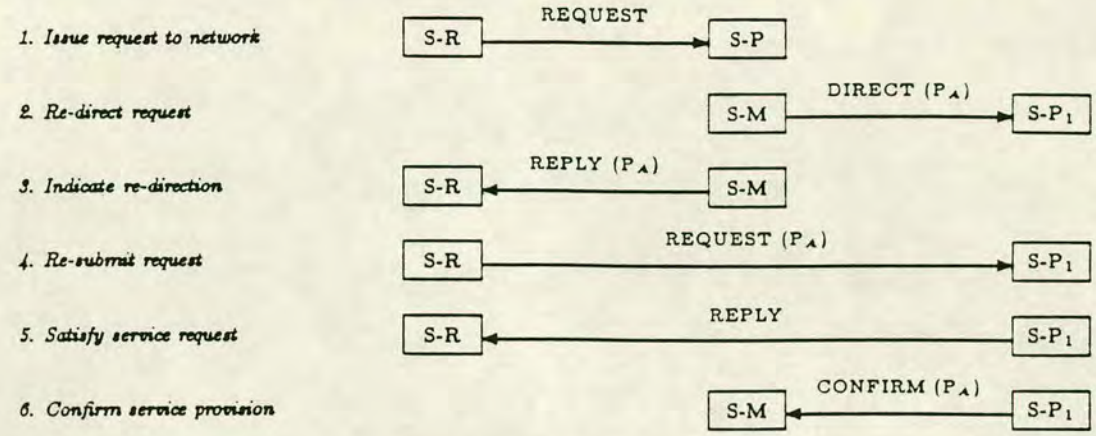

Figure 5: 3-Party Mechanism

Event	Error
S-R presents permit but S-P refuses to accept it	S-P failure or permit expiry
S-P awaits permit that is never presented	S-R failure
S-M receives no confirm	S-P failure
S-P returns confirmation but S-M confused	S-M failure or S-P failure

Table 1: Error Condition Determination

the event of failure of the service. However, there are certain types of service manager that are more permanent in nature. These include the Coordinator(s), BEM managers and the managers of communication between PWMs. Despite being permanent, as opposed to transient, these nodes may still obey the same type of protocol, with service permits, but may make rather more sophisticated use of the mechanism.

When a BEM manager receives a service request, it may re-direct the service request to the BEM, if appropriate, but as an alternative it may assume the rôle of "front-end" to the BEM, such that all communication from the service requester must pass to the BEM via the manager. This should only be performed for the transfer of essential job descriptions to enable the BEM manager to determine how the request should be satisfied. The transfer of code and data should be performed directly for greater efficiency.

Coordinator

All of the nodes on the network need only be aware of a single logical node, known as the **Coordinator**, to which all requests for use of network services are sent. In turn, the coordinator should use the Three-Party mechanism to forward any service requests to other network nodes capable of providing the required service.

The notion of a single coordinator serves to aid the provision of a uniform interface to the applications software although in reality, there should be a number of coordinator modules within the system to reduce the system's susceptibility to total system failure resulting from single node failures. Additionally, some "caching" of network addresses for services may occur so that service requests may be made directly to the provider without any need to access the coordinator in the first instance.

The coordinator is effectively just a specific instance of a service manager, but is considered by the higher levels of software to be permanent rather than transient, as is the case for most service managers. It is the only service manager visible to the higher level software, in that the software may presume that any service request sent to it will in turn be forwarded. However, no special consideration need be taken of this fact since, as far as the protocols are concerned, it behaves just like any other NSM.

Perspective

The protocols are oriented towards the management of specific activities that involve accesses to network services. This places the TNM protocols at the session layer of the ISO Open Systems Interconnection 7-layer reference model. Indeed the present implementation is based above the TCP/IP transport service. However, whilst is would be possible to develop an application layer protocol above the TNM protocols for use by higher level software, it is considered that the existing implementation provides sufficient functionality for direct use by the applications.

Minimisation of the number of protocol layers used can result in greater performance because the network overheads are lower. Hence, the high level functionality of the TNM protocols serves to improve the performance of the system by eliminating the need for extra protocol layers. Taking this approach further, the TNM protocols could be implemented directly above the network layer if the network offered a high reliability. The **Centrenet Burst Protocol** [3] provides a reliable communications substrate on which the TNM protocols may operate efficiently.

Future Directions

The **Triadic Network Model** describes the operation and interaction of three categories of device in an idealised environment. A protocol set based on the Triadic Network Model can provide a uniform interface to the applications software and thereby remove some of the difficulties involved in developing a distributed operating system.

The implementation of services by the protocol set outlined in this paper is achieved by means of a **three-party mechanism**, which involves the use of **service permits** issued by a **service manager**. These techniques may be used to enhance the security and fault tolerance of the system.

The TNM protocol set has been implemented in the Computer Science Department of Edinburgh University on a High Level Hardware ORION minicomputer running UNIX 4.2 BSD. A multi-computer environment has been simulated using the ORION, and this has been used to verify the correct operation of the protocols. A fuller indication of the performance benefits of the TNM protocol set will be obtained following completion of the implementation of the protocols on other machines and the provision of a streamlined Transport Service to allow more effective use the Centrenet Burst Protocol.

The eventual system will comprise workstations, dedicated network servers and high performance special purpose processors. The network will be composed of Ethernet and Centrenet, allowing full exploitation of the high bandwidth and inherent network intelligence of the latter, whilst permitting a wide range of devices to be connected to the system with interfaces to the former.

Acknowledgements

The author wishes to express his gratitude to Prof. R.N. Ibbett and Dr. N.P. Topham of Edinburgh University Computer Science Department, and acknowledges the financial support of the SERC.

References

[1] R.P. Bird, "A Compiler Server Node in a Local Area Network", *Proc. Int. Comp. Symp. on Application Systems Development*, BG Teubner, Stuttgart, March 22-24, 1983, p. 182.

[2] A. Bundy, "Intelligent Front-Ends", *Infotech State of the Art Report on Expert Systems*, vol. 12:7, 1984, pp. 15-24.

[3] R.N. Ibbett, D.A. Edwards, T.P. Hopkins, C.K. Cadogan and D.A. Train, "Centrenet - A High Performance Local Area Network", *Computer Journal*,vol. 28, no. 3, 1985, pp. 231-242.

[4] R.N. Ibbett, P.C. Capon and N.P. Topham, "MU6V: A Parallel Vector Processing System", *Proc. 12th Int. Symp. on Computer Architecture*, June 17-19, 1985, pp. 136-144.

[5] P.J. Leach, P.H. Levine, B.P. Duros, J.A. Hamilton, D.L. Nelson and B.L. Stumpf, "The Architecture of an Integrated Local Network", *IEEE Trans. on Comm.*, Local Area Networks Special, Nov. 1983.

[6] R.M. Metcalfe and D.R. Boggs, "Ethernet: Distributed Packet Switching for Local Computer Networks", *CACM*, vol. 19, no. 7, July 1976, pp. 395-404.

[7] J.H. Morris, M. Satyanarayanan, M.H. Conner, J.H. Howard, D.S.H. Rosenthal and F.D. Smith, "Andrew: A Distributed Personal Computing Environment", *CACM*, vol. 29, no. 3, March 1986, pp. 184-201.

[8] R.M. Needham and A.J. Herbert, *"The Cambridge Distributed Operating System"*, Addison-Wesley, Reading, Mass., 1982.