Remove this page before binding. (The page number was placed before realizing that this was to be the title page.)

Wisdom: The Foundation of a Scalable Parallel Operating Syste

Kevin Murray

Submitted for the degree of Doctor of Philosophy

University of York

Computer Science

16th February 1990

Acknowledgements

The way to Hell is easy;
night and day the gates of black Dis stand open;
but to retrace the step and reach the breezes above,
this is the task and in it the labour

Virgil's Aeneid, Book VI

Doctoral research is a great undertaking, and like any epic journey the traveller will hope to encounter much guidance and help on the voyage. I have been no different, and must thank all those who have helped me come to this point. My thanks go first to all the members of the Department of Computer Science here at York, both past and present, staff and student – it is an excellent place full of helpful people who have made my work possible. To single out any seems unfair, but I must thank Martin Atkins, Paul Austin, Charles Forsyth and Geoff Macharia for the many constructive discussions we have had.

I must also thank my supervisor, Andy Wellings, for his help throughout, and giving me enough free rein to enjoy my research whilst stopping me from hanging myself or becoming lost (I don't know how he did it!). His repeated explanations of the niceties of technical writing and understandable English should not go unthanked (even if I still have difficulty with them).

The readability of this thesis, such as it is, is entirely due to those that have proof read it: Rob Stone and Andy Wellings, and to those that have read parts of it: Paul Austin, Helen Braithwaite and Geoff Macharia. To them also go my thanks, and commiserations.

Finally my thanks must go to you, kind reader, for giving some point to my research – I hope that you find it of interest and use.

Declaration

Certain parts of this thesis have appeared in previously published papers, for which the author was the principal author, specifically the following references.

> K.A. Murray and A.J. Wellings, ‘‘Issues in the Design and Implementation of a Distributed Operating System for a Network of Transputers’’, *Microprocessing and Microprogramming, Proceeding of EUROMICRO ‘88, Zurich,* **1988**(24), pp. 169-78 (29th August-2nd September 1988).
>
> Kevin Murray and Andy J. Wellings, ‘‘Wisdom: A Distributed Operating System for Transputers’’, *Computer Systems: Science and Engineering,* **5**(1), pp. 13-20 (January 1990), Butterworth.

The conditions for deadlock, given in chapter 3, and the list of various performance metrics, given in chapter 5 come from Peterson and Silberschatz’s book Operating System Concepts.

> James. L. Peterson and Abraham Silberschatz, *Operating Systems Concepts*, Addison Wesley, Austin, Texas

Abstract

There is a demand for ever more powerful computing facilities, and for the ability to improve the power of those that are already owned, without the need to replace machines. This thesis examines a method of meeting these demands by the development of a scalable computing facility – one that can be built from as many processors as needed, and which can have more added whenever there is a need to increase the power available.

The approach taken is to provide a scalable operating system, called *Wisdom*, that will run on a scalable architecture, a network of point to point connected processors. This thesis details the issues in, requirements of, and problems with building such an operating system. The lessons of research in the field of distributed operating systems, and other specialist work in appropriate areas are used to develop a model which provides the necessary support for a usable scalable system. The model so derived consists of three degrees of transparency: interconnection transparency, location transparency and parallelism transparency. Each of these degrees of transparency is supported by one of the basic elements of *Wisdom*: naming, load balancing and routing, which represent the foundations of a scalable operating system. These three components are examined and described in detail.

The evaluation shows that a scalable system, using the techniques described in this thesis, can be built. The conclusions list areas of this work that still need further research, and based on the experience of developing *Wisdom* the characteristics of the ideal processor for a scalable architecture are listed.

Introduction

‘‘I’ve learned in Tobiki the wisdom of gracious acceptance. You see, I don’t want to be a world leader. I’ve made peace with myself somewhere between my ambitions and my limitations .... It’s a step backward in the right direction.’’

Glenn Ford to *Marlon Brando* in the film *The Teahouse of August Moon*

Early in the 1950s, when computing was in its infancy, the British government attempted to calculate the computing resources that the British Isles would need for the foreseeable future. Its conclusion was that three of the very first computers would suffice. Today, to even consider such a thing is obvious folly – no matter the power of available computing resources there is a demand for faster, more powerful machines. Yet, due both to current technological barriers and to theoretical constraints[55], there are limits as to how powerful a *single* computer can be made. In order to overcome these limits, and provide the power desired at an acceptable price, people have long considered the possibility of connecting computers together to produce machines more powerful than those built merely from a single computer.

These ‘multicomputers’ fall into four broad categories. The first are those systems which, as well as having a central processing unit, contain computers that are dedicated to controlling single devices; these have now evolved into systems with intelligent peripherals controllers. The next category category covers computers with specialised coprocessors, such as floating point units. The third category, and a more recent development, is that of distributed systems where many computers are connected together by a network. In most of these distributed systems, the individual computers exist as computers in their own right (i.e. in the case of operating systems, each has a complete, or near complete, copy of the operating system). Currently much interest has been shown in the fourth and final category, that of parallel processors. It is a subclass of this that will be studied in this thesis – loosely coupled systems using point-to-point communication interconnections.

Multicomputer systems have been classified in two main papers. The first, by Anderson and Jensen[4], divided systems up using their hardware communication mechanisms as shown in figure 0.1. Under this taxonomy, most distributed systems, are classified as DSB. That is, they have direct (D) communications between all nodes, over a shared (S) communication media, which is a bus (B). The term *bus* is taken here to include such communication media as ethernet and token passing ring. Some similar types are classified as DSM, which use memory rather than a bus for communications (M). In contrast, the parallel systems studied here are classified as IDDR, which means they have indirect communication (I, any two nodes are not guaranteed to be directly connected), decentralised routing (D, each node performs some of the message routing), dedicated paths (D, the link between two processors is dedicated to those two processors) and the network is regular (R).

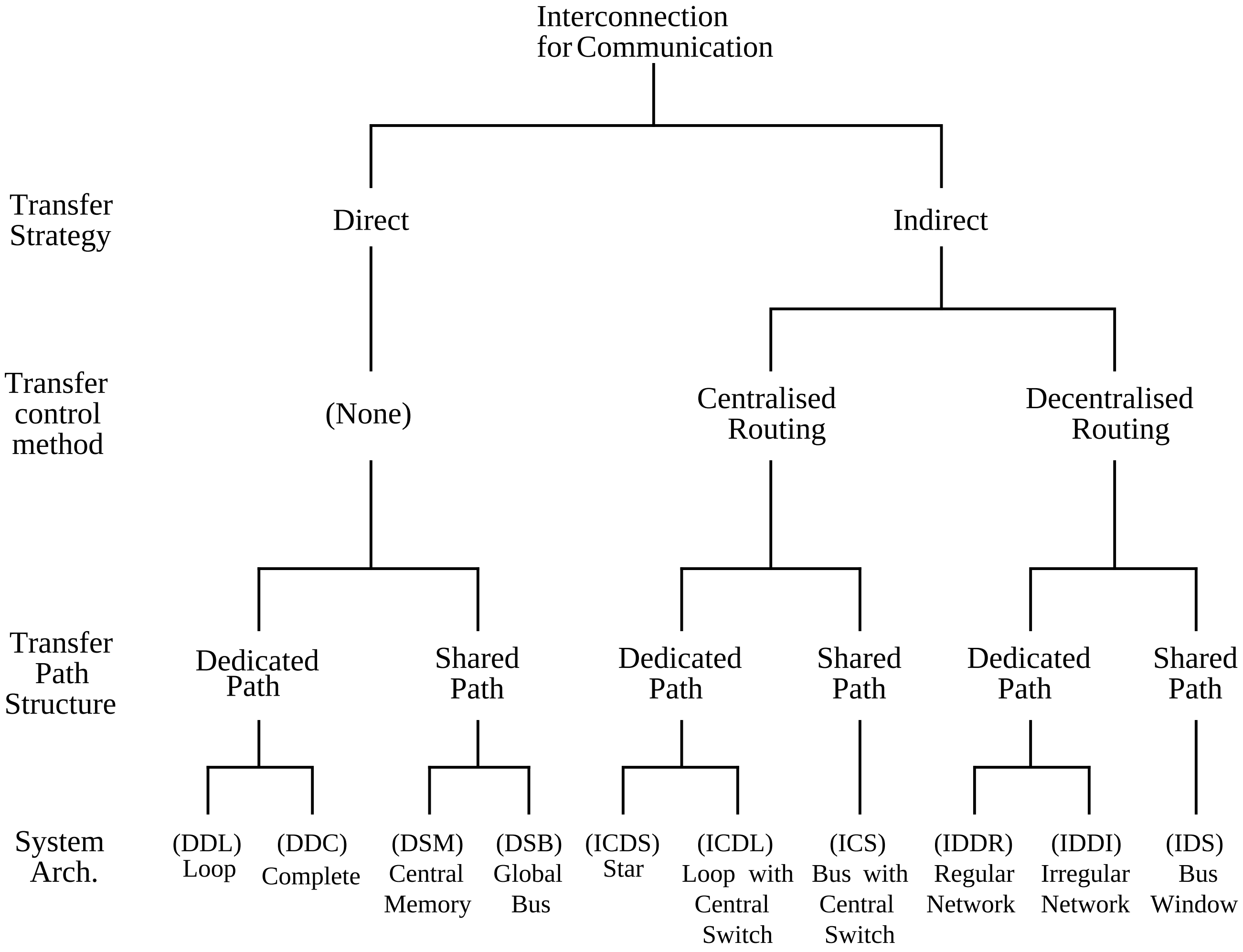


Figure 0.1: Anderson and Jensen's Distributed Computing Model

The second classification paper, by Enslow[35], describes a three dimensional space, reproduced in figure 0.2. Using this, he defines a distributed system as the area which has been shaded in the picture. The parallel systems studied here technically cover the entire plane labeled *multiple processors*, though in practice it is those with *master/slave dynamic* or greater control decentralisation.

These two classifications illustrate that it is the area of communications which distinguish a parallel operating system, the object of this study, from a distributed operating system. In the parallel systems under study, communications require routing, but in most distributed systems this is not the case. This is because distributed systems share a communication media, unlike parallel systems. This point has a vital impact on the scalability of a system, as will be shown in this thesis. The table in figure 0.3 lists some of the differences in communication between the two types of operating systems. These differences are important, and play a part in the design of the system.

This thesis will consider the use of a scalable parallel architecture to provide a general purpose computing environment. This will be achieved by providing a scalable operating system, called *Wisdom*, on this scalable architecture, which takes the form of a mesh. A scalable system represents one which can be enlarged to meet the processing needs of its users, unlike conventional systems which must be replaced if more processing power is required. The scalability dealt with by this thesis covers two distinct areas. The first represents the ability to add more processors to a system, and so enable a program to run faster, but as will be shown there are factors which limit the extent to which this can be done. The second area represents the ability to add more processors to support an increased number of users. The ability to expand, rather than replace, is the main advantage of a scalable system and provides the motivation for investigating the feasibility of such systems.

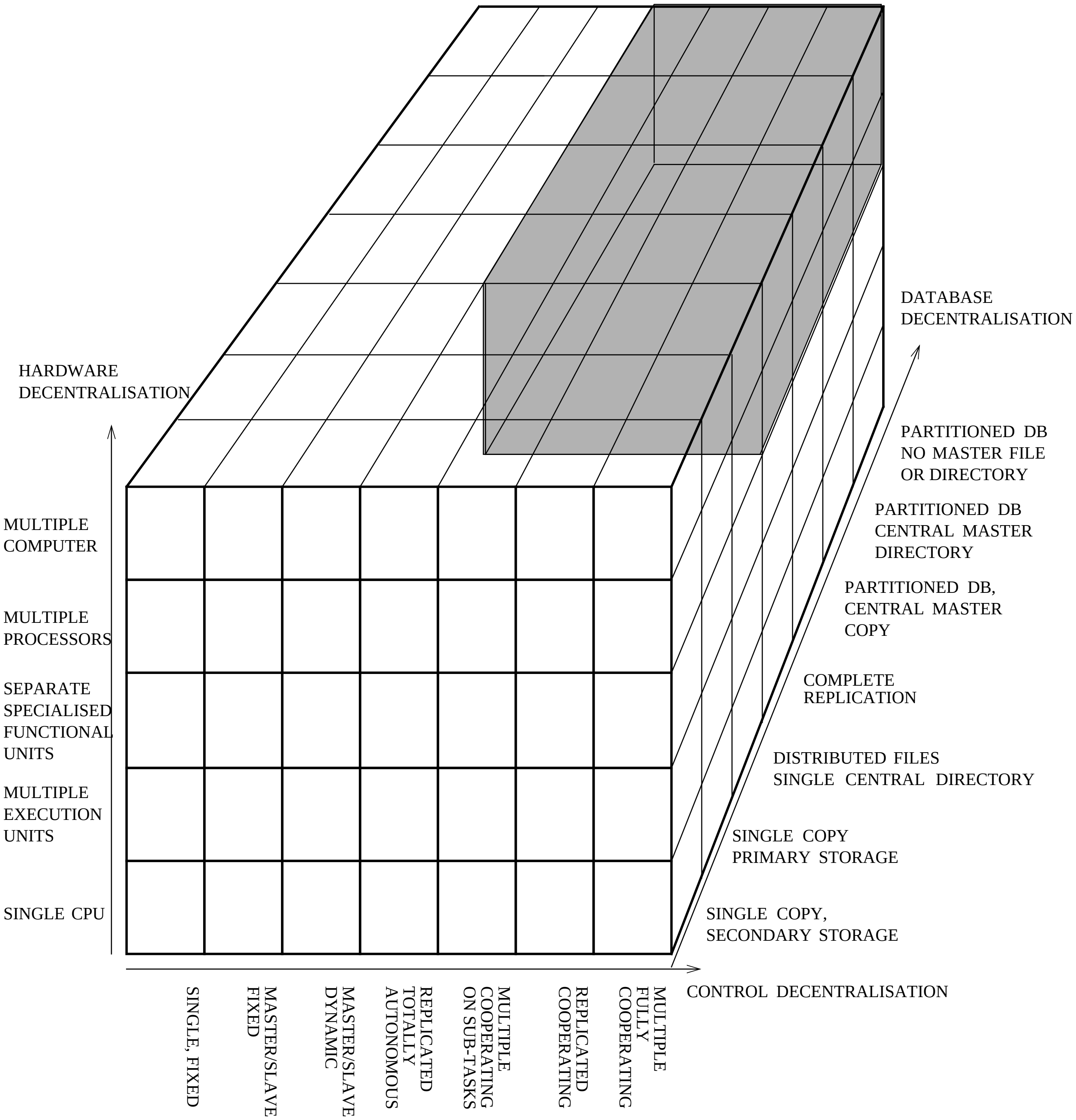


Figure 0.2: Enslow’s Distributed Computing Model

Objectives of this Thesis

The goal of the research undertaken, and described here, is to find a method of building an operating system that could be placed on scalable hardware, and to allow user programs to make full use of this hardware. Therefore the objective of this thesis is to test the hypothesis that *parallel array architectures can be used to provide a general purpose computing environment*. The operating system provided, *Wisdom*, therefore must have the following characteristics:

- *Usability:* The system should provide an environment in which the user can program without need to supply details about hardware, and the environment should be left unaltered by changes in the underlying hardware.

	Distributed, Bus based	Parallel, point to point interconnections
Routing	Direct, No delay	Routing required, with its attendant problems (e.g. delays, deadlock, starvation) and overheads (e.g. tables, processing time)
Bandwidth	Fixed, potential contention problems occur when more processors are added with the eventual saturation of the medium	Increased as more processors are added to the systems
Broadcast	Yes	No (though software mechanisms exist)

Figure 0.3: Communication differences between distributed and parallel systems

- *Flexibility:* In a general purpose computing environment it is not possible to predict all the resource needs of programs and programmers accurately, therefore *Wisdom* must be flexible enough to support a range of applications.

- *Scalability:* As one of the goals is to use a scalable architecture, the operating system itself must be scalable – that is it must make use of any new processors without increasing the load on the original processors.

- *Exploit Parallelism:* As a result of scalability, the system must be parallel itself and encourage the use of parallelism. Otherwise, scaling the system up by adding more processors has no advantage since, unless the additional processors can perform some of the work, it will not improve performance.

In order to make the task of achieving a system with these characteristics manageable, the following objective was set.

- *Minimalist design:* Experience of other systems has shown that the more complex the system the more difficult it becomes to implement, to experiment with and to maintain. A minimalist design is one in which only the essential services are provided. Consequently these essential services, the foundations, have to be identified.

Thesis Structure

The thesis starts with a survey of relevant research and some of the distributed operatings systems built and developed to date. Although these differ from *Wisdom*, there are many similarities, and many important lessons to be learnt from them. The various common parts of these systems are described, and those necessary for a minimalist operating system are identified. From this survey, the three basic components of *Wisdom* are identified: routing, naming and load balancing.

Chapter 2 is concerned with the philosophy of *Wisdom*: how the system is design and built, and the ideas behind it. Both the conceptual or logical view of the system and the implementation view are described, as well as the way in which it is expected to be programmed.

The next three chapters describe the components that make up *Wisdom*, and represent the foundation of a scalable parallel operating system. Chapter 3 deals with the routing and communication aspects of the system, describing the Router currently used and its function; other software and hardware mechanisms that could provide the same service are also discussed and evaluated. This section also examines many network topologies and concludes that a mesh is the most appropriate structure.

Chapter 4 describes the Namer, the services it must provide, and the requirements it necessarily places on the system. A naming theory is presented which is used as a framework to describe the properties of the *Wisdom* Namer. Mention, too, is also made of the problems scaling can cause for certain naming strategies, and some important namers already in use are detailed.

The final basic component of *Wisdom*, discussed in chapter 5, is the Load Balancer. This chapter deals with the problems and advantages of load balancing and the various ways it may be implemented. The results of simulated load balancing are given, and the *Wisdom* Load Balancer is discussed, and results of its operation presented.

The penultimate chapter evaluates *Wisdom.* This is done by answering two questions (is it sufficient and is it scalable) using the results presented in earlier chapters. A comparison is also made with a commercial operating system which uses a similar architecture.

The final chapter identifies the areas that show the need for further work and draws conclusions based on the evaluation of the prototype *Wisdom*.

Terminology

This section describes the terminology as used in this thesis. It is intended to clear up any ambiguities that may exist with the use of these terms. Throughout this thesis the terms Router, Namer and Load Balancer when encountered with capitals refer to the components in *Wisdom* which provide the corresponding service.

A *process* is the lowest unit of parallelism, or, more accurately, pseudo-parallelism, that can exist within the system.

A *task* is a collection of one or more processes and is used as the unit of load balancing. It is an abstract system concept, with no syntactic realisation, and is the smallest unit of true parallelism.

A *processor* is the mono-processing building block of the system. It may execute many processes or tasks in pseudo-parallelism, but can never share a single process or task with another processor.

A *module* is one or more cooperating tasks which perform a service, or help in the performance of a service. A module may be built out of other modules; it may also be distributed over many processors.

A *channel* is a communication connection between two processes, tasks or modules. Where the channel is between tasks or modules, it may be supported by the routing module.

A *message* is a collection of data and/or channels sent from one process to another over a channel. The message may be forwarded by modules (such as the Router), or modified by modules (such as the Namer).

A *guardian* is the process which serves as the interface between a task and the rest of the system. It is automatically supplied by the system, and is charged with performing the necessary housekeeping for the task.

Chapter 1

A Survey of Distributed Operating Systems

History is philosophy from examples

Dionysius of Halicarnassus

Distributed systems can be considered to comprise of individual computers connected by some communication media. On to this hardware operating systems have been placed – the first, network operating systems, represent little more than many conventional operating systems, but with communication mechanisms included to allow limited sharing of resources, or access to a common single resource. Distributed operating systems represent an increase in the transparency of the network, to the extent that the machines appear as a single, mono-processor machine[86]. However, each individual computer still maintains its own kernel, though these are often smaller than in conventional systems as more of the kernel services are provided by user processes. *Wisdom*, and the hardware it runs on, differ from these models. First the hardware is viewed as a single machine with many processors. Secondly no individual processor is viewed as holding the operating system – it is shared among the processors, all of which execute parts of it, and so the operating system is truly executed in parallel.

Despite these differences distributed, and to a lesser extent network, operating systems are an excellent starting point for this research. They represent both the most recent work in operating systems and their structuring. Also, they illustrate the basic facilities that are needed, and the form that they should take, to enable a computer to be used in a general purpose environment – one upon which are placed many varying requirements and applications. By examining these operating systems it is possible to find the essential aspects of a general purpose operating system – to identify the needed foundations.

This chapter covers the research that has been done in distributed operating systems and other related areas. First covered are the motivations for the development of, and research in, distributed operating systems, and which of them apply to *Wisdom*. Next, a brief description of each of the major distributed operating systems research projects will be given. This is followed by a summary of the major points and lessons that have come out of the research described. Finally, the transparency that a distributed system needs to support is described.

Throughout this chapter references will be made to POSIX. This represents a standardisation of operating systems and the interfaces they provide; it is based around those services provided by, and the interface of, UNIX. There are several descriptions of this in various UNIX–USER journals[18, 19, 20]. Although still undergoing work the standard exists as IEEE P1003.1, and this is now being used to describe operating systems, i.e. they are now said to be POSIX compliant.

There are many other surveys of operating systems, but the paper by Tanenbaum[86] offers a detailed description and comparison of four major distributed operating systems – the Cambridge Distributed System, Amoeba, V, Eden – as well as describing network operating systems, and it is recommended for the interested reader. Many of the other major projects have issued books based on their experience, and these are ideal if more detail on the specific system is needed. Mullender's thesis[60] provides an excellent description of the various parts of distributed systems, particularly capabilities as used in Amoeba – mechanisms for accessing resources that are controlled and manipulated by the user programs themselves, and not the kernel.

1.1 Motivations for Distributed Systems

There are many motivations for the evolution of distributed operating systems, and authors have tended to disagree slightly upon these[27, 31, 67, 72, 86]. The following, however, are the most commonly cited reasons, though generally they are not cited individually.

- *Cheaper and more sophisticated VLSI*[31]: It is often economical to buy several small computers, than a single, relatively expensive computer. The presence of many small computers encourages the natural desire to be able to share them. Hence the growth of network operating systems for mini and micro-computers.

- *Incremental Growth*[86]: Departments and businesses tend to grow slowly, and often wish to increase their computing power in a similar fashion, e.g., add a new computer for each new person. This requirement can easily be accommodated with cheap computers and networks or distributed systems. Also, as the work undertaken becomes more complex, the power that is needed will inevitably become greater. However, as Tanenbaum states[86],

  > System architecture is crucial to this type of system growth, however, since it is hard to give each user of a personal computer another 10 percent of a personal computer.

- *Communication and Resource Sharing*[67]: These are not often grouped together, but if resources are being shared, then communication is present, and if communication is present then it follows that resources *can* be shared. Potentially not all of the computers will have a complete set of resources: only one might have a laser printer, one a mail gateway to JANET and so on. However users might not wish to use the machine which provides access to that resource, or even have to specify it, therefore the operating system should support transparent sharing of these resources.

- *Reliability and Availability*: This is an objective of many distributed systems but has not always not been reached. The aim is that of connecting many computers together so that it is possible to produce a system which will survive the failure of any individual component computer. The result, unfortunately, has often been a system that failed whenever an individual computer failed. However, some systems[2, 28, 72] are now available that are able to withstand individual failures, and this capacity for survival is now one of the major features of distributed operating systems.

All the distributed systems implemented deal with one, or more, of the above points. For example, Grapevine[80], and its successor Clearinghouse[66] deal with communication and resource sharing, Tunis[36] contends with incremental growth by allowing additional processors to be added as necessary, and

LOCUS[72] deals with reliability and availability with its replicated filesystem. Other systems such as Amoeba and V deal with all of these factors.

The aim of *Wisdom* is to provide a computing facility that can be increased in power to, potentially, a very high level by the repeated addition of cheap VLSI components; thus the motivations for *Wisdom* are the first two points listed above. These two factors imply that there must be communication and resource sharing, the third point mentioned. *Wisdom*, and this thesis, do not directly tackle the final point, reliability and availability. It is envisaged that this point would be dealt with by the addition of specialised software, or the replacement of the necessary modules with versions designed to provide the support needed.

1.2 Review of Distributed Operating Systems

This section describes the major distributed and network operating systems and analyses the facilities which the various systems support. These systems will be classified by their communication networks; the reason for this has been best stated by Waterman[79]:

> A parallel processor's performance and expandability are directly related to its interconnection or interprocessor communication architecture.

In short the communication network affects the efficiency and scalability of the system. There are many more operating systems, distributed or otherwise, than can be covered within this study, however most of the major projects are mentioned, together with their relevant lessons.

1.2.1 Rings

A ring is a single contiguous 'circle' of the communication media to which the machines in the network are connected. Since the mid 1970s, rings have been used as a basis for local area networks. There are two main types[63]: *token passing*, in which a token is passed between sites, and possession of this token allows a site to transmit information. The other is the *empty slot ring*, in which slots continuously travel round the ring, and any site which sees an empty slot may place information into this slot.[†]

With a token passing ring, adding more systems to the ring results in less available bandwidth per system as the ring is shared between the computers by means of the token. In an empty slot ring the available bandwidth is still shared, though adding more stations may slightly increase the bandwidth of the ring. However, this increase is so slight that the result will still be a reduction in the available bandwidth per machine. Although rings do not generally support broadcast messages, it is very simple to make this service available if necessary. Ring networks have the advantage that it is relatively easy to ensure a fair share of the available bandwidth is allocated to all machines. Since messages can only be placed on the ring when an empty slot is present, or the machine has the token, correct control of the slot or token can be used to guarantee reasonable share of the bandwidth. This is best illustrated by the performance of rings under saturation communication – the amount of communication taking place reaches the maximum that the ring can support, and stays at that level.

Charlotte

Charlotte[6] runs on a network of computers (twenty VAX 11-750s) connected by an 80 MBit/s token-passing ring. Its goal is to be a minimalist, yet sufficient operating system (i.e. it provides the functionality needed), which supports graceful degradation, and has been designed using the following axioms:

[†] The register insertion ring[42] is ignored since it has not been used for any distributed operating systems and can be construed as a version of the empty slot ring.

- Charlotte will run on a multicomputer
- Charlotte must support a wide variety of application programs.
- Policies and mechanisms will be clearly separated.

On the basis of these axioms several design decisions were taken:

- Processes do not share memory.
- Communication uses reliable, symmetric, bi-directional paths and capabilities.
- Absolute information (e.g. machine identifier, process number) is stored at each end of a communication path.

Charlotte relies heavily on its IPC mechanism and the basis of the system is support for this, together with the necessary process manipulation mechanisms. After the initial design and construction, load balancing was added to Charlotte[7].

Charlotte IPC is non-blocking (it takes the form of a request for communication), over duplex links. When an IPC request is made by a user process, it will take place asynchronously as a transfer from the sending to the receiving process. A user process can, for any transfer it has initiated, wait for the transfer to complete, or abort any request that has not yet started. User processes can also wait for messages on ‘any’ link. Bi-directional links are used as experience with other systems had indicated that uni-direction links have problems when there is a need to send reply messages (for example exceptions or errors) or to inform processes that the receiver has died or moved. Links can be sent to other processes using the standard operations – messages may even be sent to a link that is being transferred, and they will arrive at the correct place. In order to avoid problems of deadlock within buffer allocation, and to allow arbitrary sized messages, Charlotte provides no buffering (messages are transferred directly from the sender to the receiver).

Cambridge Distributed System

The CDS[63] uses a large Cambridge Ring as its communication subsystem, and is more of a network or processor pool manager than an operating system, but provides many of the services that a full distributed operating system does. It consists of a processor pool and a group of servers. When a user logs in, a processor from the pool is assigned for the whole of the session; the servers exist on dedicated processors and are only contacted when their services are required (e.g name service, or boot service).

Three protocols have been built on top of the transfer mechanism provided by the ring hardware, and these form the basis of the CDS IPC primitives. The first and simplest of these is the *packet protocol* that comprises of up to 1024 mini-packets that the network directly supports (each mini-packet contains two data bytes). Next is a *single-shot* protocol which is akin to an RPC – that is a message is sent and a reply is awaited. The last is the *byte stream protocol* which implements a full duplex, virtual circuit. This gives a choice between packets, RPC-like mechanisms and a virtual circuit style service.

All the services are located by contacting the name server, which is, itself, a service. The name server maps ASCII names on to a network address for the service, and supplies the protocol which the service expects. There are several name spaces in the CDS – those implemented and maintained by the name server, and those provided by the file servers (and other services).

PULSE

The aim of PULSE[44] was to network together a collection of powerful personal computers (in practice, several LSI-11's) and provide a UNIX-like operating system. Each of these computers should function independently in the absence of the network, and yet provide a single, hierarchical network-wide filesystem in the presence of the network. One of its major objectives was to use the Ada language to implement the operating system. PULSE was structured as a collection of layers, each building on, and hiding, the previous to increase the services provided, and these are shown in figure 1.1.

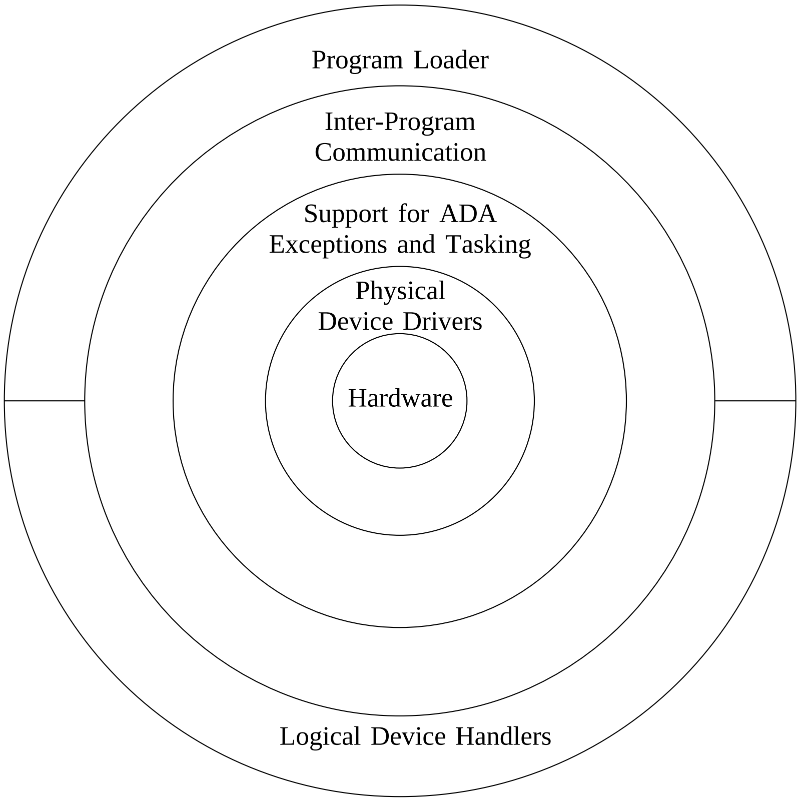


Figure 1.1: Implementation Layers of the Pulse Kernel

Pulse supplied a uniform IPC interface to both the kernel and user programs. For dynamic communication paths to be set up, processes had to be able to acquire communication capabilities from other processes. Any process could provide a service; it did this by binding a 'medium' into the system name space. A medium was an Ada task with entries for sending and receiving messages, and hence the process could send and receive messages from other processes via the named medium.

DEMOS/MP

DEMOS/MP[58] is a distributed, multiprocessor version of the Demos operating system which began life on the Cray 1 computer. Its aim is to provide a software base for the OSMOSIS (Operating System for Making Operating System Implementation Studies) distributed systems project, and allow experimentation in the design of distributed systems. The current version exists on dual Z8000S connected by a token ring, though other versions exist.

The basic elements of DEMOS/MP are processes and links. Links are IPC connections, maintained by the kernel, which support uni-directional communication with the following properties:

- messages sent from one process to another arrive in the same order sent.

- no messages are lost.
- no messages are duplicated.
- a link can be an IPC connection to a process on another machine

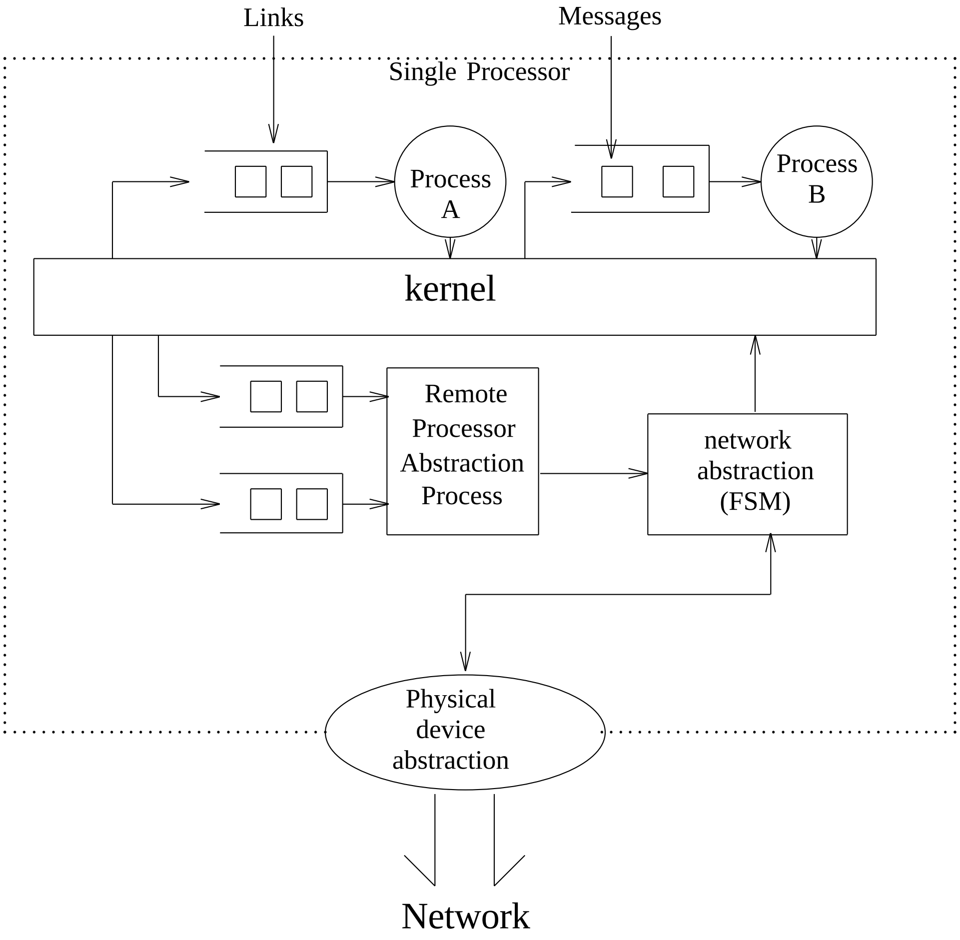


Figure 1.2: Message paths in DEMOS/MP

Links are used to deliver messages to user or system processes or to the kernel. A message sent to a link with the *DELIVERTOKERNEL* attribute will be delivered to the kernel on the machine where the process that owns the link resides. Messages intended for remote processes are passed by the kernel to a process which provides the remote processor abstraction – this arranges for the messages' delivery by passing them on to the network, as shown in figure 1.2. A link consists of a globally-unique process identifier and the last known location of the owning process. If a message is received for a link corresponding to a process that has since moved then a message is returned giving the new location. This message is used to update the last known address for all links pointing to that process on the kernel that receives this information. Because links are one-to-one connections, only one update message is required per processor that is connected to the process. Links can also be used to perform asynchronous data transfers, as each has an area of memory associated with it into which the holder of the link may write. As links are network wide addresses, the system, above the communications level, need not concern itself with multiple machines.

The design of the link was deliberately made to simplify process migration, with the result that process migration was added very quickly (in, approximately, one month[73]) with no changes to the system. Migration is controlled by a system process which moves the kernel data structures, the code and data of the task, and marks the links as moved, labeling them with the new address of the process.

1.2.2 Bus

A bus-based architecture is one in which all the processors share a common data and address bus (such as VME or Multibus). This inevitably produces contention problems when two processors wish to use the bus at the same time for example, to read or write to memory or a device. One common solution, however, has been to give each processor its own, private, local memory that is not seen from the system bus, and that, when accessed, does not require the use of the system bus.

The advantage of this type of architecture is that it provides shared memory, which can greatly speed communications and task migration. But care must be taken when designing such systems as excessive use of shared memory will badly degrade the system. Although not strictly speaking a broadcast media (there are only ever two participants in any communication – the source and destination), it can be made to act as broadcast by placing messages in common global memory which is accessed by all of the connected processors.

Tunis

Tunis[37] from the University of Toron to is a UNIX compatible operating system, the hardware for which consists of several processors, each with local memory, connected by a bus – the Multibus, as defined in IEEE-796. Communication occurs via shared global memory, and bus contentions are kept to a minimum by using the local memory on each processor as far as possible (accessing local memory does not interfere with the shared bus). The intention is to support incremental performance by the addition of more processors to the system. Tunis is implemented in Concurrent Euclid, a Pascal-like language, which supports communication and synchronisation via monitor constructs, and the *signal* and *wait* primitives. These primitives are provided through a small, assembly-language, run-time kernel executed as a monitor. There is only a single kernel (not one for each processor), and the various processors gain control of this via atomic *test-and-set* instructions, to ensure the monitor status of the kernel and the consistency of its data structures.

The internal structure of Tunis, shown in figure 1.3, consists of a hierarchy of five main modules between which control is passed in a strict order defined by this hierarchy. At the highest level are the user processes, which execute in envelopes. Intertask communication in Tunis is achieved either via UNIX pipes, or by the mailbox module from within a single Concurrent Euclid program. It is estimated that the prototype could be improved so that the time spent executing the kernel (the centralised part of the system) would be about 3% per processor, giving a maximum number of about thirty-three processors in a single system.†

The performance and scalability of Tunis are limited as a single kernel is used to support Concurrent Euclid. Originally Tunis did not perform load balancing, although it did allow task migration on the `exec()` system call. A newer version provides load balancing at any point in a process' lifetime. Unfortunately initial performance studies have shown that the increase use of the shared bus this entails limits the size of a single system to about five or six processors. This is because the processes are held in local memory, and when they are migrated they must be copied out of this into the local memory of the target processor – this copying places a heavy load on the shared bus.

1.2.3 Ethernet

The most common distributed systems are those based on ethernet[57]. This is largely because of the abundance of necessary hardware and its relatively low cost.‡ The speed of ethernet systems is often

---

† The Balance multiprocessor computers use a similar architecture, and execute a version of UNIX. The maximum number of processors that can be placed in this machine is thirty[15].

‡ Bus-based systems and the very early Cambridge rings required the investment of much time and money in the development of, at least, some hardware.

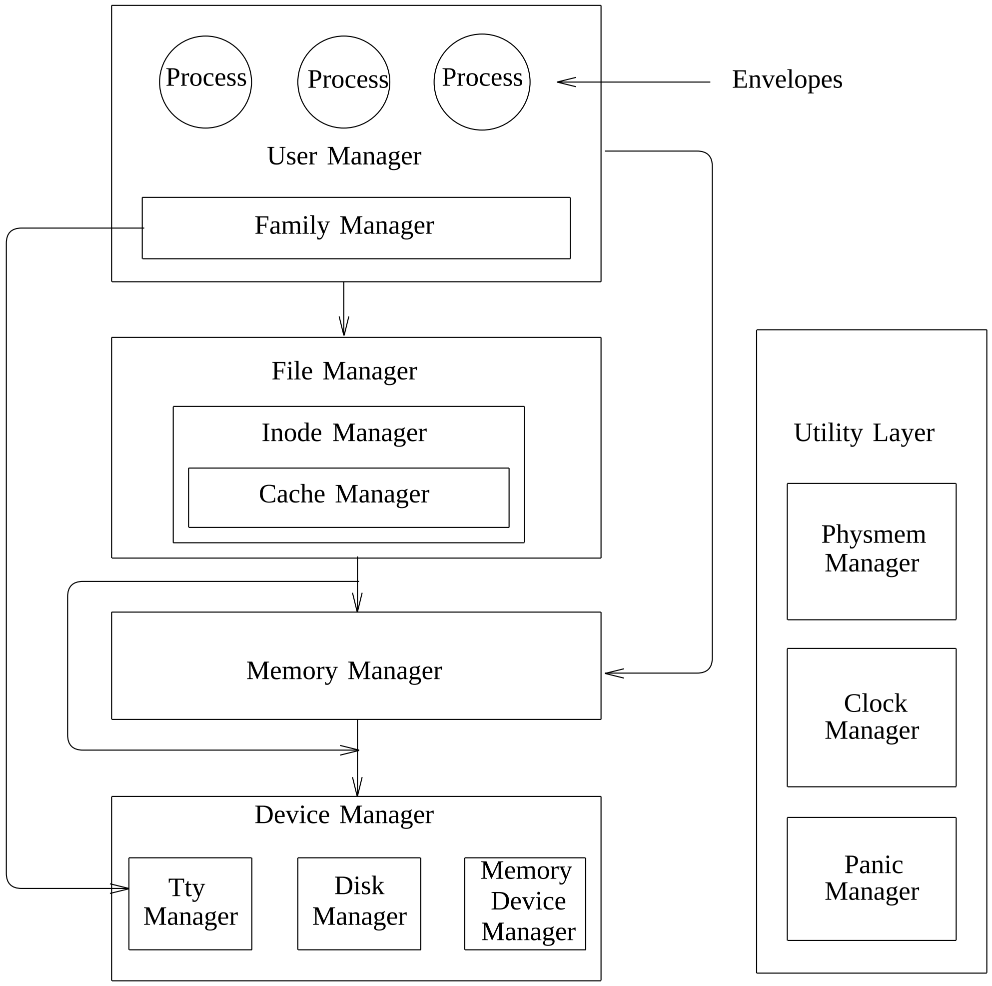


Figure 1.3: The Internal Structure of Tunis

limited as much by the software as by the speed of the media. There are two main reasons for this, firstly it is generally necessary to copy the message into local buffers before sending it to its destination. Secondly protocols are used which allow heterogeneous machines to communicate. Ethernet is a carrier sense multiple access with collision detection (CSMA/CD), which means that it will only send a message when nothing else is using the media. If, when it starts to send a message, it finds that the media is in use, it will cease transmitting and wait a random period (i.e. it will back off) before retrying. The result of this is that the behaviour of the media is good with low loads, but, as loads become higher, its performance degrades, with its saturation performance being very discouraging.

The V System

The V System[27] from Stanford University has been designed for workstations and server machines connected by a local area network, and has a large user community at several sites across America. The current system is run on such systems as the SUN network, but other machines are also supported, like the DEC firefly (a multiprocess microVAX), and the VMP (a shared memory-multiprocessor computer designed at Stanford University). Akin with most systems, V provides a UNIX-like interface, though with some additions which deal with IPC and process migration. Its aim is the management of a cluster of workstations and servers, but providing the resource and information sharing facilities of a single-machine. The tenets upon which the system is implemented are that its message-passing is fast, that the system is defined by protocols, not software (i.e. anything that ‘speaks’ the protocols can be part of the system), and, finally that a small, minimalist kernel is provided.

The IPC provided within V is aimed at supporting a very high speed remote procedure call, via three operations: `send`, `receive` and `reply/return` (which are derived from Thoth[26], an earlier operating system also by Cheriton). This speed is achieved by a tailored kernel, and by making the `send` operation block waiting for a reply. There is also support for short, fixed-size messages (32-bytes), with optional data segments (16 Kbytes) to improve performance: many messages (in V, as with Thoth, about 50%) need only the 32 bytes.

In addition to IPC, the kernel, on every node, contains support for process, memory and device management, and a clock. All of these are implemented as modules which register themselves with the IPC service, and so may be contacted by remote, as well as local, processes. They are implemented on each node for efficiency reasons as local services are faster and simpler than remote ones. Processes have single ports on which they may send and receive messages, so the IPC identifier can double as process identifiers. These entity identifiers are used to name processes, groups of processes and IPC points.

A major part of V is its naming facility[25]. This is based on a three-level model: character-string names, object identifiers and entity identifiers. Object managers `mount` themselves in the global name space by using a unique name prefix for their ‘directories’, and then adding themselves into the *name handling (process) group*. Object managers are located by performing a multicast with the character-string name of the object desired to the name handling group, i.e. sending a ‘do you know this name’ to all objects that have registered themselves as object managers, to which only the appropriate manager replies. To reduce the multicasts, all processes maintain a cache of the name prefixes so they can determine which object manager to contact without having to broadcast whenever a name is looked up. These prefix tables are inherited from the parent, together with other information, when a new process is created. Once an object is located, an object identifier is returned. This contains two items: an IPC level identifier for the manager, and an identifier. This identifier is used within the object manager to specify the object originally requested.

Eden

The Eden system[17] is implemented above UNIX on a network of SUNs, and supports an object oriented processing environment. It is intended to run on any network of personal workstations, i.e. there are no specialised servers or processor pools, and the workstations combine to provide the services. The language used, called EPL – Eden Programming Language, is based on Concurrent Euclid. Each object that makes up the system is implemented as a single UNIX process.

Within Eden, all objects have the following characteristics:

- Location independence: objects need not know where other objects reside.
- Mobility: objects can move during their lifetime without affecting the system.
- Autonomous: they exist and operate independently of other objects.
- Accessed via capabilities

Objects may also checkpoint themselves, that is they may request Eden to store information about their state so that they can be restarted in case of failure. Objects use an RPC-like mechanism, called ‘invocation’, to obtain services from one another. All Eden objects are considered to be active, although they may be made passive by the system in order to conserve resources, in which case they are automatically activated when needed. Objects can be considered to comprise of two parts: a concrete Edentype which describes the interface to the object (the operations it performs), and a data part which itself consists of two parts. The components of the data part are a long-term state representing the object, and a short-term state containing information such as local data from invocations. The Edentype is user-definable, hence programmers can define the operations that their objects will perform.

The Eden system has the following characteristics:

- A single, integrated, uniform name-space
- Secure objects by the use of ‘unforgeable’ capabilities
- No error recovery mechanism, though where necessary this may be built by use of the checkpoint facility.

The Eden system has been used as a base for many research studies, including load sharing[34, 48]. The load sharing studies have fallen into two categories: first theoretical comparisons of the various methods, and secondly applications of the results to the workstation environment of Eden. These will be mentioned in chapter 5.

LOCUS

The LOCUS system[72] runs on a network of heterogeneous machines connected by an ethernet. It is a UNIX-compatible distributed operating system whose main goal is to provide a network wide filesystem, which exhibits transparent, automatic data replication. It also supports high reliability functions, such as nested transactions, as well as transparent, remote process execution, and dynamic load balancing.

File replication is provided by dividing the process of file accessing between three processes: the user site (US) which is the process accessing the file, the storage site (SS), where the file is stored and the current synchronisation site CSS). The CSS selects a SS from those that maintain copies of the requested file and enforces synchronisation of files when updates occur (i.e. it propagates the necessary updates to the relevant SSs). These three processes can exist on any site, i.e. they need not reside on the same physical processor, although LOCUS performs placement optimisations for some combinations.

LOCUS does not support any additional IPC mechanisms over and above those provided by the modern UNIX systems. The primary mechanisms used in LOCUS are pipes – those that are named (sometimes called fifos) and the original unnamed version. Pipes are maintained using the replication and synchronisation mechanisms of the filesystem so that the semantics of pipes are exactly the same as on a conventional uni-processor UNIX system.

Remote process execution is provided in several ways. When a process is created with `fork()`, it may be created locally, or remotely. When `exec()` is used, the code which is to be loaded and executed may be so done locally or remotely. Two new calls have been added: the first is `run()`, which is a combination of `fork()` and `exec()`; the second is `migrate()`, which the calling process uses to perform explicit load balancing. In all of these cases, the information that decides where a process is to execute is set by that process, or in the case of a `fork()` by its parent, using the call `setxsites()` which supplies a list of possible execution sites to the system. Some utilities have been provided to allow the manipulation of this list for the explicit remote execution of programs. When tasks are migrated, to reduce the data that is transferred, pages are moved on demand only. The advantage of this policy is that, in the UNIX environment where most `fork()` calls are followed by `exec()` calls, relatively little of the code of the parent needs to be balanced.

Amoeba

Amoeba[84] is an object oriented distributed operating system. An Amoeba system, as shown in figure 1.4, is a network (typically an Ethernet, or a Pronet token-passing ring) which connects workstations, used for editing and other interactive tasks; a processor pool, used to supply computing power for non-interactive tasks; various servers, such as file systems; and gateways to wide area networks which link together Amoeba systems[76].

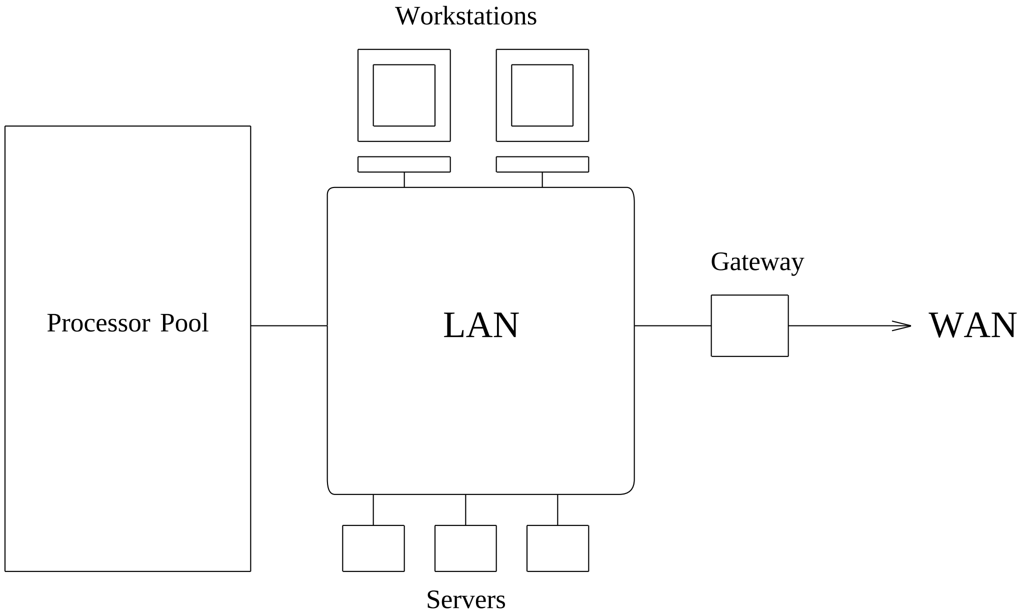


Figure 1.4: The Amoeba architecture

All objects in Amoeba are protected by unforgeable capabilities, which are 128 bit numbers comprised of four parts, shown in figure 1.5:

- server port; specifying where the service is, and chosen by the server (i.e. it can contain any value, subject to it being unique within the machine).
- object number: an internal, to the server, number used to identify the object.
- rights field; used to specify the operations this capability confers.
- check field; used to prevent forging of, or tampering with, the capability

Whenever a client wishes to contact a server, the client passes the capability, together with the necessary data, to the kernel, which then looks up the server port in its local <port, network address> cache. If the address is found, the kernel sends the message to that address, together with the capability. If it is not, the kernel broadcasts the port to the network, and awaits a reply, which is then placed in its cache and used to forward this, and future, messages. All communications between servers and clients are performed as request/reply transactions, effectively remote procedure calls. To enable the reply, all messages have a reply port sent with them. All these communications are reliable.

48	24	8	48	Number of Bits
Server Port	Object Number	Rights	Check Field	

Figure 1.5: An Amoeba Capability

Security can be provided by the (currently theoretical) function boxes (F-boxes), which act as one-way encryption routines. The port a server selects is called the *get port* since it is from this that messages are got, or received. When a process sends messages it uses a *put port* since that is where messages are put. These two ports are related by the one way function the F-box implements: when this function is applied to the get port the put port is produced. A server sends out the get port, which is automatically encrypted by the F-box to produce the put port, which the client receives. When a message requesting the location of a port is broadcast, all the F-boxes compare that port with the ports they have encoded, and reply with their network address if there is a match. The F-box acts as an interface between the computer and the network.

MOS

MOS[11], or MOSIX as it is now called, has evolved from a central computer which sent work to satellite processors, into a cluster of identical microcomputers, connected by a local area network (originally a 10Mbps token passing Pronet, but now an Ethernet). Its aim is to integrate a network of identical, loosely coupled processors into a uniprocessor-like UNIX environment. MOSIX is characterised by:

- Adaptive load balancing
- Replicated kernel
- Network Transparency
- Decentralised control
- Autonomy
- Dynamic configuration
- Unified file system

The dynamic load balancing[9] uses the MOS form of ‘information scattering’ – a mechanism which passes information between a random subset of the machines[2]. Load balancing only takes place on running processes, and, because of the high cost of migrating such a process, it only moves those which are expected to continue running for some time. As it is not possible to know a process’ life expectancy, the system uses a heuristic: the older a process, the longer it is likely to exist. The cost of migrating a process arises not just from the expense of moving the code, but also from the need to move the data segments, kernel data structures, and to update certain kernel process pointers. In the UNIX model, there is no equivalent of a ‘new’ process since the primitives provided are `fork()` and `exec()`. For this reason it was decided only to migrate running tasks, although it is acknowledged that the ability to move new processes would appear to be adequate.

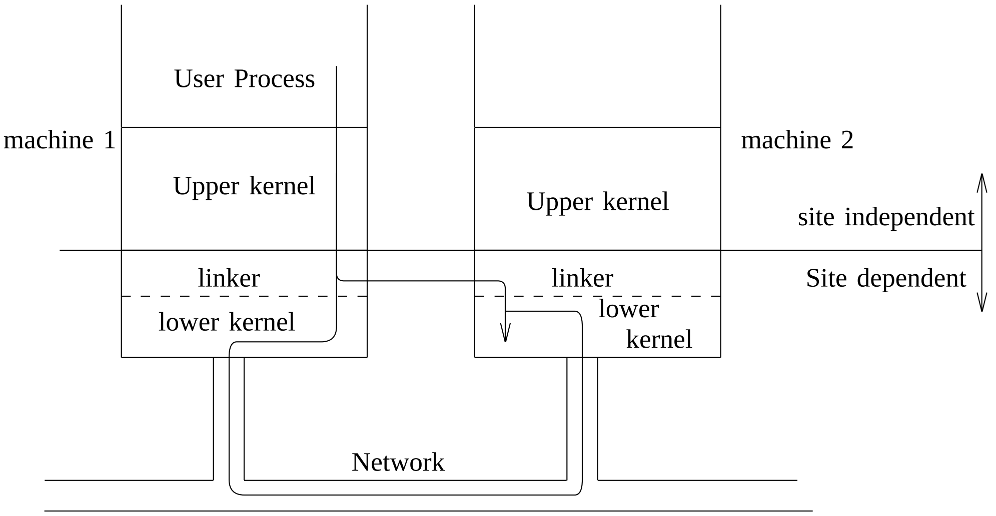


Figure 1.6: Structure of the MOSIX kernel

The replicated kernel is divided into two levels: the upper kernel, which is site independent, i.e., it knows nothing about which machine it is on, and the lower kernel, which deals with the machine on which it is executing. The interface between the two parts is via the linker which determines the machine to which a request is being made (from the information in the kernel structures) and then either executes the request locally or performs an RPC to the appropriate remote machine; figure 1.6 shows the structure of the kernel and an example of a remote machine structure being accessed over the network. This mechanism allows MOSIX to remain largely ignorant of the network. Since every machine possesses all the parts of the kernel necessary to maintain its own machine, it is capable of functioning independently of the rest of the network should the network, or individual machines, fail.

Mach

Mach[12] is a descendant of RIG[50], Accent[75] and Spice[88]. It is a multiprocessor version of Accent and UNIX, which runs on primarily VAX and SUN computers. Mach supports both tasks (heavy weight processes with a protected memory space), and lightweight threads (lightweight processes that exist within a task and share their parent's memory space). It is aimed at supporting both tightly-coupled processors, i.e. those sharing memory, and loosely-couple processors, those with no common memory, an example of the envisaged network is shown in figure 1.7. The primary goals of Mach are to provide simple, high speed, communication primitives in the kernel, and to support binary compatibility with UNIX.

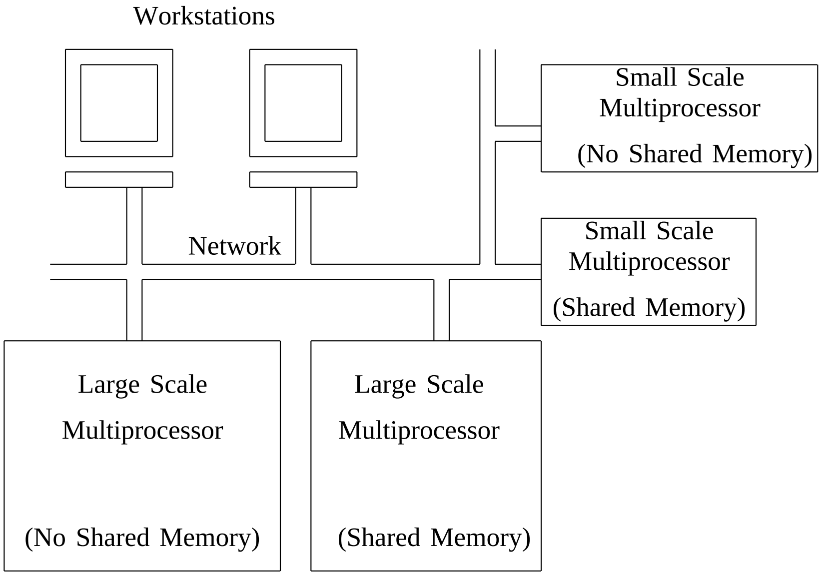


Figure 1.7: Typical Mach network

The IPC mechanism in Mach is capability based, and designed to preserve transparency and protection across network boundaries. The capabilities in Mach are similar to those in Amoeba: they both use ports, but in Amoeba capabilities are controlled by the processes themselves, whereas, in Mach, by the kernel. Logically a port is a finite queue of messages for a task, which may have an arbitrary number of senders, but only one receiver. Messages, which are sent to ports, can be transferred either synchronously or asynchronously, unlike Amoeba, where sending always blocks awaiting a reply. Since the message queues may become full a task is given three options on how to handle this: wait until there is space in the queue, receive an error code back from the send message operation, or have the message queued for delivery anyway, but to guarantee not to send more messages to the same destination until the kernel informs the task that the message has been delivered.

1.2.4 Regular Meshes

All of the above connection media (rings, buses and Ethernets) have a common attribute: all nodes share a common communication media. This means that, at some point, saturation of the media is reached and it is not possible to, or more accurately it is pointless to, place more nodes in the system. By comparison, regular meshes do not share the same communication media, as they are point to point networks, hence as more nodes are added, the total communication bandwidth is also increased. Unfortunately, these networks require message routing, which may place an overhead on all the nodes and are unable to perform broadcast style communications (though this can be done using software algorithms). Care should be taken in designing systems using meshes as it is possible that a single point to point connection might become a communication bottleneck (i.e. all, or a large proportion of, messages might have to pass through it).

Although meshes can be built up using conventional processors, with additional hardware to provide the point to point connections, many systems using this style of communication have instead been designed using the INMOS transputer[53]. Of these, only Helios will be described here, although several others do exist: PARX[49] which is based around Tanenbaum's MINIX[85] message passing, UNIX look-alike, PC operating system, transIDRIS[46] – a POSIX conforming operating system which uses explicit user commands to place programs on neighbouring transputers; Trollius (formerly called Trillium) is low level operating system that supports a portable multiprocess structuring, much as in Chertion's early uni-processor system, Thoth[24], and uses the standard UNIX system as a front-end.

Helios

Helios is a commercial operating system for networks of transputers. It is POSIX conforming and provides a degree of load balancing by the use of various specialised programs. It supports a shared library mechanism for those libraries that are commonly used (currently those included in the nucleus). The protection mechanism employed is based on the Amoeba capabilities, described earlier in this chapter.

All nodes which support Helios[†] must load the nucleus and execute the appropriate parts. The nucleus consists of six parts:

- Kernel: message passing, memory management, process creation and scheduling
- System Library: 'system call' interface, general server protocol interface, heap management, resource tracking and environment enquiry.
- Server Library: routines needed by all servers
- Utilities Library: various common library routines used by programs
- Processor Manager: process creation and deletion, signal delivery, environment enquiry, name table management, server location.
- Loader: interprets and loads code images, binds programs to shared libraries.

The only supported method of communication is the message passing handled by the kernel. The IPC service provided is one of zero or once delivery over what is, almost, a virtual circuit. When the system is forced to destroy a message, it tries to generate a new message stating that the destruction has occurred and to returns this message to the sending process. However, it is possible that this message may also be destroyed. The various library routines provided support automatic retry in the case of message failure or destruction. Messages are sent to ports which are location independent and obtained from name lookup (see later). A message consists of a header that gives the destination port, an optional reply port, a function code or result value, various flags, and the sizes of the optional remaining parts of the message, the control vector and the data vector, as shown in figure 1.8. Message routing is performed by delivering a message to a port which either points to the task for which it is destined, or to another port on a neighbouring transputer to which it should be delivered. Thus, messages follow a chain

---

[†] It is possible to have 'naked' nodes in the network which do not run Helios, or support any of its functionality, but do run raw transputer code. However these processors must be leaves in the network map. Helios 'ignores' these processors except when specifically told to communicate with them, or run programs on them.

of ports to their destination; and, if a reply port is present, whilst this chain is being followed, a corresponding chain is built up for the reply port.

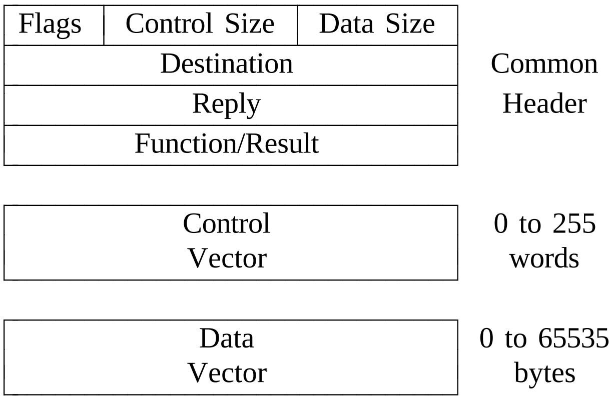


Figure 1.8: Helios Message Structure

Name lookup is performed by consulting a set of local tables in order to find the name. If that name is known then the system has a port through which it can gain access to the service corresponding to the name. Otherwise it floods the mesh with messages requesting the service to respond. When a server responds its name is added to the cache, and the connection passed to the process that initially requested the service. The flooding is controlled by each processor maintaining a list of the last few lookups (currently 32) it has seen, and it discards any that it has already seen. A result of this method is that interrogations only yield those services that have already been used (i.e., it is not possible to determine all the services provided, only those so far used), and so other services may exist besides those that can be seen. When load balancing takes place it is possible to receive two very different views of the name space, as each processor has its own cache.

Load balancing is performed by several programs. Any set of tasks which require balancing have their interconnections described in the component distribution language (CDL), which is then compiled (or interpreted) into a CDL object file, and passed to the Task Force Manager (TFM). The TFM compares the map of the tasks with its own map of the network and performs a heuristic-based, near-optimal mapping of the requested tasks on to the available hardware. The shell, in this instance, has been modified so that, when requested to generate pipelines, it will create a CDL object file used by the TFM to achieve an appropriate balance.

1.3 Summary

The above has described several distributed systems; the various design decisions and the lessons learnt from them are now studied. Some of the characteristics of these systems are summarised in figure 1.9.

1.3.1 UNIX

The summary in figure 1.9 shows that many systems have opted either to emulate, or to support fully, the UNIX or POSIX model. Although this provides a well known and well liked system, with large amounts of available software, it does impose several limitations:

- There is no ‘create new process’ system call at which point load balancing may be performed with little or no overhead, i.e. where the task has no associated context.

- There is only a limited set of IPC available under UNIX: primarily pipes. Providing the full semantics for multiple readers or writers is not trivial, and LOCUS goes to considerable lengths to preserve this.

OS	UNIX	Media	IPC	L.B.	Naming
Charlotte	Yes	Ring	Reliable bi-directional	Yes	Filesystem only
CDS	No	Ring	Datagram, virtual circuit and RPC-like (all reliable)	No	Server for all services, disjoint from names supported by services
PULSE	Yes	Ring	Ada-style rendezvous (reliable)	No	Single global namespace within which services are provided
Demos/MP	No	Ring	Uni-directional virtual circuit, guaranteed delivery, (unreliable versions available for efficiency)	Yes	Filesystem only
Tunis	Yes	Bus	Unix pipes (reliable uni-directional) or mailboxes from *within* concurrent Euclid programs	Yes	Filesystem only
V	Yes	Ether	RPC like-reliable, based on send, receive and reply-return primitives	No	Single global namespace with broadcast location for servers
Eden	No	Ether	RPC-like, invocation of objects	Yes	Filesystem only
LOCUS	Yes	Ether	Unix Pipes (named or normal)	Call	Global replicated
Amoeba	No	Ether	port-based RPC-like and reliable	No	Single global with broadcast location
MOS	Yes	Ether	Unix-based	Yes	Filesystem only
Mach	Yes	Ether	Port-based with capabilites (like Amoeba, except maintained by kernel)	No	Filesystem only
Helios	Yes	Mesh	Uni-directional with reply, not reliable, though all user routines provide reliable send and receive	Yes	Global namespace, though not necessarily uniform; broadcast (flood fill) location

Figure 1.9: Summary of characteristics of various distributed operating systems

The solutions to these problems generally have been to add extra functionality by the creation of new system calls, or to accept the restrictions that have been imposed and work within them as best as possible.

Many of the problems of IPC stem from using either BSD UNIX or System V UNIX as the reference, both of which provide mechanisms of limited flexibility, or impose requirements on the hardware (such as shared memory in system V UNIX).† However, the AT&T Bell Labs research versions of UNIX (Eighth Edition and beyond) offer improved facilities and semantics through the use of streams[77].

The problems encountered by load balancing – specifically that there is no place at which a process exists with no associated context – are inherent in the design of UNIX. When a process is created (via `fork()`), it inherits an environment (connections to the outside world) from its parent, and the UNIX kernel is responsible for this inheritance. If, when the process was created, the parent passed any needed context to its child rather than using automatic inheritance, then many of the problems of load balancing are alleviated. In fact, when a program forks, it generally only sets up the environment before executing a new program; in which case, this could be replaced by a parent *passing* to the child the necessary environment instead of creating it first. The remainder of the occasions where `fork()` is used could be replaced by the use of threads, or lightweight processes.

1.3.2 Communication Media

Most of the systems described use a broadcast media – one in which it is possible to send messages to all processes, or processors, with ease. The presence of broadcast facilities has manifested itself in the design of many location policies, such as the way V locates a resource corresponding to a name. Unfortunately, in a system that performs broadcasts, there are limits on the number of machines that may be connected together, and systems such as V and Amoeba have been forced to look at methods to solve this. The solutions have involved a combination of caches (to reduce the number of broadcasts), connecting networks together and arranging that broadcast messages are forwarded to the correct network and querying specific name servers.

In a mesh network, it is possible to simulate broadcast by flooding the entire network – each processor sends the message to all of its neighbours, who duly behave likewise. To control this, every processor must remember which messages it has seen, and not pass on these messages, otherwise the network will become totally swamped. Although this technique provides broadcasting, it also places restrictions on the scalability of the system (each processor must remember a sufficient number of the last messages that were broadcast to prevent swamping), places extra load on many nodes, and generates many messages.

1.3.3 Interprocess Communication

Almost all the versions of IPC provide the user with a reliable mechanism, even if the lower levels are not always reliable. However, there are several cases where the information to be transmitted is not ‘important’, and so some systems offer an unreliable, but far cheaper, form of communication. Typically this contains housekeeping information such as the load or the active users; it often makes up a large part of the communication traffic[13].

The systems are evenly divided between bi-directional and uni-directional communication, though it has been remarked that bi-directional communications simplify system and program design as it allows errors and results to be returned. Several systems provide bi-directional methods over a uni-directional system by incorporating, in every message, an IPC channel pointing back to the source. This allows the RPC-like mechanism to be built by sending out a message containing a channel, on which a reply is awaited.

More complex IPC methods, such as full RPC or the Ada rendezvous, have, unfortunately, been characterised by bad performance and low speeds. Given the importance of speed in the IPC mechanism, as noted by Cheriton[23], systems have generally tended to avoid these in favour of faster methods. The lack of functionality is not a major problem, as these complex mechanisms can be constructed using the existing systems when needed; it should also be noted that often the complexity is not required.

---

† These problems have arisen largely through the age of the systems used as a reference, and the fact that much of the development for them assumed that they would run on a uni-processor system.

1.3.4 Load Balancing

When load balancing is present it has generally been added after the system was originally built, though the original design sometimes has taken account of load balancing, or task migration, making its subsequent addition far easier. As has been already mentioned the UNIX model imposes severe limits upon the efficiency of this, and results in some systems either not attempting it, or limiting it to explicit utilities which select the best site to run a program. Various studies done at Berkley, the originator of BSD UNIX, have considered load balancing on networks of UNIX machines[3, 14, 38]. The results of these studies, although satisfactory in some areas, have been hampered by the high cost of transferring a process, often running into several seconds[94].

The presence of load balancing can also place a strain on the communication media, as witnessed in the case of Tunis[37]. To overcome this, some systems use demand paging techniques (that is, move the memory only when it is accessed) to migrate tasks, and these do lessen the communication overhead. Another problem associated with the migration of tasks is that often state information held by the kernel must be updated, and all communication points have to be moved, causing difficulties with maintaining consistency of system structures and guaranteeing the correct delivery of messages to tasks.

1.3.5 Naming

All the systems support naming at some level; at the lowest level it is simply used for file location. Some files are considered ‘special’, that is they have properties beyond those of the ordinary file used to store data: they are used to represent devices.

Supporting naming, the location of resources, in a distributed system tends to come in one of three forms. Firstly a special service is supplied which performs name lookup for services only. In these, filesystems are disjoint from the name space and are supported by special services, i.e. a request is made for the file server, and after it has been located a request is made for the file. Naming, in these cases tends to be within a flat name space. The second type of naming systems ‘graft’ services into the filesystem in the same way devices are grafted on to a standard filesystem, as in UNIX, or some other global name space. The final approach is not to support a name server, but instead a server registers itself in a group that is known to provide services. When a service is required a message containing the name of the service, is multicast, or broadcast, to all members of the group, and the one supporting the name then replies.

All of these approaches have their own advantages and applicable areas. Using the third method ensures that the service is available whenever the server is present – the absence or failure of a name server, or any other server, does not necessarily prevent the location and use of the desired service. However, as has already been stated it is an expensive method if the media does not support broadcast. Grafting services on to the filesystem provides a clean interface, and allows them to provide a filesystem themselves, if they so wish. Providing a single name server with which all services must register means that it is easy to control the services offered, and who offers them, and to whom. This method is both simple, and provides a single address that need be known to find a given service; unfortunately it also provides a single failure point – if the namer fails then no services can be located until the namer recovers.

1.3.6 Capabilities

The use of capabilities is increasing because of their flexibility and power. They can be implemented so that the processes themselves hold the capabilities, removing the need for the kernel to maintain and keep track of them. With the use of simple, one-way functions they give a high degree of security as these capabilities cannot be forged in reasonable times, if at all. As the kernel does not maintain them, transferring capabilities between processes does not need the intervention of the kernel, further simplifying the transfer mechanism and the kernel.

In Amoeba they have a rights field which specifies those operations that can be performed on the object to which it corresponds, much in the way that the protection fields on a file does in UNIX. This design is such that it is possible to *remove* rights for an object when only the capability is possessed, but not to add new rights unless the capability allows this (i.e. it has the right to add to it). Thus, a process can ensure that another process will not be able to alter an object by modifying the capability to that object before passing it on.

1.3.7 Concluding Remarks

The choice of UNIX as a the model for the operating system does greatly restrict what the system may do, and it places constraints on the hardware in order to implement certain of the constructs. Providing compatibility via a set of library routines is far preferable, though it may be advantageous to support different methods of process manipulation and communication.

The IPC must be fast, and it should provide simple uniform methods of communication, with a simple way of returning results or errors. Although reliable communication is important, there are occasions when the lower cost of unreliable communications is preferable. Capabilities provide a very flexible and powerful method of communication which the processes themselves can control.

Load balancing can give a system a far better response, but it can place a high overhead on the communication system, which may offset the gains of balancing processes. This can be overcome by balancing processes when they are created, which reduces the amount of state information that must be transferred.

Although a broadcast based communication media greatly simplifies communication and naming strategies, it does limit the scalability of the system when more than a few tens of processors or systems are being considered. For example, despite its simplicity, using broadcast messages to locate services causes many problems when the systems are scaled, and to solve these the components are given caches, the searching of which could be compared to the actions of a name server (that is each node is given its own name server).

1.4 Analysis of Systems by Support for Transparency

The ANSA Project[5] defines eight forms of transparency:

- Access Transparency: communications with local and remote objects appear the same
- Location Transparency: the location of objects are hidden.
- Concurrency Transparency: objects may be used, by many other objects, without these other objects being so aware.
- Replication Transparency: Multiple instances of an object may exist without this replication being apparent.
- Failure Transparency: Object failure is hidden from other objects.
- Migration Transparency: objects may move within a system without affecting the system.
- Performance Transparency: objects are maintained in such a way as to optimise performance of the system.
- Scaling Transparency: The system can be resized without its components needing to be altered.

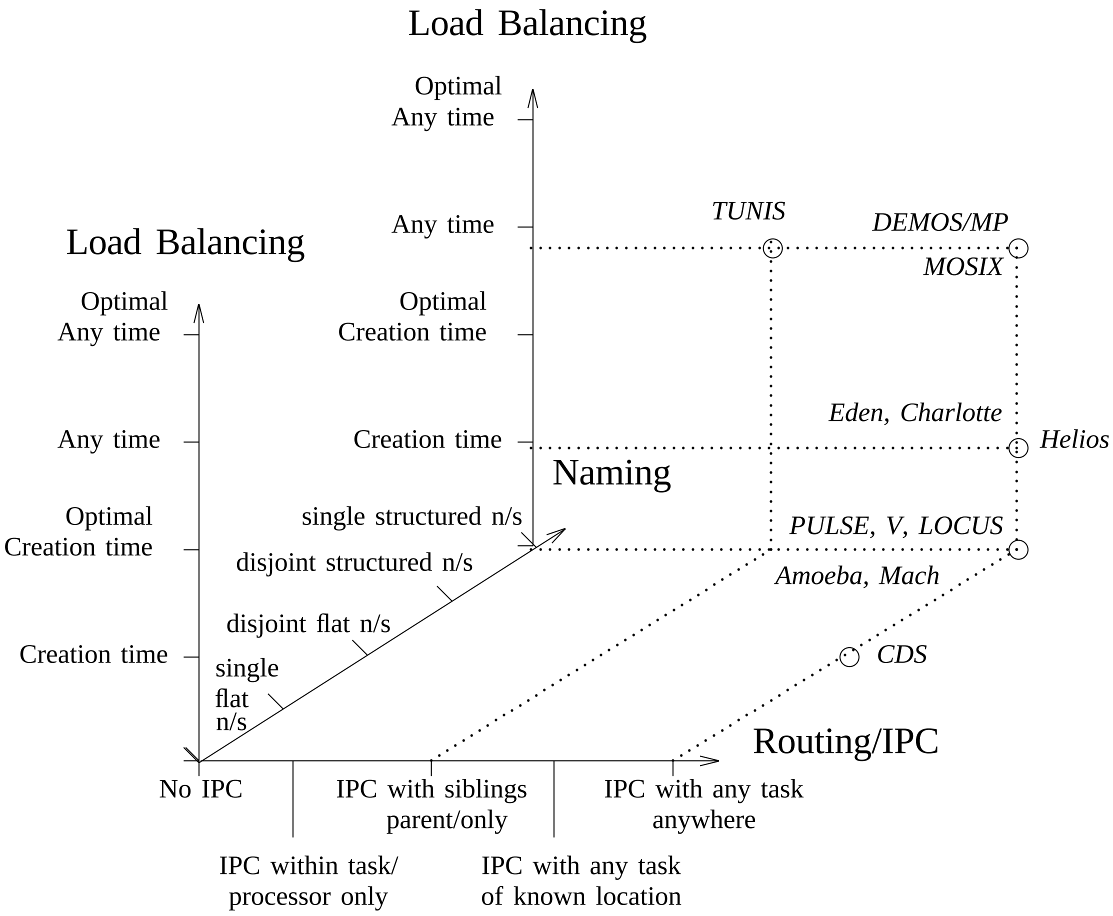


Figure 1.10: Comparison of Systems by Transparency Support

Of the above, replication and failure transparency deal with reliability and availability. These areas are not directly addressed by this thesis, and are not considered further. Concurrency transparency is a property of an object, not the system, as it is the object itself that implements this transparency. This form may, or may not, be desirable, depending on the resources being managed, e.g. file accesses may wish this, but access to the raw disk may not. Migration, performance, and scaling transparency can all be supported, at least partially, with the same mechanism: load balancing. When correctly implemented load balancing allows objects to migrate transparently, increases the performance of the system, and allows the system to be easily scaled. Access transparency is an aspect of routing, or the underlying IPC mechanism: it is this that hides or reveals the remote or local nature of other objects with which the communication is taking place. Location transparency can be provided by naming, since the name should, ideally, give no hint as to the location of an object; the system converts the name into a location.

Thus the support a system gives, ignoring the issues of reliability and availability, can be described by the support of three areas: routing and IPC, naming and load balancing. The provision of these areas is not simply a yes or no, for example consider electronic mail. To mail someone you have to both know their name and their site, thus the provision of naming gives only partial location transparency, since the single site name pretends to be many sites. Thus the ideal namer is one which supports complete location transparency. Routing and IPC are also complex since they may support uni, or bi-directional communication, and may support the movement of IPC channels, and a range of other options as exhibited by the above operating systems. Hence the ideal IPC mechanism is one which gives the maximum flexibility whilst still concealing the physical locations of the communicators. Likewise load balancing: this can be made transparent to the process or not. For example the process may have to request explicitly to be balanced, or it may have to list sites to which it is willing to run at, which limits scalability. Therefore the ideal load balancing policy is one which is totally transparent to the processes which are running with it.

Figure 1.10 describes the above operating systems based on how well they meet the criteria of transparency discussed above. In the figure, stating that there is no load balancing does not mean that it cannot take place, simply that the process must explicitly request this – in which case balancing is not transparent. The load balancing axis represents a range of abilities of load balancing, and not a few fixed points. It should also be noted that this axis is somewhat strange in that ‘any time’ load balancing does not necessarily mean that ‘optimal creation time’ load balancing takes place. Transparency was chosen as the method of analysis and comparing varying distributed operating systems because of the important effects it has on the design and usability of a system. If transparency is not present then both the system and the programs that run under it it must have explicit knowledge of certain factors that may change during the system’s lifetime, i.e. programs may cease correct operation because the numbers ‘hardwired’ into them are no longer valid.

Chapter 2

Philosophy of Wisdom

> *Perfection is reached, not when there is no longer anything to add, but when there is no longer anything to take away.*
>
> *A. Saint-Exupery*

Multiple processors are being used in this system as they provide an easy and affordable way of providing a large amount of processing power. However, the programs must be designed and written in a way that can exploit the parallelism inherent in the multiple processor environment of *Wisdom*. For this reason *Wisdom* must adopt a philosophy and programming model that encourages the use of parallelism and provide support for it at a usable level.

The philosophy of a system is the set of ideas upon which both the system, and the programs written under it, are designed. In UNIX, the philosophy was simplicity, which manifested itself in each 'command' existing as its own program, and having its own task within the system (the same had been suggested for MULTICS[32] but was abandoned for efficiency reasons). In some cases the commands are constructed from parallel combinations of other commands, but more commonly they are single programs, or a sequence of single programs. The power of UNIX, and one now frequently overlooked by programmers, is the ability to combine programs together to solve problems.

The philosophy of *Wisdom* is similar to that of UNIX, simplicity and the combination of programs, but with an added emphasis upon minimalism – the provision of *only* those services that are needed. The philosophy extends into the design of the operating system itself. It also takes the view that programs should abide by the philosophy: they should be made from many simple cooperating parallel components.

In order to encourage simplicity, to support the parallelism needed, and to provide the mechanisms necessary to connect the components together, foundations are required upon which programs can be built. These foundations are described in terms of a model of transparency in which the various components providing the transparency hide, or more accurately make transparent, the various details of the hardware. The ANSA[5] work on transparency has already been introduced, and this is used as the basis of the model, ignoring those aspects which deal with reliability and availability which are not dealt with in this thesis.

2.1 Transparency Model

As with all operating systems, *Wisdom* supports the modules that run under it, however *Wisdom* is not described in terms of its support, but rather in terms of what it hides from, or makes transparent to, these modules: in other words the transparency it provides. The transparency creates a virtual machine, and the components which provide the transparency map the modules executing on this virtual machine on to the physical machine. The most important aspect of the transparency is that it should provide a natural mapping on to the hardware provided, for example a transparency of a single, immensely powerful processor does not give an easy mapping on to the parallel hardware that is being investigated. However, the transparency should still be sufficiently general so that a wide variety of problems may use it, i.e. the model should be flexible in terms of what it is able to support.

The elements of the transparency that *Wisdom* provides are listed below, and have been chosen based on the above criteria (flexibility, natural mapping to hardware in use, simplicity and minimalism), and, as stated above, are also derived from the ANSA work.

- *Parallelism Transparency*: the virtual machine has as many processors as the module requires, and the processors are found and allocated to the components of the module, without the module being aware.
- *Interconnection Transparency*: the virtual machine is one with complete interconnections between all the virtual processors that it supports, i.e. anything anywhere may communicate with anything else anywhere else.
- *Location Transparency*: the virtual machine supports a mechanism which allows any virtual processor to locate any other virtual processor, which is willing to be located.

The parallelism transparency is vital in a system which uses, and exploits, parallel hardware, as without it all modules must explicitly match themselves to the hardware, thereby reducing their portability. To use parallelism effectively, it must be possible to communicate with any other part of the system, hence the interconnection transparency becomes necessary. For a task to communicate with another, there must be some method of specifying the task to be communicated with, and this should be possible without reference to the physical location of the task (as this may not be known) – this is performed by location transparency. These three components are supported by the load balancing, routing and naming modules respectively.

On the basis of these elements of transparency, the conceptual, idealised form of *Wisdom* may be compared with the various systems described in the previous chapter. This comparison is shown in figure 2.1. Whether or not the prototype achieves the transparency implied by its position in the figure will be dealt with in chapter 6.

2.2 Modules, Tasks and Parallelism

The above section has described the transparency *Wisdom* provides, and hence the virtual machine it creates: fully connected with unlimited processors. This section describes how modules are structured to make use of this virtual machine, and the following section describes the programming paradigm which is used in *Wisdom* to combine and access services provided by modules.

Structuring is achieved in *Wisdom* by the use of modules, which are the equivalent of a program in a conventional system. However, modules are comprised of one or more tasks,[†] with each task potentially given its own processor; modules may contain other modules (or be parts of other modules themselves). Thus there is parallelism within a module. It should be noted that these tasks form the

---

[†] In the prototype system modules are limited to a single task to maintain simplicity.

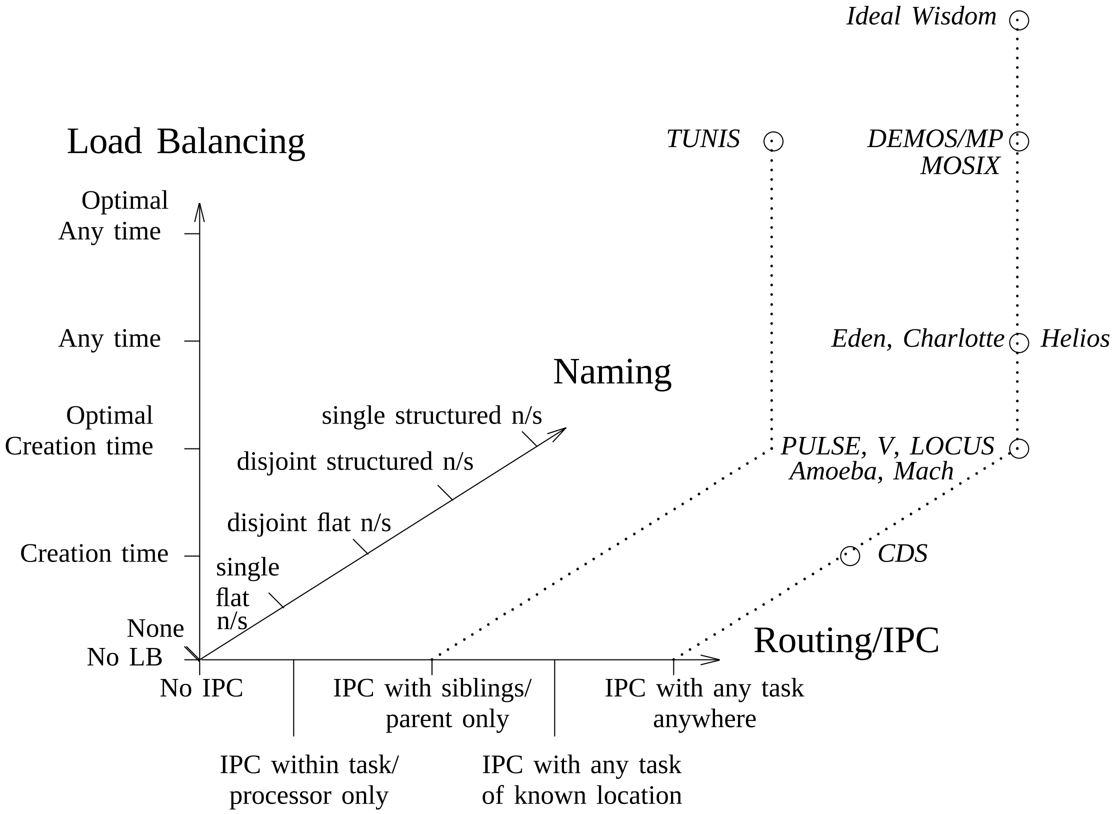


Figure 2.1: *Wisdom* Transparency in the context of other systems

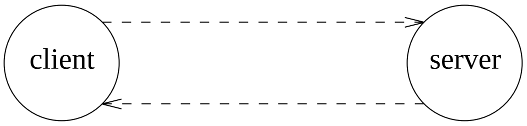
smallest element of true parallelism supported by *Wisdom*, so any parallelism within a task would be that of concurrency, and not true parallelism, and would therefore have the semantics of threads.

A module is considered to be capable of providing some useful service, such as sorting, which might be used in conjunction with other modules. A task, by comparison, provides a specialised service which might mean little in other contexts, such as comparing two items of data. Thus a module allows a convenient grouping structure for many parallel tasks that are not of use in themselves; a very good analogy is that of procedure libraries (with tasks) and utility programs (with modules) in the UNIX system.

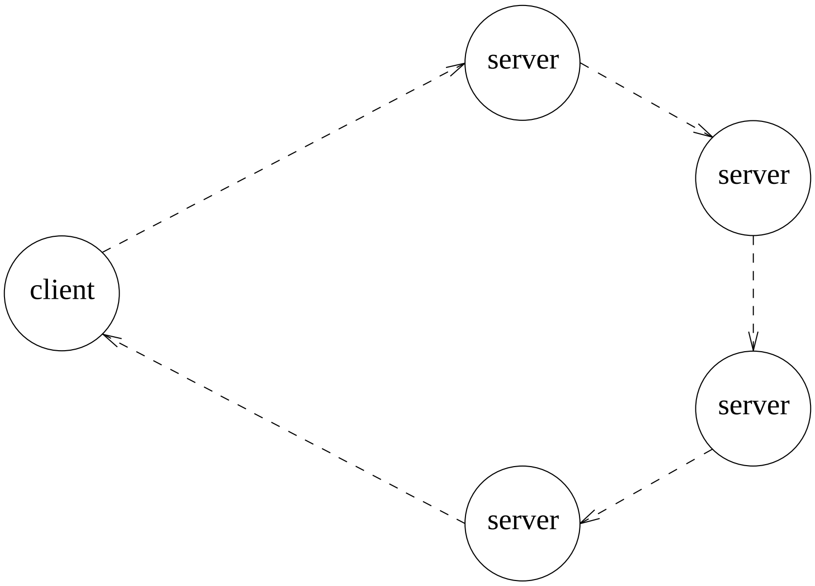
2.3 Models of Computation

All distributed and parallel computing systems are based upon a model for distributed computation, and they are often tailored to give an optimum performance for this model. The most common is the client-server model, in which one module, the client, contacts another module, the server, and requests of it a service, and the client then waits for the server to perform the service and reply. This maps very well on to the workstation-server model of distributed systems, where the workstations are the clients, and where the server needs not normally contact other services.

Figure 2.2(a) shows an example of the conventional client-server model – one client and one server. This is also the version most commonly used, though it is, in fact, a degenerate form of the general client-server model. In this there are potentially many servers, and an example is shown in figure 2.2(b) which has one client and four servers. This is more flexible than the degenerate case, and this form is needed by the *Wisdom* philosophy as services should be built from many small components. This version is also more efficient than using the degenerate case, and having the single server contact other services.



(a) client-server model



(b) Wisdom client-servers model

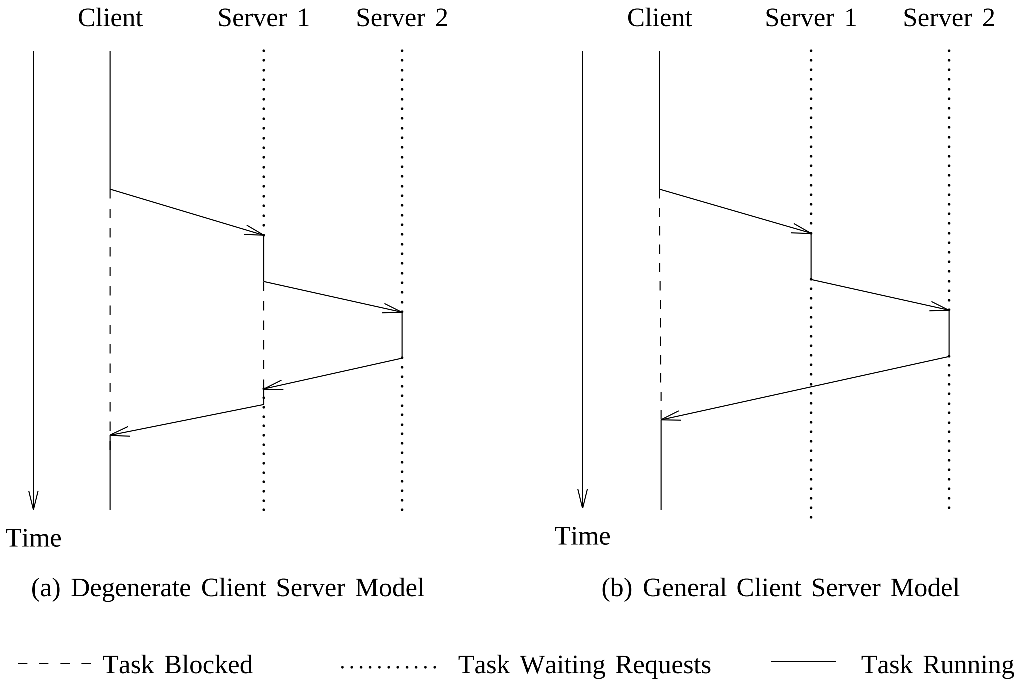
Figure 2.2: Various Client Servers Models

To explain the differences consider a scenario with the degenerate client-server model in which the server must contact a third party to complete the request made by the client, e.g. a file server must request the disk driver to read some data before it can return the results of a read to the client. Figure 2.3 shows the points at which these may block. In the degenerate model the server is blocked until such time as the third party has performed the request and replied, as in figure 2.3(a), whereupon it immediately returns the results to the client. With the general client server model the server is free to continue as soon as it passes a request to the third party, as is shown in figure 2.3(b).

This model makes certain demands upon the underlying communication support since a bi-directional communication link is no longer satisfactory. The reason for this is that the requests made in this model do not necessarily wish to have the results returned to the place from which they came, for example when the fourth server in figure 2.2(b) returns results, they do not go to the third server but back to the client. This could be considered analogous to a procedure call not returning from whence it came, but rather to a place specified by the caller.

2.4 InterModule Communication

At the heart of any parallel or distributed system is the communication between the components that comprise the system; *Wisdom* is no different. The design of the intermodule communication (IMC) facility is based upon the requirements of the client-server model (uni-directional, transferable, many-to-one) and on the structuring methods. In addition, care was taken to ensure that tasks should be able to execute whenever possible (non-blocking). All messages in *Wisdom* are sent over channels, and these have the following characteristics.



(a) Degenerate Client Server Model

(b) General Client Server Model

Figure 2.3: Execution Blocking in the Client Server Models

- *uni-directional:* All the channels support a flow of messages in one direction: from a writer to a reader. A message may, optionally, include in it a channel to which a reply is to be sent.

- *transferable:* Channels can be transferred over channels, as is obvious with the reply channel. However, only the writer end of a channel may be sent. When a writer end is sent, if the sending module owns the channel then it retains the reading end, otherwise it forfeits the right to send messages to the channel (unless it retains a duplicate). When a writer end is received the receiving module gains the ability to send messages to the reader end of the channel.

- *one-to-one or many-to-one:* A channel may have only one reader, that is only one module may receive messages on a channel. It is possible, however, to duplicate the writing end of a channel, and so there may be many modules holding a writing end of a channel, and hence many modules sending messages to a single module.

There are also aspects of a communication facility which are not determined by the requirements of the model. These may be chosen based upon other criteria, which in the case of the prototype *Wisdom* is ease of implementation. These aspects include blocking or non-blocking, reliability, datagrams or virtual circuits – these will be dealt with in chapter 3 which deals with the Router.

Although only a writing end of a channel may be transferred it is trivial to set up bi-directional communication between cooperating modules. When the first module sends out a writing end of one of its channels it then proceeds to listen, on the reading end of the channel it sent out, for a reply. The other module waits to receive the writing end to the first, and, when it receives it, sends down this channel the writing end of one of its own channels. Hence both modules have writing and reading ends to each other.

Restricting the channels so that only the writing end may be sent greatly simplifies the support necessary for intermodule communication (IMC) since it is only necessary to transmit the destination address of the channel. If, once running, a module can move then it is necessary to provide a mechanism to forward messages and update the destination information the writing end of the channel holds. However, in the prototype *Wisdom* the task, once running, never moves hence this mechanism is not needed.

2.5 Ideal Wisdom

The objective of *Wisdom* is to use a loosely coupled, high-connectivity parallel architecture (that is one with no shared memory) and provide a general purpose operating system (GPOS). Past operating systems have shown that a high degree of transparency is required if the physical details of the underlying hardware are to be hidden and this is also true if parallelism is to be used by modules without them having to be reconfigured, or rewritten for differency hardware configurations. This transparency is achieved by the use of model described above, the virtual machine, and is exploited by the use of highly parallel models of programming, such as the client-server model and the highly parallel structuring.

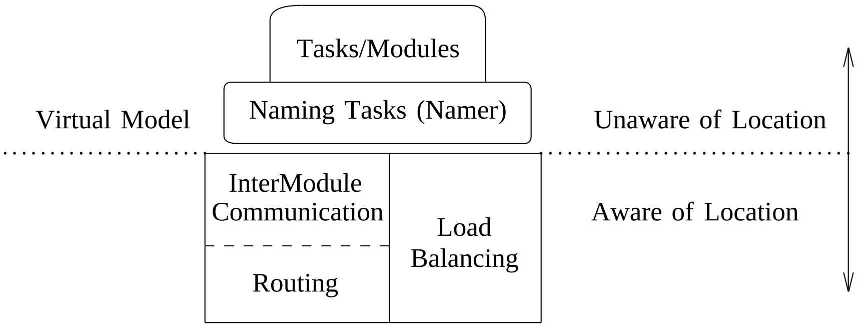


Figure 2.4: The idealised view of *Wisdom*

The virtual machine supported by *Wisdom* was described as comprising of three parts, and each of these parts is supported by a single module. In the idealised view of *Wisdom* these modules are the parts from which *Wisdom* is made – they are the foundations and all else is built above them. The arbitrary parallelism is supported by load balancing, the arbitrary interconnections and the IMC are supported by routing and the arbitrary location is supported by naming. Figure 2.4 shows how these three components are combined: the Router and Load Balancer combine to support the user tasks, one of which is the Namer. The IMC shown in the diagram is not a physical piece of code, but instead the protocols used to communicate, using the methods described above.

Of these three modules only the Namer is seen by user tasks, as it is the only user task. The Router is hidden, and its existence is only known by the ability to send messages from anywhere to anywhere. In a similar way the Load Balancer is hidden: arbitrary tasks are created, but the fact that they may have been placed remotely is transparent to the user tasks. The following sections will describe these modules, and mention is also made of the method and properties of task creation as this is an integral part of the system, and the effects this has on the design of the Load Balancer.

2.5.1 Router

The Router performs the task of accepting a message, which is an arbitrary item of data, and a destination for the message. It then conveys this message to the destination in exactly the same manner regardless of the destination. This module provides the arbitrary interconnection transparency by allowing all the processors to communicate with all other processors regardless of location. Since all tasks must communicate through the Router, regardless of whether or not the destination is on the same physical processor, there is no difference between sending messages to local or remote processes (ideally that is – in practice there will be differences in speeds between these instances).

This module must route messages to their destination as complete interconnection is not assumed. This differs from the conventional systems described in chapter 1 as their communication media allowed messages to be transferred direct from source to destination. A detailed description of the Router is given in chapter 3.

2.5.2 Namer

The naming service, the Namer, is provided by many component namer tasks. Figure 2.5 shows eight naming tasks connected by channels (the lines) which are implementing (a section of) a name space. The term Namer is used to refer to the collection of all the tasks, a single one of which is referred to as a namer task.

The namer tasks receive requests for a name, this is then mapped on to an IMC channel down which the request is then sent. This mapping is called name resolution. The set of names that the Namer can resolve is referred to as the name space. The Namer acts as an interface between the tasks and the Router: it allows tasks to refer to other tasks with abstract names, typically character strings, which it then maps on to IMC channels over which the tasks can then communicate.

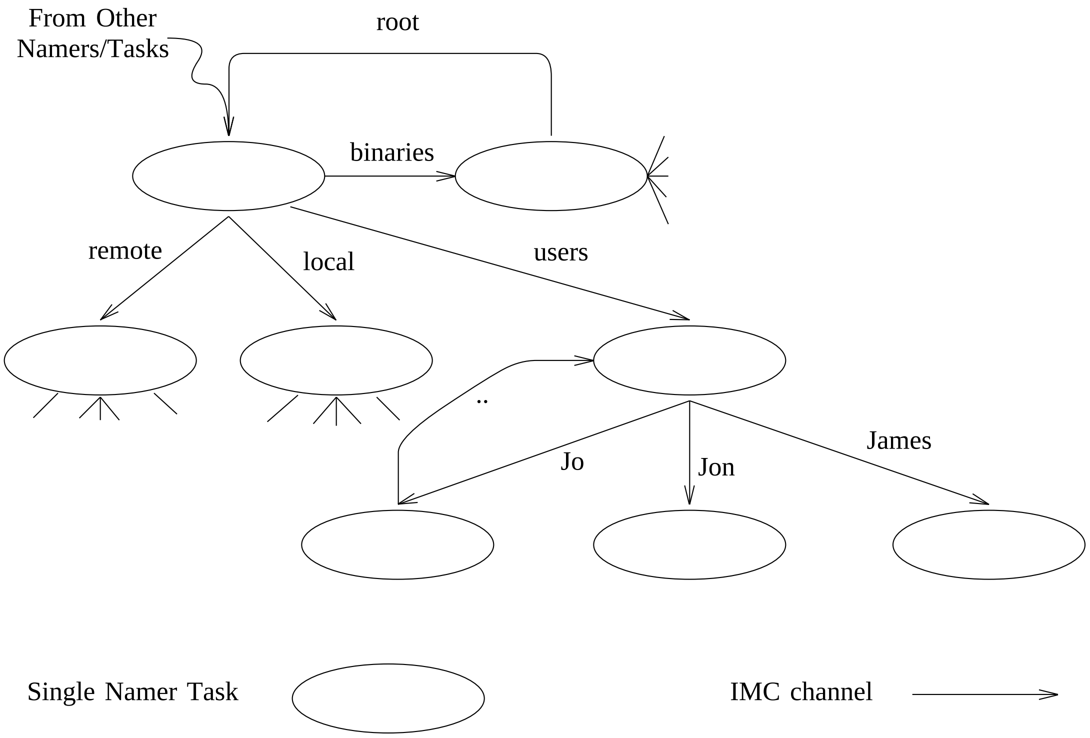


Figure 2.5: Example of Multiple Namers Implementing a Name Space

The structuring of the Namer has been designed to increase the parallelism in the global name space, and hence speed up name resolutions within that name space, or more accurately to allow more to occur at the same time. Every namer task implements a mapping of names on to IMC channels, and this mapping is flat; that is, it is a single directory. Structure in the name space is imposed by the way in which namer tasks are combined. Name resolution is performed by each namer task mapping the first ‘segment’ of the name on to an IMC channel and then passing the remainder of the name to be resolved down this channel. This process is repeated until there is no remaining name to resolve and an IMC channel has been identified which corresponds to the name being resolved. To maintain orthogonality the namer task then sends the degenerate request (i.e. one now without a name but simply a channel) down the channel and the service to which that channel is connected is then responsible for performing the necessary actions.

This approach gives a very high degree of parallelism, as every directory is its own task. Also, by allowing each task to complete the resolution of its name, the task can maintain control of the connections it has (e.g. to prevent a service becoming overloaded). Since the name resolution requests are passed on to user tasks in the same way that they are passed between the tasks of the Namer, user tasks can implement names exactly as if the system Namer was responsible for them.

2.5.3 Load Balancing

The Load Balancer receives requests for a task to be run (ideally at creation time and at points where it is timesliced); it then consults the processor on which it resides, and as many other processors as feasible, for their load. These loads are compared and a processor is selected for the task to execute on. This is performed transparently to the task (or in the case of creation, its parent): a task is simply requested to run. This is akin to a parent in a conventional system not being aware of swapping, the creation of a memory map, and the loading of code from disk.

Automatic, transparent load balancing aids the scalability of a system as this performs a mapping of the tasks on to the available processors. If a given application comprises of more tasks than there are processors, the Load Balancer will be forced to map several tasks on to the same processor. When more processors are added the Load Balancer will have the chance to map fewer tasks on to a single processor, and hence speed up the application by taking advantage of the new hardware. This is totally transparent to the application: it has not been altered in any way, but now should run faster.[†]

Load balancing supports arbitrary parallelism by supporting an arbitrary number of tasks, since parallelism is expressed in tasks. By mapping tasks on to the available hardware, the act of selecting an (ideally) optimal physical processor, it provides transparency to the parallelism of available hardware, i.e it hides the details of how many processors there are and their physical locations, and obtains the best performance it can with the available resources – this would ideally be optimal, however, in practice it will be as close as the chosen heuristics can achieve.

2.5.4 Task Creation

When a task is created it is completely dependent upon its parent, that is it has no way to communicate with the outside world other than through its parent. In other words the task does *not* inherit anything from its parent, the parent must explicitly pass, to the child, those things that it needs. Connections to the outside world that a task needs, for example to the Namer, come initially from its parent as does any special information destined for it, such as might have come from a command line. Typically a parent would be expected to pass connections to one or more Namers, the standard input, output and error connections, and any arguments that the task needs. Since the task must receive connections from its parent, it is possible to support the same functionality as automatic inheritance by the parent explicitly passing connections to its children, without the disadvantages for creation time load balancing (connections and state information to be moved, see section 1.2.4). If the task has been given a connection to the Namer, then it will be able to create additional connections as necessary, if not then it is limited to those that its parent gave it.

This model of task creation results in one in with a clearly defined point at which the task has no associated context, and, therefore, a place where load balancing can occur very cheaply. The need for this was mentioned in sections 1.2.3 and 1.2.4. As was also mentioned, the most common use of `fork()` was to create a new process which set up the connections to the child and then execute the code of the child. This is replaced with requesting the execution of the code for the child and then passing, explicitly to the child, those channels that the child needs. This also has the advantage of preventing additional connections being inherited by tasks when they would not normally be able to create these connections themselves. It also prevents more connections from being passed than are needed.

[†] The conditional tense has been used as there may be a single task which acts as a bottleneck, and so the system may not be speeded up by balancing, unless it ends up on its own processor and was not before.

2.6 Real Wisdom

The above sections describe the conceptual view of *Wisdom* and the basic parts in its design. This section deals with the differences between this model and what has been implemented; this consists of additional modules to enable the system to be 'glued' together, and the combination of the Load Balancer and Router into a kernel, though each remains an independent module. The additional modules are firstly to provide tasking support, and, secondly, the Guardians which act as an interface between the tasks and the remainder of the system. *Wisdom*, as implemented, is shown in figure 2.6.

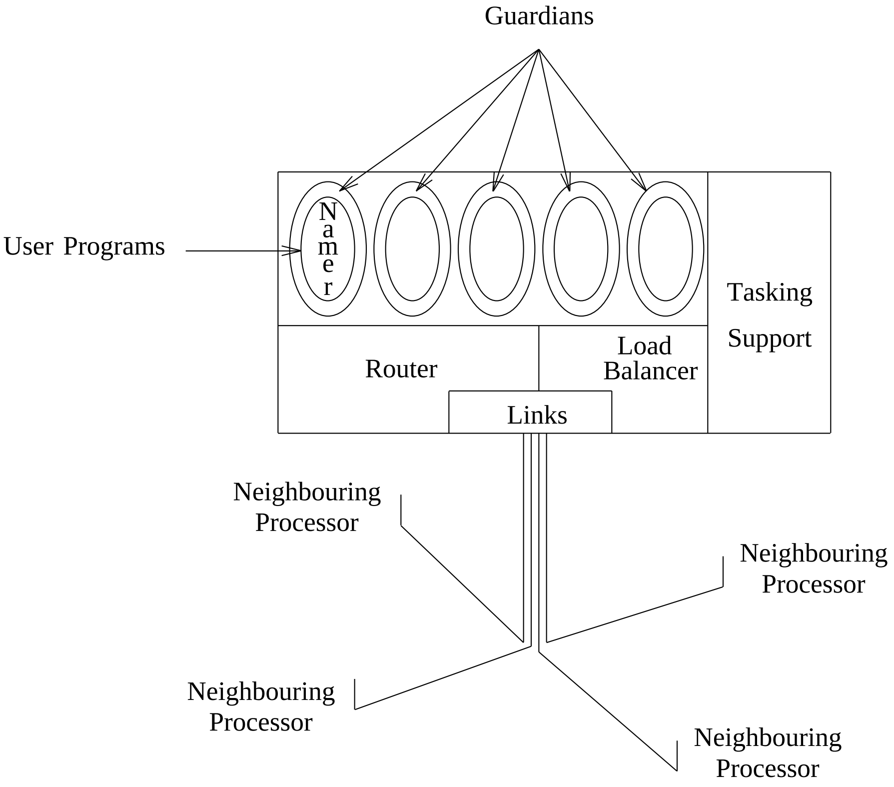


Figure 2.6: *Wisdom* Structuring as currently implemented

The processor on which *Wisdom* is built is the transputer[54] and the language used is that of occam[52]. The reason for this choice was that it held the promise of being an excellent building block for a scalable computer. It also had hardware support for multi-tasking and communications resulting in cheap tasks and fast communications. In practice the hardware multi-tasking proved to be lacking in functionality which was one reason why the tasking module was added. The effects of these choices will be ignored in this thesis as far as is possible, and the discussions will be limited to the ideas instead.

There were also some areas that were modified for efficiency purposes: both the Load Balancer and Router have access to the interprocessor links. The reason is that the Load Balancer has extensive communication only with neighbouring transputers, and the cost of using the Router was not considered acceptable.

2.6.1 Kernel

In a conventional operating system the kernel consists of the support for the filesystem, IMC, process management, resource management and accounting. With distributed systems many services are removed from the kernel, which instead implements a reduced set of services, primarily IMC and process management. In *Wisdom* the idealised view is one without a kernel, however for the system to operate it is necessary for certain services to be present at all nodes: the Router, tasking support and Load Balancer, and so these *may* be considered as the kernel. It is possible, when necessary, to bypass the load balancing software, as will be mentioned in chapter 5.

The kernel also performs the necessary bootstrapping of all the neighbouring processors, and most importantly gives every processor its unique (x,y) coordinate pair. This is done by initialising the row (?,0) completely, and then initialising each of the columns in turn. Hence the kernel automatically configures and uses all the processors in the mesh, assuming they are laid out in an acceptable grid (in this case a Euclidean grid). The kernel should also listen for new processors added to the edge of the network and automatically boot them, but this has not been implemented in the current version.

2.6.2 Tasking Support

It had been originally hoped that the target hardware would provide all the support for tasking that was needed, however it soon became apparent that the support given did not provide the necessary functionality. Consequently it became necessary to provide this additional functionality: primarily task destruction, which in turn required that task creation be handled so that the necessary information could be obtained.

There is no support for large virtual memories, since the current hardware provides no support, and since it is not reasonable (or feasible) to allow such on a large mesh. When a process has a large virtual memory, some of its memory space must be stored on disk when not in use (i.e. swapped out). If the mesh is large (of the order of hundreds of processors) the amount of disk space that is required will become excessive. There would also have to be many disks to prevent the single disk becoming a bottleneck. As a consequence of this the support for memory management is confined to supporting the creation of new tasks, and not to increasing the memory size of tasks. The view is that when more memory is needed, a new task should be created to manipulate this memory.

2.6.3 Guardians

A Guardian encapsulates a task, and can be viewed as a special, system provided, part of a task whose purpose is to interface with the modules that comprise the kernel, and thereby with the system and other tasks and their Guardians. Its primary purpose is that of receiving messages from the task and packaging them for the Router, including the conversion of channels into ‘network addresses’. Likewise it receives messages from the Router, decodes them and sends them to the task, and if necessary sets up a mapping between a received network address and a channel so that all messages on that channel will be sent, via the Router, to the network address. In addition to this, it packages the messages into the fixed size packets used by the Router. In effect part of the guardians actions are to implement IMC.

The guardian also receives requests from the task to create new tasks which it passes to the load balancing module on behalf of the task; it also converts the channel sent with the task into a network address which the Guardian of the new task will map on to a channel for the new task to use. It may also receive a ‘destroy’ message from the tasking module which indicates that the task is to be destroyed and the guardian should perform any ‘cleaning up’ that it can. This message could be generated by the task entering an error state, or by a request for the arbitrary, unavoidable destruction of the task by another task, such as the `kill` program in UNIX.

The guardian is shown in figure 2.7 with its connections to the other modules. The guardian will not be dealt with further in this thesis, since its actions are simple, effectively little more than an intelligent multiplexor and converter for the Router, with a little knowledge of a tasks state to enable certain cleaning up operations to be performed if necessary.

2.6.4 Load Balancer

Some minor additions to the ‘ideal’ design described above have been made to allow tasks to be explicitly placed if necessary. The commands come in two forms: either it is specified that a task is executed locally or the (x,y) location of an *adjacent* processor is given. The processor thus specified is then used for the task, regardless of the load. These additions were initially intended only to allow

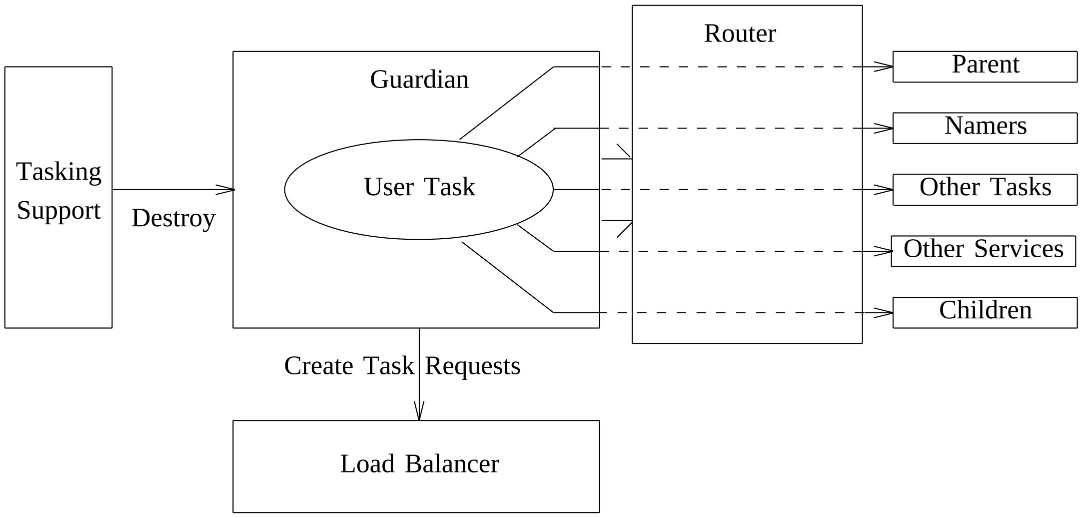


Figure 2.7: The Guardian in *Wisdom*

benchmark modules to be run where load balancing was not needed. Although this breaks the transparency of the model, when the restrictions of its implementation were studied (specifically those that limited load balancing to its immediate neighbours), it was discovered that these facilities could be of beneficial use (see chapter 6).

2.7 Summary

This chapter has introduced and described the philosophy that underlies *Wisdom*: simplicity and minimalism. The application of simplicity and minimalism is such that problems should be solved by combining many components in parallel and that each component should solve a simple part of the problem.

The model of computation in *Wisdom* is one of many modules and tasks cooperating to solve problems or provide services using the general form of the client server method. This dictated the form that the IMC, the communication mechanism, should take: uni-directional, transferable and one-to-one or many-to-one.

The philosophy above has been applied to the design of *Wisdom*. As minimalism is an objective, only the very basic set of functions – the bare foundations – are provided, yet they are general enough to provide support for many different problems and to be applicable to the available hardware. This resulted in a model comprised of three aspects of transparency: location, interconnection and parallelism, which are implemented by naming, routing and load balancing.

Chapter 3

Routing and Topology

> *Too-eager youth leaps to his first confusion: ‘‘know’’. For ‘‘know’’ he imagines you mean ‘‘remembers’’. Humbleness remembers more than most and therefore knows more routes than most, but that is not how he route-finds. Most moles never route-find, they go along so desperate to keep their talon hold on what little they know that they become confused the moment – as you now have, very average Sir, – they loose hold. In short, they learn the way, remember it, and inevitably when they put a paw out of line get lost. In fact, humbleness avers that most become so afraid of becoming lost they go nowhere new at all which, he mildly suggests, is far worse than being lost since it is next to being dead.*

Mayweed explaining the art of route finding to Beechen after having gotten him lost, both moles in *Duncton Found* by William Horwood

---

There are many differences between routing messages in the computer area network (CAN) of *Wisdom* and routing messages in a local area network (LAN) or wide area network (WAN); these include:

- The likelyhood of an error occurring in signal transmission is far lower in the hardware used for a CAN (virtually non-existent) than in a WAN.
- The hardware topology of a WAN is free; in the current version of *Wisdom*, it is fixed in structure, though not size.
- The network nodes, those which route messages, are not (currently) dedicated to message routing in *Wisdom*, unlike those in a WAN, or the dedicated hardware used for a LAN.
- In a WAN, the routing software maintains large tables of routing information – this is not acceptable in *Wisdom*.

However, in common with other routing policies, the *Wisdom* Router should avoid deadlocks. In local area networks, communications are ‘broadcast’ based, which means, in effect, that all nodes are connected to all other nodes via a single medium. Consequently local area networks, and systems based around them differ in that, unlike *Wisdom*, they do not require routing.

In the mesh or torus network of *Wisdom* many nodes will not be connected to one another. The number connected will depend upon the topology chosen (e.g. array, cube, hypercube), and the number of connections an individual node can support. However, in a scalable system, it will not be possible to maintain complete connection using a point to point network (unless an arbitrary number of connections are available on each node). The IMC in *Wisdom* allows *any* two tasks to communicate with each other; but these tasks may be on any node, and could well be on differing nodes, ones which potentially do not have direct connections. An objective of the system is to make all communications appear the same, regardless of whether or not these communications must cross many processors. Thus, it is necessary to provide routing to simulate the connections needed.

The primary objective of the Router is to carry the messages that are sent between tasks and therefore help support the IMC. The current Router transports packets of arbitrary data of a fixed size, so it is assumed that all messages have already been 'packaged' before sending to the Router.[†]

The Router in *Wisdom* provides the following functionality:

- deadlock avoidance
- zero or once delivery
- partially adaptive routing
- partially deterministic routing
- unordered delivery

The proof of deadlock avoidance will be given later in this chapter. The zero or once delivery is a result of the Router reserving the right to preempt messages so that deadlocks are avoided. However, experimentation has shown that this preemption is a very rare occurrence. The routing is partially adaptive and partially deterministic since any message has a set of one or more routes it may take, and the choice between this limited set of routes will depend upon the traffic on individual steps of the route. The routing is not properly adaptive as the message will not be given an overall route which avoids heavy traffic, but will merely avoid it at some stages, if the set of permissible routes includes a less loaded route.

However, as with all modules in *Wisdom*, the Router can be replaced with another providing differing characteristics (or with hardware, such as that described later) or it may be built upon to provide fuller services. Exactly once delivery can be supplied by using well known techniques (end to end acknowledgements and unique message identifiers), and this could be supported transparently by the Guardian. Ordering of messages could be supported in a similar manner.

For a more comprehensive description of routing and related issues, the interested reader is referred to Andrew Tanenbaum's book *Computer Networks*[83]. This chapter limits itself to those issues directly relevant to the topologies and messaging strategies under consideration for *Wisdom*, for example problems associated with routing tables are not relevant here, and hence are ignored.

The remainder of this chapter details the various mechanisms, methods and techniques that may be used to provide a routing service, then describes some of the available routing hardware. The objectives of the router are then given, and these are used to determine the topology of the network, and to select between the routing techniques described above. The Router as implemented in *Wisdom* is then

---

[†] This is carried out by the Guardians: a special task which acts as an interface between the ordinary tasks and the Router. The Guardian receives messages from the tasks and then forwards them to the Router; at the same time it receives messages from the Router and passes them on to the tasks. A fuller description of the Guardian is given in section 2.6.3.

described and a proof of its deadlock avoidance is given, and results from its implementation are given. Mention is made of how the Router is used from occam, and could be used from other languages, and finally a summary of the major points is given.

3.1 Routing Techniques

There are various methods that may be used in routing, or differing services that a router may provide. Some of these methods and services are discussed in the following sections: first is the form of communications they support: virtual circuit or datagram, next is the way they transfer data from one place to another: wormhole or store and forward. There are also many properties that are important when dealing with routing: the first and most vital is deadlock, which is the ultimate form of congestion, but others such as starvation and fairness will not be dealt with. The reasons are that fairness does not have the advantage of a single agreed-upon definition, as is the case with deadlock, which would make generating a proof excessively complex. However, starvation is well defined – it represents the denial of resources to a task – but proofs of starvation avoidance are generally handled by analysis of the implementation code rather than the basic ideas.

3.1.1 Virtual Circuits and Datagrams

A virtual circuit can be considered as a special piece of ‘‘wire’’ created by a router along which data can be transmitted. It is a single, contiguous connection, over which data items are guaranteed to arrive in order and uncorrupted. A virtual circuit has to be created before messages are to be sent over it, and destroyed once finished with (i.e. its resources released). The first of these operations is the only one requiring the destination address, though both these operations can incur processing and time overheads.

A datagram service, by comparison, accepts messages and then attempts to deliver each as an independent entity, without knowledge of any other messages that may be being delivered between the same source and destination. Messages may be delivered in any order, or may not arrive at all. Each message needs its destination address, and the overhead here is that of converting the address to a route.

Issue	Virtual circuit	Datagram
Destination address	Only needed during setup	Needed in every packet
Error handling	Transparent to host (done in the subnet)	Explicitly done by the host
End-to-end flow control	Provided by the subnet	Not provided by the subnet
Packet sequencing	Messages always passed to the host in the order sent	Messages passed to the host in the order they arrive
Initial Setup	Required	Not required

Figure 3.1: Datagram and virtual circuit Comparison, from Tanenbaum[83]

The differences between datagrams and virtual circuits are summarised in figure 3.1. Each method can obviously be simulated using the other: a datagram service is provided on virtual circuits by creating a virtual circuit for each message. A virtual circuit can be made out of a datagram service by either sending all the data of the virtual circuit as one datagram (if that is possible). Alternatively end to end acknowledgements on messages may be used in conjunction with with sequence numbers, timeouts and retransmissions.

3.1.2 Wormhole and Store and Forward

Store and forward routing involves the receipt of the message, which is then stored; at some point after this, it is sent out towards its destination. Since the messages have to be stored, this mechanism forces them to be of a fixed (maximum) size.

In wormhole routing, the message is viewed as a sequence of mini-packets, or flits. The first of these flits, the header flit, contains the destination of the message. The header flit is sent out first, the route determined by inspection of the address it contains. Each node that receives the header flit sends it closer to its destination, the route again chosen by the address it contains. It differs from store and forward since the header, once transferred, is immediately followed by the next flit, which is in turn followed by the next, all taking the same route. Thus the message slowly becomes spread out across the network, with the head of the message leading the way (like a teacher leading a line of school children). To enable all the flits to follow the same route the communication links connecting nodes are dedicated to a single message – that is no other message may use the link until all the flits have been transferred across it. If the header flit is held up at any point (for example waiting for a link to become free) then the entire message is halted (like the line of school children halting because the teacher has). One advantage of this method is that the number of flits in a message is not limited, and so the message may be of an arbitrary length.

Issue	Store and forward	Wormhole
Message Size	Fixed	Arbitrary
Memory Requirements	Potentially Large (several buffers of message size)	Very small (only a few flits need be stored for each channel)
Channels	locked during ‘packet’ transfer	locked during entire message transfer
Deadlock Avoidance	by allocation of buffers	by allocation of channels
Speeds	distance dependent	message size dependent, or distance dependent for small messages

Figure 3.2: Differences between wormhole and store and forward routing

The differences between wormhole and store and forward are summarised in figure 3.2. Wormhole routing can be built on store and forward in the same way that virtual circuit can be built on datagram. Store and forward can be built using wormhole by either transferring the packet as many flits, or using packets of the same size as a flit.

One important point to note is that the relative speeds will depend heavily on the cost of setting up the transference of messages across the link, and the distance a message has to travel; that is, if the cost of setting up is comparable with the time taken to transfer a single flit, then wormhole routing may be far slower than store and forward. If, however, the cost of setting up is insignificant, then wormhole routing may be faster, but it will be the distance the message must travel that determines this. A store and forward message of m bits to be transferred over n links each of b bandwidth (bits/s) will take $\frac{m \times n}{b}$ seconds. The same message travelling over the same network using wormhole routing with f bits per flit will take:

$$\max\left(\frac{m}{fn}, 1\right) \times \frac{m}{bf} \times n$$

Although wormhole routing may have the appearance of a virtual circuit, this is not the case, since the header flit ‘creates’ the channel along which the message is to travel, and the tail flit then ‘destroys’ it. Hence, the channel exists *only* while the message is travelling along it.

3.1.3 Deadlock

Freedom from deadlock is one of the most important properties of any routing scheme. There are four conditions that are required to hold before deadlock may occur:†

- *mutual exclusion* – only one process (or a fixed maxima of processes) can use a resource at once (i.e. the resource is non-sharable).
- *hold and wait* – there must be a process which holds one resource while waiting for another
- *no preemption* – a resource can only be released voluntarily by a process
- *circular wait* – a circular chain of processes must exist such that each process holds resources which are being requested by the next process in the chain

To prevent deadlock, one, or more of these conditions must be shown, and made not, to hold.

The resources used by the Router to transfer a message must be dedicated to a single message, so that we cannot break the *mutual exclusion* requirement. In store and forward and wormhole routing, the *hold and wait* requirement is inherent in their design: they *hold* on to the resource (the storage space for a flit or packet, or a channel) and *wait* until another resource has been allocated; into this they transfer the contents of the first resource, before releasing the first.

Thus, deadlock avoidance generally involves making sure that the third and fourth conditions do not hold. This is generally done by ordering the resources (e.g. buffers in store and forward and the channels in wormhole) to break the *circular wait* requirement, or by preempting the resources used by messages to break the *no preemption* requirement.

The TORUS routing chip[91] avoids deadlock by creating ‘virtual channels’ on top of the actual channels, and labelling them (these virtual channels are then multiplexed on to the actual channels). The virtual channel selected is based on the destination of the message, and it is shown that the dependency graph of the channels is acyclic, so that deadlock cannot occur: i.e. the channels are ordered to break the cyclic wait condition.

Merlin and Schweitzer[56] proposed a method of deadlock avoidance by ordering the buffers, and accepting a message only into a buffer of a higher order. A message would then either be delivered or removed, because there is no buffer of a higher order.

Both of these proofs are, effectively, the same. A graph of resources (buffers in the former, channels in the latter) is generated; the resources are ordered and one is made dependent upon another of a higher (or lower) order. Hence it is shown that there cannot be cycles in the resource graph (cycles represent potential cyclic wait conditions), and that deadlocks cannot occur.

---

† These conditions are taken from Peterson and Silberschatz’s Operating System Concepts[67].

3.2 Routing Hardware

Since software routing is relatively slow, some groups have looked at developing hardware to perform the task of routing. The advantages are not only that it speeds up routing, but that it releases the node processor to be dedicated to the tasks executing on that node. The ease with which this can be done will depend on both the topology of the network, and the format that the messages take.

One such VLSI chip is the previously mentioned TORUS routing chip. It supports deadlock free wormhole routing over a *k*-ary *n*-cube. A *k*-ary *2*-cube is a torus, or mesh with wraparound of *k* by *k* nodes. The authors claim network latencies (that is, message transfer times) which are several orders of magnitude better than those available in previous systems. The TORUS chip is being used in the MDP[33] processor which expects to deal with messages approximately every 20 instructions, and expects a data transfer rate of 360 Mbits/s in the production system, and a latency of the order of only hundreds of nanoseconds.

The Xputer[45] is another processor which provides hardware routing. Messages are sent over switches which provide a communication subnet carrying various items of information. The messages are used to access the memory of the Xputer.

Another VLSI implementation is the Hyperswitch network[30] for the hypercube computer. This is designed to set up communication between nodes in a hypercube by allocating available hardware channels and switching them to connect the appropriate nodes in a hypercube structure. The simulations performed give latencies in the order of tens of microseconds for up to 1000 processors (previous latencies went up to 10ms for similar numbers of processors).

INMOS (as well as MEiKO) have developed the ‘link switch’. This is a crossbar switch implemented in hardware which allows any of its inputs to be connected to any of its outputs. Its inputs and outputs are the INMOS link (a 10 Mbit/s serial link). The link switch is generally programmed by a transputer which configures a network which is subsequently used for program execution. Although INMOS initially claimed no speed degradation when using the link switch some studies have shown that, if more than one link switch is used, then the speeds of the links (typically 10 Mbit/s) can be slowed by as much as one half.

3.3 Wisdom Routing Objectives

As has already been stated, the primary objective of the Router in *Wisdom* is to simulate the arbitrary interconnections needed to support full IMC in a transparent manner. The design of the Router must be such that this objective is met, but it must also be scalable; that is must be possible to add more network nodes without reducing the performance of those already present. These requirements, and the fact that the network node supports both the message routing and the user’s tasks, give the following constraints:

- The Router must not deadlock
- Routing must need as little processing power as possible
- Routing cannot use routing tables

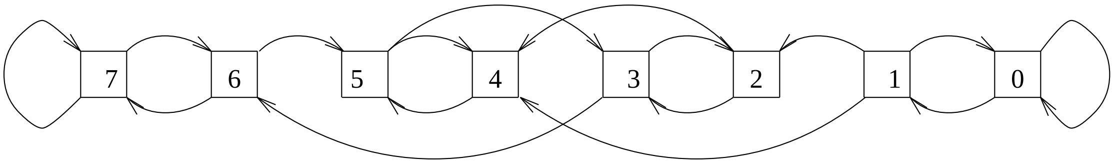
Since routing tables cannot be used, it must, therefore, be possible to determine the route of the message based on the destination of the message: i.e., the following is true:

$$l = R(d)$$

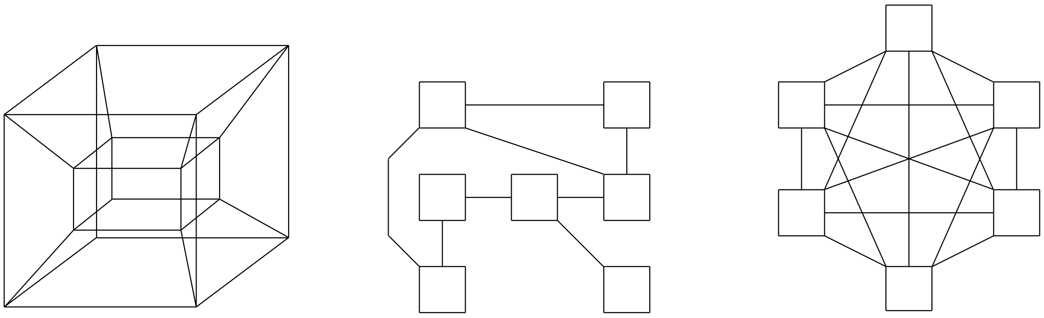
where l is the output link, and d is the destination. Since the processing costs have to be low, $R()$, the routing function, must be computationally cheap.

3.3.1 Choice of Topology

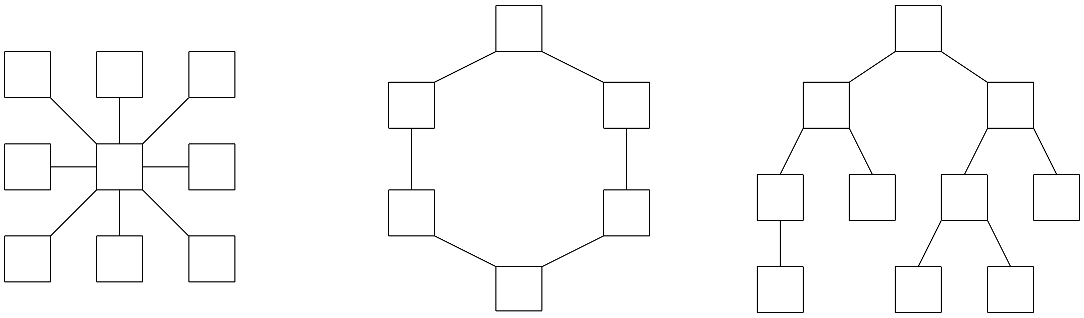
The choice of the topology is important as it determines both the ease with which the system may be scaled, and the simplicity of the routing function. This section considers eight well-known forms of network and weighs their relative merits, and so explains the selection of a mesh over various other possible forms. In all cases, the worst case separation is considered although the system will generally ensure communicating tasks are on closely adjacent nodes, regardless of topology, due to the behaviour of the Load Balancer. Examples of these networks are shown in figure 3.3; the squares represent processors, and the lines interprocessor links. To maintain clarity in (a) the processors were omitted: they occur at the corners of the cubes from which the interconnections emanate.



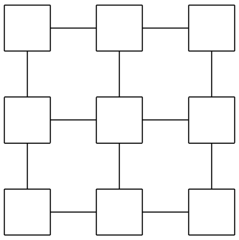
(a) Shuffle-Exchange



(b) Hypercube (c) Irregular (d) Complete



(e) Star (f) Ring (g) Tree



(h) Mesh

Figure 3.3: Examples of common processor networks

Shuffle-exchange

The shuffle-exchange network, an eight node example of which is shown in figure 3.3(a) requires only four links and provides reasonable interconnections. In this, 2^n nodes are sequentially numbered. Each node a is connected to node b, which is the left rotation of the binary representation of a. This is the so-called shuffle link: it shuffles the least significant bit up by one place. a is also connected to node c which differs from a in the least significant bit. This is the so-called exchange link: it exchanges the least significant bit and moves it up one place. Scaling is a major problem, since it requires complete rewiring of the network and that the number of nodes in the network be a power of two. The maximum separation of nodes with this network is n; unfortunately this is also the best separation of nodes in the network, given the way the routing function works.

Routing of messages is divided into stages, each stage moving the message closer to its destination. There are as many stages as there are bits in the address (n represents the number of bits in the address); at each stage s of the route, if the destination bit $n-s$ agrees with the current node's least significant bit, then the message is sent out on the shuffle channel; otherwise it is sent out on the exchange channel.

The shuffle-exchange was not chosen because of its scalability problems and the unacceptable separation of all nodes. Although the routing function is cheap, equally cheap ones exist for other topologies.

Hypercube

In a hypercube, an example of which is shown in figure 3.3(b), 2^n nodes are arranged as a n dimensional cube with two nodes in each dimension. The maximum distance two nodes may be separated by is n. This gives very good interconnections, but means that the topology is not easily scalable as others since adding more processors involves more links and a higher dimensional cube. The interconnections in higher dimensional cubes very rapidly become unacceptably complex, and, even with a four dimensional hypercube, it can become fairly tortuous.

The routing function for a hypercube is relatively trivial, however, since all that is involved is labeling nodes with an n-bit number, where n is the dimension of the cube. Any node with number a is connected over link l to a node with a number b, which is derived from a by changing the value of bit l. Hence, the routing function can select a set of links by comparing the destination with the current node and choosing any link corresponding to a bit in which the two node numbers differ.

The Hypercube was not chosen because of the complexity of scaling the system; however several systems have chosen it owing to the simplicity of the routing function, and its high interconnectivity[43].

Irregular

Irregular networks, an example of which is in figure 3.3(c), are very easy to scale, because, as there are no rules as to the connections, new nodes may be connected anywhere. This means that in a network of n nodes, the worst case separation of nodes is n, though it is obviously possible to improve connectivity in some areas.

Providing a routing function for the message is impossible since it is not permissible to have routing tables. Even if these were allowed, their presence would introduce problems of updating and maintenance, as well as ensuring their correctness. Other methods of routing exist which do not use routing tables, but rely on creating a path to the destination by searching the network (or flooding it – sending a message to every node) which may result in many additional messages.

Thus, despite the attractive scaling factors, irregular networks cannot be used because of the problems of routing – the cost of creating and maintaining routing tables or of flood searches of the network.

Complete

Complete networks, the example being figure 3.3(d), are those in which every node is connected to every other. Scaling the system becomes very hard as every new node must be connected to every other existing node. This also requires an arbitrarily large number of links.

All nodes are separated by a single link, and the routing function is redundant: it simply specifies the link based on the destination node. By use of careful numbering, the address of the node will correspond to the link which is to be used.

Complete networks cannot be used because of the scaling problems: the need for an infinite number of links on all nodes in order to maintain the complete interconnection.

Star

A star network, such as the one depicted in figure 3.3(e), is one in which all nodes are connected to a single, central node. Scaling the system appears trivial since it only requires connecting the node to the central node. However this means that the central node must have an arbitrary number of links.

All nodes are separated by two links in this topology. The routing function for the central node is that same as that for a complete network, and in the other nodes the routing function becomes a constant (the link to the central node).

This network cannot be used because of the problems with scaling the central node. Even if the problem of supplying an infinite number of links at the central node could be overcome, a bottleneck would arise since all communications must go through it. Although not considered here, the central node represents a single failure point.

Ring

A ring network, such as in figure 3.3(f) is a sequence of nodes connected in a ring. Scaling the system is simple and only involves breaking the ring, adding in extra processors and then reconnecting the ring. The connectivity of a ring is not good: the maximum separation of nodes is $\frac{n}{2}$ and the average is $\frac{n}{4}$, and each node is only connected to two others.

The routing function is simple: it selects one of the two links depending on which moves the message closer to the destination node (basically only a comparison). Some rings are unidirectional, which makes the connectivity worse (a message may have to go almost completely round the ring to get to its destination), though the routing function becomes a constant, and deadlock avoidance trivial.

The ring was not chosen because of its bad connectivity and the consequently high latency. There are also problems since most messages will pass through many nodes, placing extra load on all of these nodes. With a ring topology there is a greater chance of the paths used by two communicating pairs overlapping, and this overlap will slow down both pairs.

Tree

A tree network, a binary version of which occurs in figure 3.3(g), is one in which the nodes are connected to form a tree. Scaling this network is simple – new nodes are connected to existing nodes as determined by the arity of the tree. The only problem with adding nodes is ensuring that the tree remains well-balanced. The connectivity is good: the worst case is $2ln_2(n)$, the average is $\ln_2(n)$, with the best being 1.

The routing function is also relatively simple: every node has a path from the root to the node. For example consider a binary tree: if a left child is labelled as a 0, and a right child as a 1, then a binary string is built up which is used as the node identifier. The routing function compares the destination node with the current node using a string comparison. If the destination is shorter, or if, over the length of the current node string, they differ then the message is passed up, otherwise it is passed down to the link corresponding to the next bit after the length of the current node. This could easily be changed to support n-ary trees by replacing the binary digits with n-ary digits. One unfortunate effect of this is that parent nodes become heavily loaded with messages, far more so than leaf nodes, and this can become a bottleneck.

Despite its attractive scaling properties, and the simplicity of the routing function, the tree network had to be discarded because of this potential for overloading.

Mesh, or *n*-dimensional Euclidean space

A mesh network is one in which all the nodes are connected in a grid, or Manhattan, structure, such as that of figure 3.3(h). Scaling this is simple: more nodes are added at the edges, much in the way that a brick wall is built by adding more bricks to its edges. Assuming the network is maintained in a square structure, then the connectivity is reasonably good. For n nodes the maximum separation is $2\sqrt{n}$, averages $\sqrt{n}$ and has a best case of 1.

The routing function is relatively simple: every node is labeled with its coordinates in the Euclidean space (in the case of the mesh, these will be the x and y coordinates). The routing function compares its coordinates with the destination, and returns a set of suitable links that will move the message closer to its destination.

The mesh network was chosen because of the simplicity of the routing function, and the ease with which the system can be scaled, as well as its reasonable separations. This meets the objectives set out earlier (or allows them to be met) – it is simple to build a deadlock free router for it, the routing function is computationally cheap and does not require tables, and routing through the mesh has no bottlenecks.

3.3.2 Virtual Circuit versus Datagrams

In deciding between virtual circuits and datagrams the prime criterion is the need for minimal resources; both memory and computational. For a datagram service, the computation overheads are those of the routing function which must be applied on each hop to determine the route. By comparison a virtual circuit requires this function (or a similar one) to be applied only at its creation and destruction. Memory overheads for a datagram service are only those needed to store a number of datagrams at each node. A virtual circuit service requires memory to maintain the details of all the virtual circuits passing through the node, and memory to store a message for each virtual circuit it supports.

In the model of IMC which *Wisdom*, and consequently the Router, supports, the channels over which data is sent are allowed to be moved, and this is a potentially frequent occurrence.† It is also expected that once a channel starts moving, it may move several times before messages are sent over it. Thus another criterion is that moving a channel must be a ‘cheap’ operation. It is also possible that many channels may exist at any one time, whilst only a few are used. To supply virtual circuits for these would mean allocating and locking the requisite resources when these are to be very rarely used.

---

† For example, the Namer (described later) has a channel with all requests it processes. These requests are expected to be passed a number of times before the name is completely resolved, and hence are transferred many times before the channel with the request is used.

If virtual circuits are to be used, then every channel must be implemented as an individual virtual circuit, or multiplexed on to virtual circuits between tasks. In the first case moving a channel involves destroying the virtual circuit and then creating a new one. In the second case it may involve the destruction of the virtual circuit, depending on the number of channels it is supporting, as maintaining unneeded virtual circuits uses up resources. In both cases supporting a virtual circuit service requires more resources (primarily memory) to be dedicated at each network node. A datagram, in contrast, has no overhead in moving a channel, as only the destination of messages need be changed, and requires less memory resources at each network node.

Thus the datagram was chosen for the following reasons: cheaper channel movement, no additional memory overhead, although more applications of the routing function are necessary these are cheap, and the absence of any setup overheads. The datagram, on to which the virtual circuit may be built, is also a much simpler method to implement. This is also more in keeping with the philosophy of *Wisdom*, than the selection of a more complex, but functionally more powerful, virtual circuit.

3.3.3 Store and Forward versus Wormhole

As in the previous section the prime consideration is still minimal resources, but also of great importance is the achievement of high speed. Of store and forward and wormhole routing, the latter requires less memory resources, and is faster if the cost of setting up for communication is low. The determining factor in the decision is the cost of initiating communication since it requires use of the processor, and with software this is high. With store and forward there is one overhead for each link, but with wormhole routing there are as many overheads per link as there are flits in the message. Hence wormhole routing is unacceptable because it requires far greater processing power than does store and forward.

Once the cost of setting up the communication has been reduced the advantages of Wormhole routing become great indeed. Not only does it reduce the memory requirements, but causes much of the message to be in transfer at any one time, thereby increasing the speed over that of store and forward. Also, it does not limit the size of messages, which can be of use where arbitrary protocols are supported. It is an ideal method for hardware implementation, but problematic for software.

3.4 Wisdom Router

The above sections selected the best methods and mechanisms for the *Wisdom* Router. The following sections describe the Router as implemented. First the router function is presented and described, and then the assumptions that the Router makes are given. The Router does not deadlock, and the proof of its deadlock avoidance is then given. The implementation is then discussed and results from its operation are presented. Finally mention is made of ways in which guaranteed delivery could be provided.

3.4.1 Router Function

As the topology is a mesh, or two dimensional array, it is logical to label each processor with its (x, y) coordinate, using a Euclidean geometry. Hence, each message's destination is the (x, y) of its destination processor. This allows the routing function to be trivial: by comparing the destination (x_{dest}, y_{dest}) with the current (x_{cur}, y_{cur}), an acceptable link, one which moves the message closer to its destination, may be chosen.

The occam code in figure 3.4 implements the router function in the current system. `x` and `y` are the x and y coordinates of the current node (x_{cur}, y_{cur}). The `link.valid` represents the set of valid links on which the message may be sent.

Mathematically, the routing function can be described as

$$R(d, p) = S,\ d = (x_{dest}, y_{dest}),\ \ p = (x_{cur}, y_{cur})\ \ S = \left\{ l : dist(d, n_l) < dist(d, n) \right\}$$

```
PROC routing.function([4]BOOL link.valid, VAL INT dest.x,
                                                  dest.y)

  SEQ
    SEQ i = 0 FOR 4
      link.valid[i] := FALSE
    IF
      x = dest.x
        IF
          y = dest.y
            SKIP        -- message is for this node
          y > dest.y
            link.valid[SOUTH] := TRUE
          TRUE
            link.valid[NORTH] := TRUE
      x < dest.x
        SEQ
          link.valid[EAST] := TRUE
          IF
            y > dest.y
              link.valid[SOUTH] := TRUE
            y < dest.y
              link.valid[NORTH] := TRUE
            TRUE
              SKIP
      x > dest.x
        SEQ
          link.valid[WEST] := TRUE
          IF
            y > dest.y
              link.valid[SOUTH] := TRUE
            y < dest.y
              link.valid[NORTH] := TRUE
            TRUE
              SKIP
:
```

Figure 3.4: Occam Routing Function

where $dist(d,n)$ is the distance between nodes d and n in hops, and n_l is the node connected to n by link l. S is the set of links that a message may take from the current node towards the destination, and hence has a cardinality of two with the mesh network.† The choice of which link, when more than one exists, is made depending upon which is free. This gives the partially adaptive nature of the routing: where an unloaded link exists, which moves the message towards its destination, it will be chosen in preference to a loaded link.

3.4.2 Router Assumptions

Based on the above routing function, the Router enforces the following assumptions on messages and their routing to break the circular wait requirement for deadlock to occur on buffer allocation. The implementation of the Router is such that these assumptions are enforced.

α At each move the distance remaining for a message to travel is reduced, which means that, at every step, the message must approach its destination. This is obviously the case with the above router function.

---

† In fact the cardinality of the set S is equivalent to the dimensionality of the mesh.

β When a message arrives at a destination, it is consumed. This assumption can be met by placing a timeout on all messages arriving at a node and destroying messages after the timeout expires. In practice the messages are delivered to the Guardians, which are permanently ready to receive messages, hence the Router does not destroy messages: the Guardian does.

γ A message for a non-existent node will travel towards its destination until further moves would violate α, at which point the message is discarded.

δ When all the buffers in a node become full, the node will not accept further messages unless the message has a shorter distance to travel than one of those already in its buffers. At this point, the node will preempt (i.e. destroy) the message with the greatest distance to travel to its destination.

3.4.3 Deadlock Avoidance Proof

For any node n, let $longest(n)$ and $shortest(n)$ represent the greatest and least distances that messages at that node have to travel, and let $messages(n)$ be the set of messages on node n. Let $dist(m)$ define the distance that a message m has left to travel. Hence,

$$\forall\ m \in messages(n_i)\ shortest(n_i) \leq dist(m) \leq longest(n_i) \tag{3.1}$$

For deadlock to occur some resource, in this case a buffer, must be held by some message, which cannot proceed because of a chain of dependencies. Let $needs(m)$ define the set of nodes on which a buffer required for message m to proceed is currently unavailable.

$$\begin{aligned}&\exists\ m \in messages(n_i)\ n_{i+1} \in needs(m)\\ &=> longest(n_{i+1}) < dist(m)\end{aligned} \tag{3.2}$$

Otherwise m would be able to preempt a message and use its buffer (in which case $n_{i+1} \notin needs(m)$). By application of 3.1, 3.2 may be rewritten as:

$$\begin{aligned}&\exists\ m \in messages(n_i)\ n_{i+1} \in needs(m)\\ &=> longest(n_{i+1}) < longest(n_i)\end{aligned} \tag{3.3}$$

Now recursively define sets $N_0, N_1, \cdots$ such that N_i is the set of nodes on which buffers are required, but not available, for messages on nodes N_{i-1}; i.e.

$$N_0 = \{n_0\}$$

$$N_i = \{\ n_i\colon \exists\ n_{i-1} \in N_{i-1}\ m \in messages(n_{i-1}),\ s.t.\ n_i \in needs(m)\}$$

Thus,

$$\forall\ n_i \in N_i\ \exists\ n_{i-1} \in N_{\ i-1},\ m \in messages(n_{i-1}), s.t.\ n_i \in needs(m)$$

By application of 3.3

$$\forall\ n_i \in N_i\ \exists\ n_{i-1} \in N_{\ i-1},\ s.t.\ longest(n_i) < longest(n_{i-1})$$

Hence, by induction,

$$\forall\ n_i \in N_i\quad longest(n_i) < longest(n_0) \tag{3.4}$$

N_i represents the set of nodes on which messages on processors in the set N_{i-1} are dependent, hence cyclic deadlock may be defined by $\exists\ i \neq 0\ s.t.\ n_0 \in N_i$.

Assume deadlock can occur. Then, by definition, $\exists\ i \neq 0\ s.t.\ n_0 \in N_i$. Combining with 3.4 generates:

$$longest(n_0) < longest(n_0)$$

As this is an obvious contradiction, deadlock cannot occur.

3.4.4 Implementation Discussion

The routing policy described above has been implemented in occam. This code has been run on a single transputer, simulating a network of transputers of the appropriate size. During these tests, counts were kept of the messages which were preempted by the network, refused entry into the network and delivered. The following table shows the percentage of messages sent and those preempted – those refused entry repeatedly retry until they are accepted.

Array Size x	y	Lost	Sent	Percentage Lost (preempted)
1	9	1028	9000	11.42
1	16	3969	16000	24.8
2	8	871	16000	5.44
3	3	0	9000	–
4	4	0	16000	–
5	5	0	25000	–
6	6	0	36000	–
7	7	7	49000	0.01
8	8	19	64000	0.03
9	9	449	81000	0.5
10	10	3325	100000	3.3
11	11	4130	121000	3.41
12	12	6418	144000	4.45
13	13	6975	169000	4.12
14	14	7937	196000	4.05
15	15	8949	112500	7.95
16	16	12406	128000	9.69
17	17	15745	144500	10.89
18	18	22554	162000	13.92
19	19	31121	180500	17.24
20	20	34754	200000	17.37
25	25	179512	625000	28.72

Figure 3.5: Message Loss in Routing on varying sized Networks

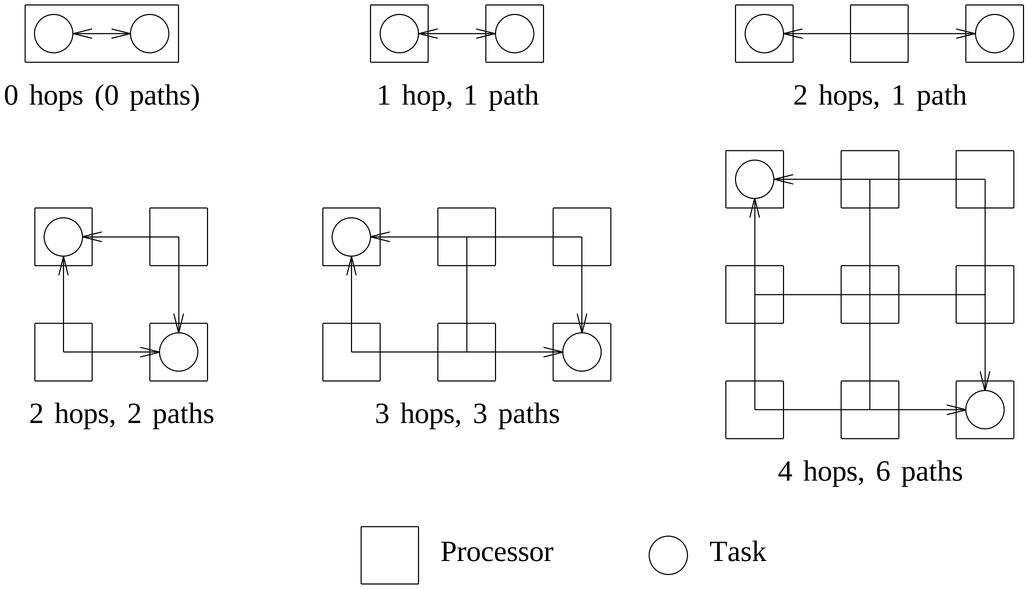


Figure 3.6: Task locations in figure 3.7

Results Discussion

There are several points worth noting about these results, which will necessarily affect the way in which they should be interpreted. Firstly, the messages generated were for random addresses, so on average they would have to traverse about half the mesh of processes. In a real system, this would be unusual since most communicating processes would be kept fairly close together, in which case the message loss rate is far lower. This means that scaling the system up would not produce the massive increase in message loss shown above.

Secondly, the system is one in which messages are generated continuously. In a real system, the messages would be generated at a lower rate, giving the network a better chance to deliver them, though it might well consume them at a slower rate.

There is also a marked preference for a square array structure; possibly due to the way messages are generated. Since a message is randomly created for anywhere in the network, the distribution is best suited to a square array, but in a real system it may transpire that a rectangular array is better.

The code has also been run on a network of nine (three by three) transputers, in which a very small number of messages were lost. This was due solely to the presence, in the network, of a transputer, running at a lower clock speed, which acted as a bottleneck forcing the loss of messages (approximately 0.01%). In the prototype version this processor is avoided and the only occasion of message loss has been where a task has been incorrectly designed and has not been willing to accept messages delivered to it.

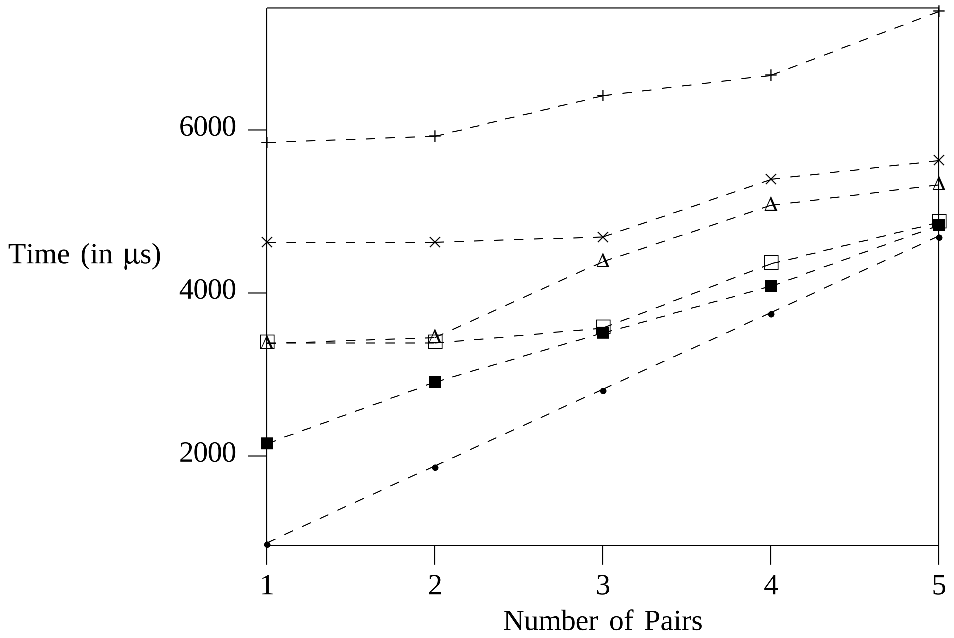
Speed of Communication

Only a small amount of work has been spent on trying to achieve high speed routing. Consequently, it should be possible to improve upon the results that are detailed below. The speeds obtainable have been limited so far by both the choice of processor and that of the language. The actual speed of communication is not of great significance, but rather the relative speeds and the effects of scaling that matter, since it is possible to improve the speeds. The following results are displayed in pairs of graphs, the first gives the average time taken to deliver a message, and the second represents the total number of messages delivered per second by all the processors in the system. All the tests show the effects of increasing the number of processes that are communicating.

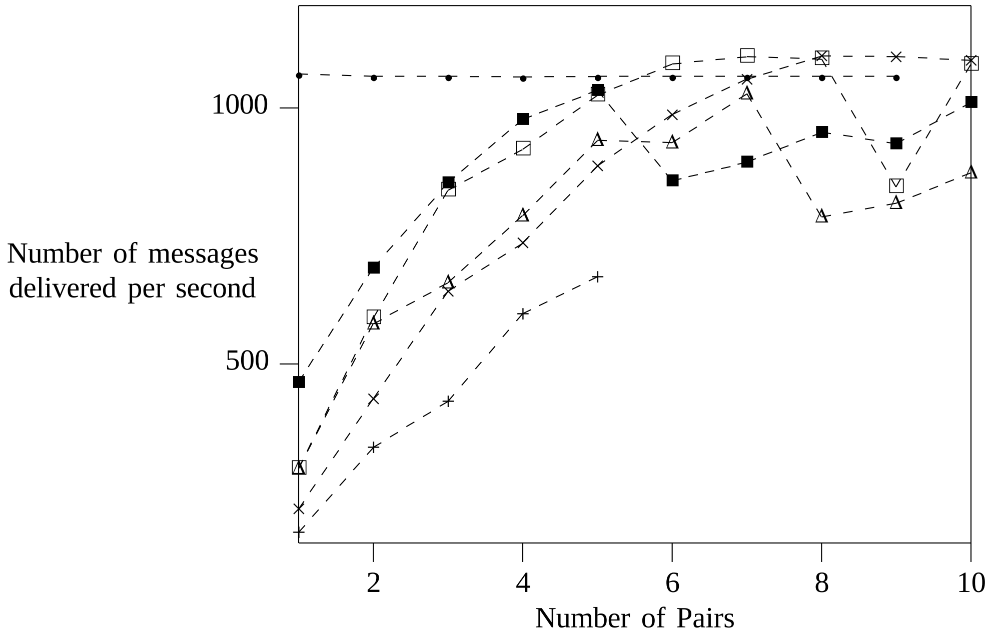
The first results, in figure 3.7, show the time taken to deliver messages for differing distances. The locations of the tasks and the paths that the messages between them may take are shown in figure 3.6. The second graph shows this, but up to a maximum number of ten pairs. The dips in the graph correspond to where the Router attempts to send messages over the same connection in two different directions. Since there are only two signal paths, one in each direction, each path carries both data and acknowledgement: when they become overlapped the speed slows down. The partially adaptive capability of this Router is shown by the difference in performance for the messages travelling two hops when an additional path becomes available (those labeled □ and ▵ in figure 3.7). The time taken for a message to be delivered can be calculated approximately from the following equation (which was derived empirically from the results):

$$t = 0.92ms + (h * 1.2)ms$$

where h is the number of hops that the message has to travel. There is, in the 1.2ms per hop, an amount of time during which the processor is idle awaiting the message. This idle time is used if there are many messages waiting for delivery, so the delivery time does not scale linearly with the number of messages to be transferred. The graphs show the maximum number of messages that may be transferred between two nodes is about 1100, and that the system will utilise the dead time between messages until this maxima is achieved. Where the dips occurred, the graphs show that the system slowly returns to this maxima, but making use of the additional dead time created by the slower link communications. The dips shown in the figure 3.7(b) represent facets of the hardware communication mechanism. The hardware transmits acknowledgements on the same wire as the messages in the opposite directions. The dips occur when a conflict arises between messages and acknowledgements on the same wire.



(a) Time to deliver messages (in µs)



(b) Number of messages delivered per second

0 hops (0 paths)	•	2 hops, 2 paths	□	1 hops, 1 path	■
3 hops, 3 paths	×	2 hops, 1 path	△	4 hops, 6 paths	+

Figure 3.7: Effects of Distance on Router performance

The graphs shown in figure 3.10, and their task layouts depicted in figure 3.8, repeat three of the above tests to exhibit the non-interference characteristic of the Router: that is that two communicating tasks do not slow down other tasks unless the routes of their messages overlap. In the graphs the dashed lines show the results obtained in the above tests.

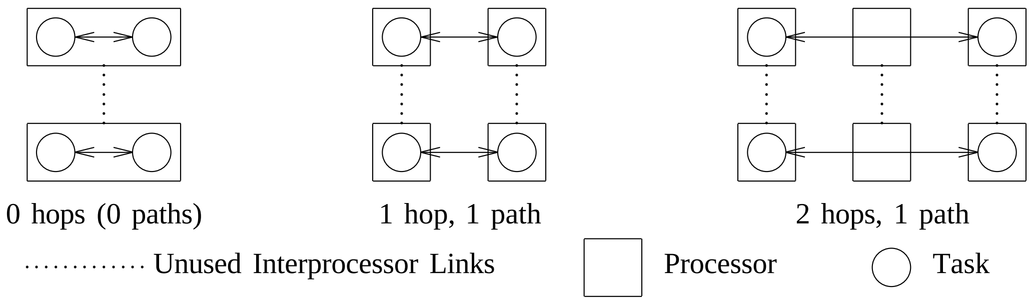


Figure 3.8: Task locations in figure 3.10

The next set of tests examine the behaviour of overlapping communication paths, and the results are shown in figure 3.11 with the task layouts and potential message routes shown in figure 3.9. These graphs again show that the maximum number of messages a single processor is capable of processing is about 1100, but where a path exists between the communicating tasks that can avoid this single node, the Router dynamically avoids it, and the performance is little worse than had there been no overlap.

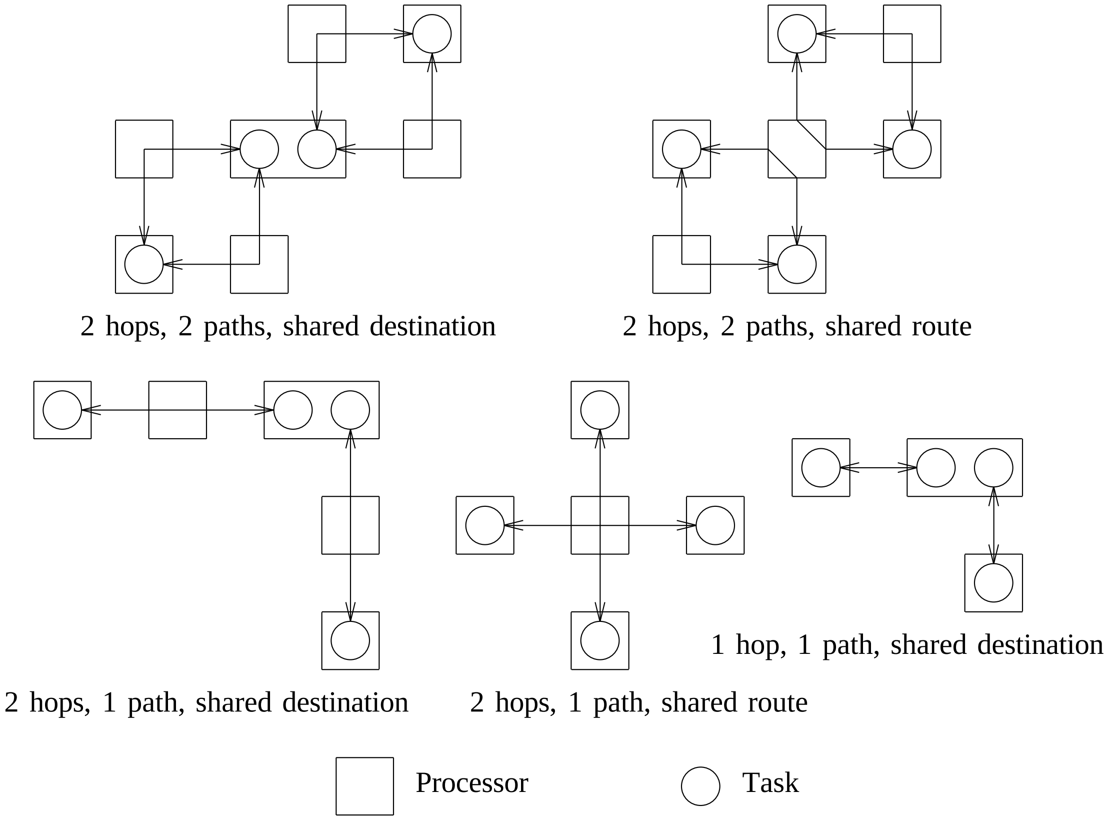
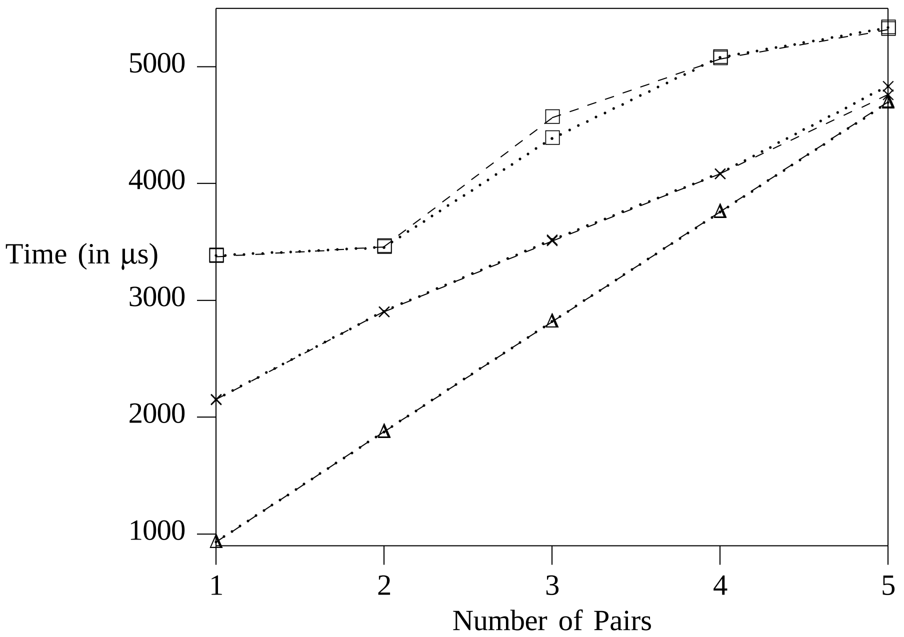


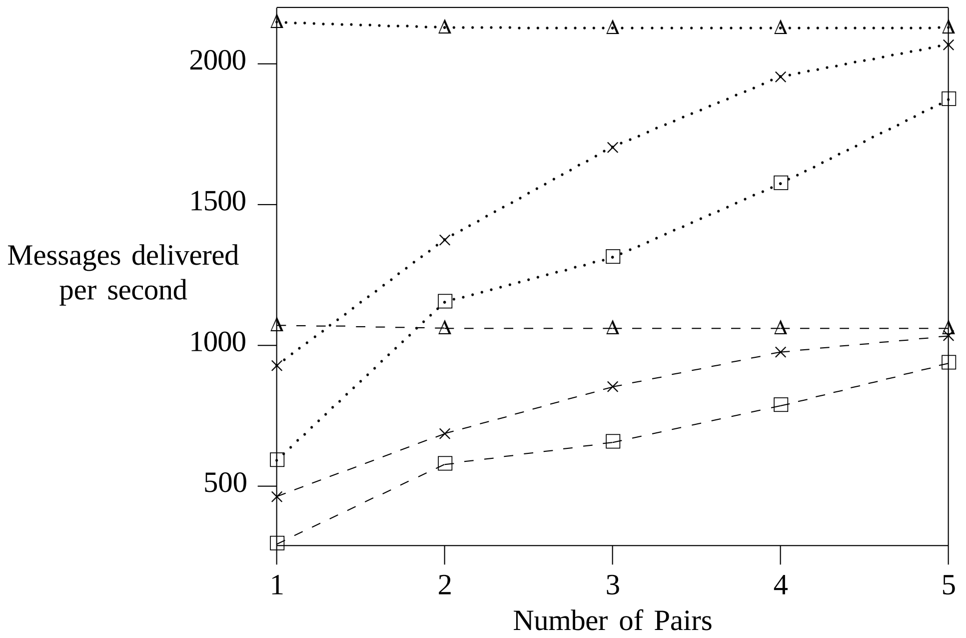
Figure 3.9: Task locations and message paths in figure 3.11

Conclusions

The system, as designed and implemented, performs very favourably when compared with many other systems, such as Charlotte, SUNOS (which takes about 10ms for a local message transfer, and about 12ms for remote transfer of a single word), and is not significantly worse than Amoeba which has been tailored for high performance (which takes about 0.8ms for a local transfer and 1.4ms for a remote transfer, also of 1 word). However this is still significantly slower than would be desired for a system which in its very design (many parallel parts communicating) relies on fast communications, and encourages the use of communication. Further improvements can be made by altering the implementation, though these, at



(a) Delivery Times



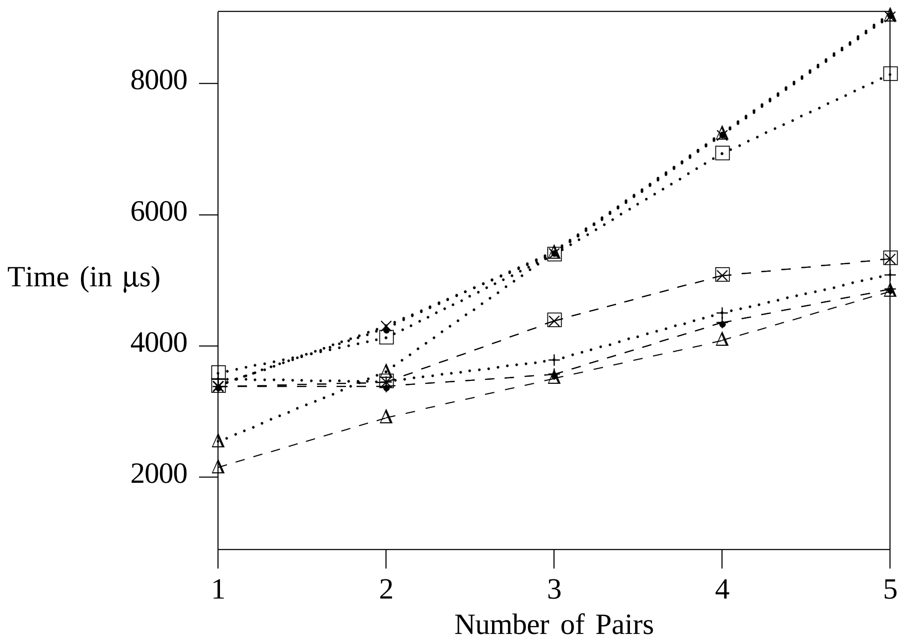
(b) Messages delivered per second

0 hops, (0 paths) △ 1 hop, 1 path × 2 hops, 1 path □

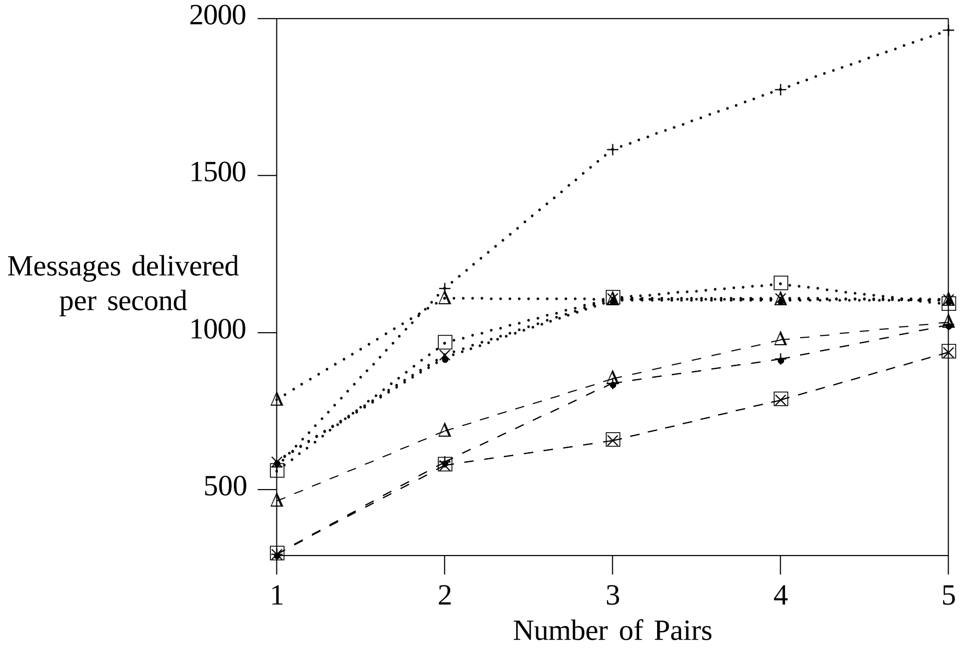
………………… non-overlapping pairs – – – – – – – – – – single pair

Figure 3.10: Effects of non-overlapping communications on speeds

the very best, are limited to an improvement of an order of magnitude. Hence, what would seem to be implied is that some form of high speed hardware routing is needed



(a) Time to deliver single message



(b) Messages delivered per second

1 hop, 1 path, shared destination △ 2 hops 2 paths, shared route +
2 hops, 1 path, shared destination × 2 hops, 2 paths, shared destination •
2 hops, 1 path, shared route □

………………… overlapping pairs – – – – – – – – – – single pair

Figure 3.11: Effects of overlapping communications on speeds

The method of routing does exhibit a vitally important characteristic, which is clearly illustrated in the results. Two communicating tasks do not necessarily affect the progress of two unconnected tasks which are also in communication. This attribute is essential to the scalability of the system. Without it there are limits to the size to which the network can be taken before the communications become saturated.

The node that performs the routing in the current version of *Wisdom* is also the processor which executes the workload placed on the system. This means that when a node is handling its maximum of 1100 messages, it is effectively incapable of executing any additional work, thus the greater the number of messages sent, the less processing time left to perform calculations. It is for this reason that message delivery should be very fast, and ideally it should be handled by hardware support. This represents a potential problem in that some nodes may perform little other than message routing – the solution to this is dedicated routing hardware (which could be built out of a second processor).

3.4.5 Guaranteed Delivery

Providing guaranteed delivery, or exactly once semantics, upon an unreliable communication subnet is well understood. Many methods exist, and it is relatively easy to build such a system for message routing as it is an inherently atomic operation: either the message is delivered or it is not.

The simplest method is to give sequence numbers to all messages on a given channel, whereby any message that arrives can be detected as having already arrived and be thrown away, or as a new message which is then delivered. If, upon message arrival, an acknowledgement is returned, then the sender will know that the message has been received correctly. If, after a timeout interval, the sender has not received an acknowledgement, it assumes the message has been lost and retransmits it, and then waits for the acknowledgement. The receiver then sends an acknowledgement on all messages, though not passing on messages it has already received. It is not possible for messages to be created or altered by the subnet (the Router), which means that the above method will suffice.

It is clearly possible that, if the two tasks communicating messages are far apart, many messages will be lost, given the mechanism used for message preemption in the Router. The way to overcome this is to reduce the likelihood of messages being destroyed by increasing their priority. This is relatively simple to do in the policy described above. Every message has a hop count which decides whether or not it will be received, and whether or not it will be preempted. Modifying this number modifies the priority of the message: the lower it is the more likely it is to get through. Thus, by giving an adjustment to this number, the Router can be ‘encouraged’ to deliver certain messages: for example, the retries of lost messages. This does not break the deadlock avoidance proof as this relies on the comparison of a count which is less on the destination node than on the sender node, when message buffers become full.

There are several points to be borne in mind when providing guaranteed delivery. Firstly that the cost of doing this is high as it, at least, doubles the time taken to deliver a message (since the acknowledgement must be returned). Secondly, the simulations have shown that, in practice, message loss is very low, to such an extent that the cost of this may not be worthwhile in many systems. Finally, it would not be necessary if the underlying Router were replaced by hardware which guarantees delivery (which hardware generally does).

3.5 Routing, Occam Channels and Other Languages

This chapter has largely been concerned with how messages are moved from one processor to another in the network used by *Wisdom*. This section briefly describes how this routing is used to provide communication between programs.

The routing function so far described has had its destination tuple as (x, y). In fact, the system uses the 4-tuple $(x, y, pid, chid)$, where *pid* specifies a task and *chid* specifies a channel in that task. The Router delivers the message to the task specified, and leaves it up to the task to deliver this to the channel specified. As has already been mentioned, this is handled transparently to the task by the Guardian. These two additional pieces of routing information are necessary, since, in the system it is assumed possible that more than one task may be supported on a given processor.

All these pieces of information are hidden from the task: it only sees the message transfer, via the `!` and `?` operators in occam. The differences are that the messages are non-blocking (asynchronous), and that there is a small chance of message loss – both of which may be altered either by the Guardian on behalf of the task, or by replacing the Router.

When channels are transferred between tasks, the Guardians of the tasks perform the necessary conversion of the channels to the 4-tuple mentioned above and back again without the tasks' knowledge. The Guardians also arrange that all messages on those channels will be correctly delivered by using the transferred 4-tuple. When the channel to be sent out is a channel that the task had previously received the Guardian sends out, not a new 4-tuple, but the 4-tuple it initially received. Hence, the messages will always go to the owner (the reading end) of the channel, even though the writing end may be moved between many tasks.

The channel can be viewed as a port or mailbox, i.e. a place where messages will arrive for the task. With this in mind it is clear that any language will be easily supportable in the framework designed for *Wisdom* and occam. Under the POSIX standard, for example, data is often transferred with the `read` and `write` system calls. These map directly on to the model used:

POSIX C standard	Wisdom Occam
`read(from,nbytes,ptr)`	`from ? nbytes :: ptr`
`write(to,nbytes,ptr)`	`to ! nbytes :: ptr`

This, effectively, is how they are supported by Helios[46] – a commercial operating system for the transputer. In the same way that the POSIX standard is applicable to many languages (for example FORTRAN and Pascal), this model may be applied equally as well.

3.6 Summary

In any parallel scalable system, routing must be present since it will not be possible to provide full interconnectivity and maintain scalability. This routing may be in the form of a software module, or hardware support, but it must be be fast, efficient and deadlock free. These requirement on the Router, together with the need to be easily scalable place restrictions on the underlying network, and in this system a mesh has been shown to be the best choice.

The Router can provide its service in several different ways. The data can be transferred over virtual circuits: reliable data independent paths which must be set up before, and destroyed after, use, or in datagrams, which are message packets about whose arrival, and order of arrival, nothing can be said. The cheaper cost of datagrams, the expected dynamic nature of the communication that the Router is to support, and the intent to maintain simplicity make the datagram a more attractive choice. The data, when being transferred, can be either sent as store and forward: the entire message is copied from one node to another node before being forwarded on its way, or as wormhole: small parts of the data are sent out one at a time and the message becomes spread out between the nodes. Each communication has some overhead associated with its initiation, and wormhole requires a large number of these in comparison with store and forward, and consequently is faster only if the cost of these initiations is low. In software the cost of communication initiation is high, and so the store and forward mechanism was chosen. However the speed advantages of wormhole routing are such that this would be the best choice if hardware routing was being built, and the modular design would allow an easy switch to using such hardware if it became available.

The system that has been built performs very satisfactorily: it scales effectively linearly once the maximum throughput has been reached where communications overlap, and non-overlapping communications do not affect one another. Despite this the speeds are slow, for in a highly parallel system the communication *must* be fast to allow tasks to be efficiently broken into parallel components, and thereby allow scaling. As the speeds obtained are comparable with the fastest software systems currently in production it implies that the routing should be provided by a hardware supported communication subnet, which is built along the principles outlined above. Such systems are coming into existence: the TORUS chip described earlier in the chapter and DECs autonet system[81]. Speed is not the only reason for supplying hardware routing support: when messages are being delivered they are using processor time that should be spent on the user tasks, hence slowing the system down.

Chapter 4

Naming

The naming of cats is a difficult matter,
it isn't just one of your holiday games:
You may think at first I'm as mad as a hatter
When I tell you a cat must have three different names,

from The Naming of Cats, T.S. Eliot, Old Possum's Book of Practical Cats

---

The terms names and naming are used, and applied to, a great variety of areas in both computing in general and operating systems in particular. Many researchers differ on what can be called a name, the purposes it serves, and hence the actions that the service that implements, maintains and administers names (a namer) should perform.

Shoch[82]:

> The *name* of a resource indicates **what** we seek, an *address* indicates **where** it is, and a *route* tells us **how to get there**.

Watson[90] (who uses the term identifiers in place of names):

> Identifiers are used for a wide variety of purposes, referencing, locating, or scheduling, allocating, error controlling, synchronising, and sharing of objects or resources (information units, communication channels, transactions, processes, files, etc.).

Needham[64]:

> Names have several purposes in computer systems. One is to facilitate sharing. If various computations require to act upon the same object, they are enabled to do so by each containing a name for the object. · · · Another use of names is exemplified by unique identifiers · · · . It is arranged that names of this form are never re-used, so that · · · [it] either refers to nothing or, if it refers to anything, to the same thing at all times. · · · another distinction relevant to names. This is between human-sensible names such as RMN/ANIMALS/PIG and names intended for internal use only such as #257A 9B3F CD45 B1C8.

Mullender[60]:

> Names form a hierarchy identifying objects at different levels of the system, so it is possible that an object is referred to by different names at different levels of the system. · · · At one end of the spectrum, the name of an object could be a composed of the name of the processor where the object resides, the name of the process that manages the object, and a number used by the process to refer to the object. At the other end of the spectrum, the name of an object could be a character string chosen by the owner, and interpreted in the owner's environment. The first example shows a name that can be conveniently used by an interprocess communication mechanism. · · · The second example shows just the opposite; the name is convenient for human users: it can be independent of what service process currently maintains the object, and of the location of the object.

Here we will consider a name to be a method by which some resource may be referred to, or specified. This is effectively Mullender's view, and is compatible with the others listed. The existence of names allows the resources to which they correspond to be shared, and allows resources to be referred to with human-mnemonic names. Furthermore, by carefully controlled name creation, it is also possible to guarantee that a given name refers to the same valid object, or to nothing at all – this is Needham's view. Shoch's concept of a name is a subset of those we consider – names, addresses and routes all represent names by the chosen definition. Since we view a name as being a mechanism by which 'something' may be specified, anything could, potentially be named, and hence Mullender's, and our, view could be said to encompass Watson's view as well.

The chosen definition of a name encompasses program variables, memory addresses and communication channels. Although in this chapter the term name is used in accordance with the definition above, elsewhere in this thesis it will be used only to refer to objects created by, or manipulated by, the tasks that comprise the Namer in *Wisdom*. Thus, in effect, the names that the Namer supports correspond to the subset that Shoch labels as names.

Any namer should provide a basic set of operations: the most important is that of lookup which finds the required object, but also needed are delete, to remove a named object, and add which adds or creates a named object. The behaviour of add and delete operations are similar to that of lookup – the differences are the error conditions and the behaviour on finding the service: in lookup if the service is found a connection is returned otherwise an error occurs, in add if the service exists an error is returned, otherwise it is added, and in delete if the service does not exist an error is returned, if it does exist it is removed. Since the behaviours are the same – the differences are in the states which generate errors – this chapter will deal only with the action of lookup.

Naming is often considered, or modelled, as a string translation problem: the action of the namer is to convert a string to some form of access mechanism. Although this is often the way that namers are implemented, it is more accurate to describe namers as functions providing a mapping from some form of high order name to some lower order name (the lowest order names can be viewed as those supported by the address space of a task, or the number of a disk block).[†] This is also far more accurate given the definition of names with which we are working.

Naming – the service of service location – has developed from one of two areas. The first is that of the filesystem: here names refer to files, places of permanent storage of information, and sometimes to peripherals, such as printers. The second area is electronic mail: here names refer to mailboxes, or destinations for messages, and in some cases to the paths the messages should take. A good naming

---

[†] It could be argued that even these are not the lowest: the task's address space may be mapped by a memory management unit on to another address, and the disk block is mapped into a physical location by the disk controller.

scheme should be able to support either of, or both, filesystems and electronic mailing as well as the ability to locate and access other services.

There are similarities between these areas since both are expected to support relative and absolute naming. In mail systems relative naming means sending mail to ‘‘user’’ results in the mail going to ‘‘user’’ on the local site, and not to a remote site which also has a ‘‘user’’. This is the same as selecting the ‘‘file’’ in the current directory as opposed to some other ‘‘file’’ elsewhere in the filesystem. Absolute names in a filesystem specify a unique file, and *always* refer to the same file; more recent electronic mail systems have started to support this, but problems with the very distributed nature of these systems have made it difficult to guarantee this (primarily since each namer maintains its own independent, and potentially incorrect, list of names). Although mail systems are sometimes viewed as providing a flat name space they are, usually, structured since in user@site, site is broken into a hierarchy such as country, community location and site. When the user wishes to avoid articulating certain parts of the hierarchy defaults are supplied by the mailer.

Filesystems and mailers are becoming increasingly overlapped, and in some cases nearly unified. Grapevine[80] is, basically, a network mail system: messages are delivered to ‘inboxes’. However the ‘inboxes’ have been expanded to cover other services and, even files or filesystems.[†] In a similar manner distributed and network file systems have resulted in mailboxes at remote sites being identified by their pathnames[22], hence the filesystem becomes capable of supporting network mailing.

The principles that make for a good naming scheme are well known, even though they have not always applied; in essence they rely on a simple, uniform syntax – the original UNIX filesystem supports such a name space, however VMS, or some of the modifications made for BSD UNIX,[‡] do not. The advantages of a simple syntax are that it makes expansion easier, and interpretation of a name clearer for both human and computer. Pike and Weinberger[71] have produced a very good paper which deals with these issues in both the areas of filesystem and network mailing, and this is recommended as giving a good insight into the practical aspects of these principles.

Pike’s principles of good naming (as set out in his paper), and the requirements for the Namer that were set out in chapter 2 (supporting location transparency, parallelism, scalability, extensibility), give the following objectives for the Namer:

- *Location Transparency*: The Namer should support methods of accessing services so that the physical location of the service is irrelevant, and so that the location can change if necessary.
- *Homogeneity*: The Namer should support a uniform name space which can be described in a simple syntax.
- *Extensibility*: It should be possible to extend the naming mechanism to support, for example, remote services.
- *Distributed and Parallel*: It should be possible to spread the tasks that comprise the Namer out among the processors that comprise the system.
- *Scalable*: It should be possible to increase the number of requests the Namer handles and processes without altering it – in essence it should not become a potential bottleneck. This can be considered as a more general requirement of which the previous one is a part, or simply an implementation technique.

[†] This is a slightly generalised view, though the ability to move files, and even filesystems, between remote machines is provided by Grapevine via its registry service (which is effectively the namer).

[‡] Specifically the inclusion of the ‘‘:’’ and ‘‘.’’ notation that was handled by some commands for accessing remote machine’s filesystems.

A knowledge of the theory underlying naming is also essential since it allows reasoning about names, their interpretation, and the expansion of the name space, to take place, and can thus result in a more general and predicatable naming space. Since such theory and formalisms have only just appeared and are still in their infancy, a simplified version will be presented here that will be used as a base from which the Namer in *Wisdom* can be described.

The remainder of this chapter will be divided into three main sections. The first will describe a current theory of naming, which is based on the work by Peterson and Comer[68], though some modifications have been made to this for the purposes of using it as a base to describe the Namer in *Wisdom*, and to give definitions for terms in naming, and to produce a simpler and more uniform treatment of the name translation process. The following section then covers the major issues in naming. Finally the Namer as implemented for *Wisdom* will be described both in terms of the implementation and the naming theory that was presented earlier.

4.1 Naming Theory

In the other main areas dealt with by this thesis (routing and load balancing) there are supporting backgrounds of theory and definitions which provide a framework which can be used to describe and understand these areas. Terms such as deadlock, datagrams and store and forward are well understood, with queuing theory providing formalisms for analysing message delivery and task scheduling. Unfortunately, naming does not have a similar background with tight definitions that can be used to describe a namer. There are terms that may be applied to naming systems, but they have developed from various implementations and refer to those implementations rather than giving general terms. Consequently many terms have developed which have the same, or similar, meaning (e.g. directories and folders) or which have unclear meanings.

Work has been undertaken which tackles these problems and provides a set of formal definitions which aid both understanding and reasoning about naming schemes. Initial definitions were provided by Watson[90] based on the translation of names from a higher to lower order. More recently Peterson and Comer[68] have developed notation and tight formal definitions describing a translation between names and simple names (which are effectively addresses).

The theory and definitions which follow are based on the work by Peterson and Comer, but have been somewhat simplified since the full definitions (and their flexibility) given are not necessary for the purposes of discussion and description to which the theory is to be put. Also, within the context of the Namer in *Wisdom*, the additional theory covers either situations which cannot arise, or which can be handled by the remaining theory. These omissions and simplifications should *not* be taken as complete, or that the following theory has all the power of the original. They are only for the purposes of removing unnecessary complexity from the following discussion.

Throughout the following descriptions the following terms will be used

- *object*: that which has a name; in practice a named service of some form.
- *target object*: the object to which a name refers, i.e. that which will be found by the name resolution.[†]
- *source object*: the object from which the name resolution started. Although it would be valid to consider any object involved in the action of resolving the name as the source object, only the very first object is the source object.

[†] The term resolution will be defined formally later: informally it is the act of locating of an object based on the name given.

- *knows*: A source object is said to *know* a target object if it has a valid name for that target object.

4.1.1 Definitions

A <*nametoken*> is the simplest possible name, or component of a name, that can be given to an object. If strings are being used as names then <*nametoken*> is an arbitrary string, containing any character but the delimiter, ||. If numbers are being used the <*nametoken*> is a single number, with the delimiter being implicit in the separation of the numbers. A *name* is defined as a sequence of delimiter separated nametokens, and is represented by the regular expression:

$$<name> = (<nametoken> \; \| \;)^* \; <nametoken>$$

N is the name space, which is the set of all possible names, and has the following definition:

$$N = \left\{ <name> \,|\, <name> = (<nametoken> \; \| \;)^* \; <nametoken> \right\}$$

A name, *n* is said to be a *simple name* if it comprises of a single <*nametoken*>, and this is denoted by *simple(n)*. Two other functions are also defined on names: *leftmost(n)* returns the first (leftmost) component of a name *n* and *rightmost(n)* returns the last (rightmost) component of name *n*.

The naming network is the directed annotated graph $G = (V, E)$. V is set of all named objects in the system. E is the set of directed edges (u,v) annotated by n. $(u,v) \in E$ if and only if object u knows object v by the nametoken n. If objects $u,v \in V$ are linked by the edges $e_1, \cdots, e_n \in E$ then u knows v by the name $n_1 \| \cdots \| n_n$. u is then said to be able to access v, since a *path* exists between them, which is denoted by $p: u \rightarrow v$.

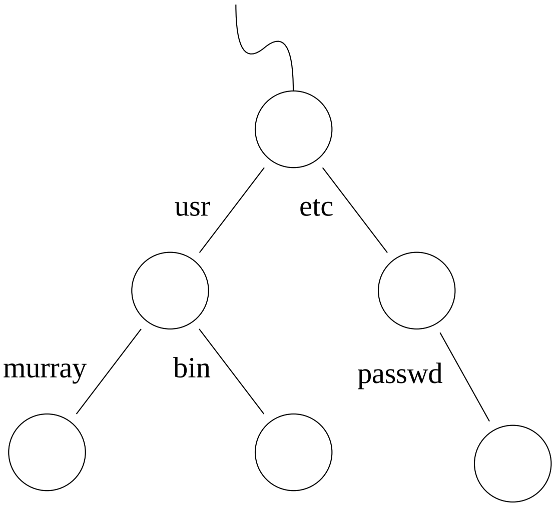


Figure 4.1: Example naming network graph

Figure 4.1 shows an example of the graph G. The objects, V, are represented here by circles, and the edges, E, are represented by the lines. In this the set of names, N, comprises of the eight names that can be generated by reading the annotations of the edges whilst tracing a path through the graph, with each name being separated by the appropriate character, in this case `/`. The eight names are: `usr`, `usr/murray`, `murray`, `usr/bin`, `bin`, `etc`, `etc/passwd` and `passwd`

The *name resolution mechanism* is the method by which a name is converted into an entity which allows access to the resource to which the name refers. This is denoted by Φ and is defined as Φ: $V \times N_{higher} \rightarrow V \times N_{lower}$. This states that given an object and a name Φ converts this to a lower order name which accesses another named object − the target object. If $\Phi(u,n) = (v,n_{lower})$ then n represents a path $p: u \rightarrow v$ and u knows object v by name n. The v returned by Φ represents the object named from u by name n, and n_{lower} represents a lower order name that may be used to access this object (potentially via further name resolutions).

A transition function, β, used in the process of name resolution, is defined as: $\beta\colon V \times N \rightarrow V \times N \times N_{lower}$ and is written as $\beta(u, n) = (v, n', n_{lower})$. The values returned by β have the following interpretation: n' is the name formed from n by the removal of a nametoken, v represents the object named from u by the nametoken removed, and n_{lower} is a lower order name for v from u. This states that β maps an object and name to a new object, a new name and a lower order name, e.g. mapping 3.0.2 and `usr/murray` on to 3.12.2 and `murray`, as is the case in figure 4.2. Thus β can be considered to resolve a single nametoken, that is it accepts a starting object and high order name, from which it removes a single nametoken, and returns a low order name, a new object from which to continue the resolution, and the remainder of the high order name. Note that this lower order name could be null, or comprise of many nametokens.

Based on the above definitions Φ is constructed from repeated applications of β until the high order name has been completely resolved and a lower order name produced (which may comprise of zero or more lower order nametokens). Also by use of the above definitions all the edges in naming network, G, can be considered to be annotated with two names. The first is the high order name, and the second is the lower name which is returned by β. It would be more general to state that the second annotation of the graph is, in fact, a set of names since this would allow for the mapping of a single object on to a number of implementations or instantiations, effectively allowing for replication. However, replication is not being considered here and so it is deemed sufficient to deal with a single lower order name.[†] The presence of multiple lower order names corresponding to a single higher order name can result in the actions of β being non-deterministic since it now has more than one valid output for a given input.

Figure 4.2(a) shows the same name space as displayed in figure 4.1, but it has now been annotated by the lower order names generated by applications of β and the corresponding lower order name space. Figure 4.2(b) shows the lower order names, and hence the name space that figure 4.1 is translated into by the applications of Φ. It also shows three new objects that have names in the low order name space, but not in the higher one, and hence were not visible to it.

The final definition is that of *closure*, which selects an initial named object from which the resolution may start. Its definition follows.

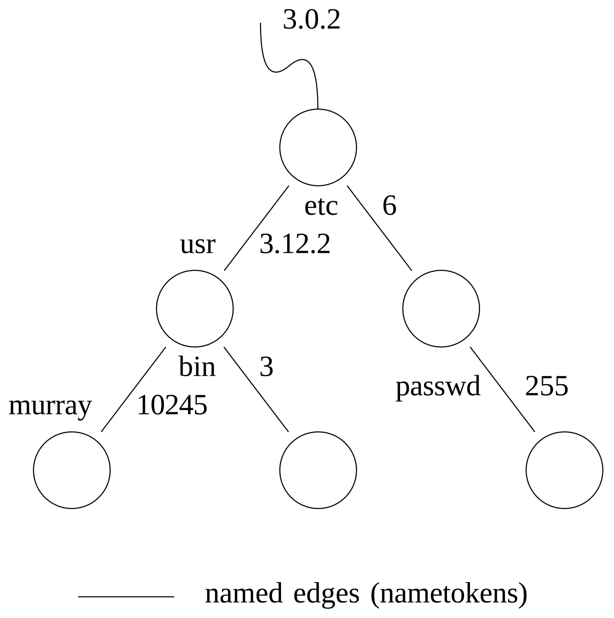
$$closure\colon V \times N \rightarrow V$$

The use of *closure* will become clearer when the properties of namers are discussed. Its purpose is to act as a switch to allow the selection of a source object dependent upon the name, thereby allowing absolute and relative names to be implemented in the same namer. In Peterson and Comer's paper, *closure* was used to generate a context from an object, and the name resolution involved transitions between contexts, rather than between objects as it does in the simplified version.

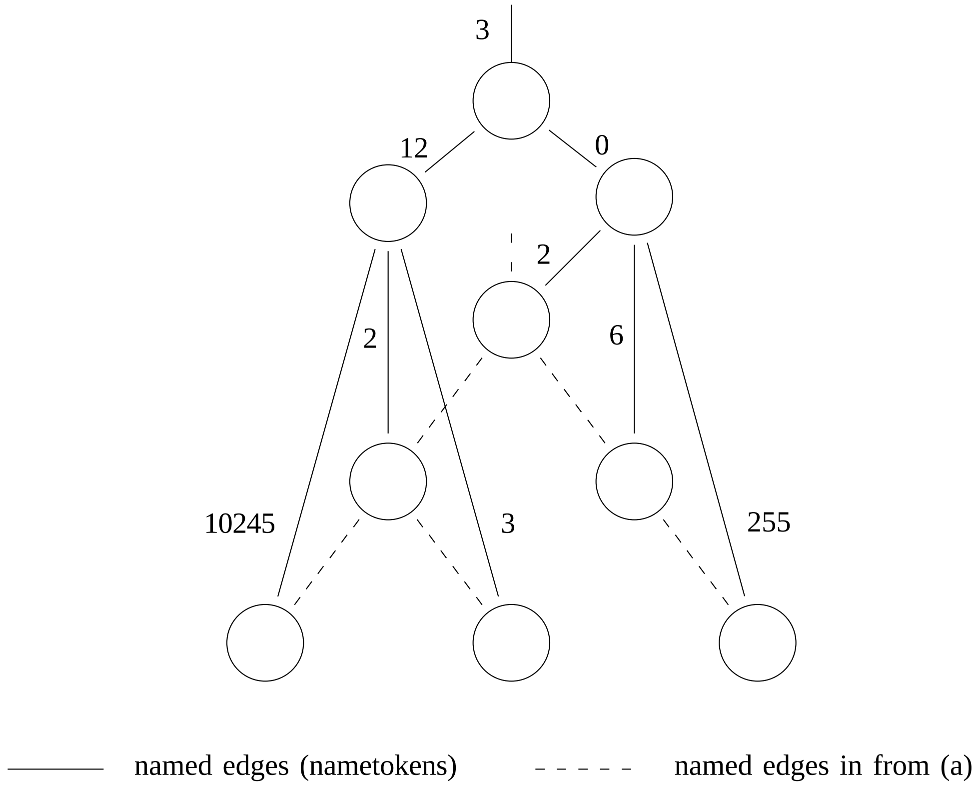
4.1.2 Contexts, Directories and Objects

This section deals with the differences between Peterson and Comer's paper and the simplified verion of the naming theory that is being presented. The first of the two main alterations was to remove *primitive names* and *primitive name spaces*, which represent what are now termed lower order names, and replace them with a single, more general definition of names and name spaces, and the subsequent altering of the requisite functions and definitions. This allows one to deal with names and name spaces in a more uniform manner, and removes the need for two different descriptions of names depending on their ‘‘level’’. It is also more in line with other work, such as Watson[90], and closer to the behaviour of actual naming systems. The second main alteration is the removal of *contexts* which Peterson used to represent the flexibility of the name resolution mechanisms. However, all contexts can be reached by a name, so here they are treated as named objects.

[†] There are instances besides replication when multiple low order names would be desirable, specifically where is single object is implemented by multiple components (e.g. many disk blocks making up a file), but for simplicity these will be dealt with as a single entity.



(a) Annotated higher order naming graph



(b) Lower order name space

Figure 4.2: Translations Between Higher and Lower Order Name Spaces

In the original paper, Φ is said to have a set of states, or contexts through which it passes as it resolves the name, the set being denoted by C. This results in the definition for β as being $\beta: C \times N \rightarrow C \times N \times N_{lower}$, and is denoted as $\beta(c,n) = (c',n',n_{lower})$. The original definition of $closure()$ was $closure: V \times N \rightarrow C$ which provided an initial context from which Φ may operate.

Under the original definitions of β, c' can be said to be named from c by the name, or part thereof, n. In fact the original paper describes a name space as being the graph produced using the contexts as vertexes and the transitions defined by a restricted version of β as the edges. The restriction is that the edges are defined by a homogeneous application of β (see later), which means that all edges are labelled by a single nametoken, as in the case of our definition of G above.

If an application of β is non-homogeneous then it either prunes off more than one nametoken, adds nametokens, leaves the name unaltered or performs a translation of the name. Removing more than one nametoken is the product of caching or prefix tables: this is equivalent to knowing in advance the results of resolving the part of the name that is removed, hence removing more than one nametoken can be considered as repeated applications of β which take it through the multiple nametokens one at a time.†

Adding nametokens is equivalent to the action of symbolic links which state that an object is known by another name, and the resolution of this other name will render the desired object. The difference between symbolic links and ordinary links, which are simply names, is the additional level of indirection provided by the symbolic link: the name provides the lower order name directly whereas the symbolic link requires that the new name be resolved to produce the same. This has the effect of increasing the location transparency of the object. However, for the purposes to which the modification of the original theory is being put, it can be considered to be equivalent to a single nametoken resolving to a lower order name, but that the lower order name is achieved by recursive application of β rather than table lookup.

Translating a name can be viewed, or modeled in a number of ways depending on how the name is altered. If the name is converted into another of the same order, that is one in the current name space, then the actions can be viewed as an application of caching to remove the existing name, followed by a symbolic link to add the new name. If the name is translated into one of a different order, that is one other than the current name space, then the actions can be considered as translation to a lower order name, that may need further resolution to obtain the target object.

The only remaining action that β can have when it is non-homogeneous is that of outputting a low order name without altering the name, i.e. $\beta(v,n) = (v',n,n_{lower})$. This represents a service being implemented by many objects: the current object (v) is unable to identify the named object (or perform some action) so it passes this on to another object (v') to identify (or perform the requested action). This does not form part of the defined name space since the arcs between the objects are not labeled with a nametoken, hence this form of non-homogeneous action cannot be handled by the reduced version of the theory described here. If, however, the theory were modified to cope with multiple low order names as the output of β, rather than a single one as is the case currently, then this case could be handled. Given the uses to which this simplified theory is being put (semi-formal description) this is considered unnecessary – the situation cannot arise with the *Wisdom* namer.

4.1.3 Properties of Namers and Name Spaces

The above definitions allow formal definitions for many properties of namers and name spaces. In the following $names(u,v,n)$ means that object u knows object v by name n. This can be defined by: if $names(u,v,n)$ then $\Phi(u,n) = (v,\ n_{lower})$ where n_{lower} labels a path $p : u \rightarrow v$.

The above definitions state that once a name has been resolved it is ‘lost’ – there is no way in which a previous component of a name can be determined to help the resolution of the current nametoken. Some information may be present since the object currently resolving the name must itself have been named, but it is not possible to determine this name.

---

† Some systems (such as V[25]) perform similar operations, but do not require that the objects named by the intervening name tokens actually exist: resolution is performed by broadcasting the name and awaiting replies. Conceptually, however, the intervening objects can be considered to exist since the system behaves as if they did.

Homogeneity

A named object implements a homogeneous naming, if, for all successful applications of $\beta(v, n) = (v', n', n_{lower})$, exactly one of the following holds for all the names.

(i) $n = \mathit{leftmost}(n) \,\|\, n'$

(ii) $n = n' \,\|\, \mathit{rightmost}(n)$

(iii) $\mathit{simple}(n)$ and $n = n'$

That is, an object implements a homogeneous naming if all applications of β either prune a nametoken from the name or preserve the name given that it is a simple name. In practice the first two (cases (i) and (ii)) are common in hierarchical filesystems, whereas the latter (case (iii)) is more common in electronic mail systems. It is also clear that for cases (i) and (ii) given a finite name that Φ will terminate, since each application of β removes part of the name.

The property of homogeneity is one of the major aspects of a good naming scheme[71]. It represents a simple syntax, the expandability of the name space and the absence of special, or context dependent, interpretation of names.

Absolute Names

A name space is said to support absolute names if the following holds:

$$\forall\, u, u', v, v' \in V \quad \mathit{names}(u,v,n) \wedge \mathit{names}(u',v',n) \Rightarrow v' = v$$

That is, regardless of the starting object, the name n refers to the same object, e.g. inodes within a single filesystem. Absolute names can also said to be supported if

$$\forall\, u, u' \in V,\ \mathit{closure}(u,n) = \mathit{closure}(u',n)$$

i.e. if the starting object is the same for all resolutions of a name, then the name is absolute. Though this is less general than the first definition, it is the way that absolute names have been implemented in filesystems. In mailing systems, however, this is not the mechanism used, and instead a combination of a complete graph (i.e. all namers know all systems) and unique names has either been used or been simulated.

Relative Names

A name space is said to support relative names if the the following holds:

$$\exists\, u \neq u',\ u, u', v, v' \in V \ \ \text{such that}\ \ \mathit{names}(u,v,n) \wedge \mathit{names}(u',v',n')$$
$$\Rightarrow (v \neq v' \wedge n = n') \vee (v = v' \wedge n \neq n')$$

This states that the same name, when applied from two different objects, refers to different objects, or that a different name applied again to different objects refers to the same target object. The latter case is not equivalent to synonyms since the starting objects differ. An example of the former would be the file name `.profile` in UNIX – the file found will depend on the current directory, or source object in the theory terminology. This can also be stated as the converse of the second definition for absolute names: $\forall\, u, u' \in V, \forall\, n, n' \in N\ \mathit{closure}(u,n) \neq \mathit{closure}(u',n')$.

Synonyms

A name space is said to support synonyms or pseudonyms, nicknames and aliases as they are sometimes termed, if the following holds:

$$\exists\, u, v \in V \wedge n \neq n'\ \text{s.t.}\ \mathit{names}(u,v,n) \wedge \mathit{names}(u,v,n')$$

This states that a synonyms exist where two (or more) names when applied to the same starting object, refer to the same object. For example a user with two or more mailboxes on the system, such as root and postmaster.

Unique Names

A name space is said to support unique names if the following holds:

$$\forall\ u,v \in V \wedge \forall\ n,n' \in N\ \ names(u,v,n) \wedge names(u,v,n') \Rightarrow n = n'$$

This states that unique names exist where there is only one name by which a given object is known by another given object. It should be noted that the name is unique only with respect to one specific source objects: differing names, when applied to differing source objects could result in the same target object without violating the uniqueness criteria above.

Routing Name Spaces

A name space is said to be a *routing* name space if the lower name space is isomorphic with the higher order name space from which it was generated. There are two forms this can take, *source-routing* and *implicit-routing*. Source-routing can be defined by placing the following constraint on β:

$$\beta(v,n) = (v',n',n_{lower}),\ simple(n_{lower}) \wedge (n = leftmost(n) \parallel n' \vee n = n' \parallel rightmost(n))$$

This means that every nametoken of the name maps on to a single lower order nametoken, and hence the name directly represents the route from source to destination. Among the better known versions of this are the old UUCP style names: site3!site2!site1!user.

In implicit-routing all source objects identify a destination object by means of the same name,† such as JANET style addresses (e.g. uk.ac.york.minster always corresponds to the mail machine at the University of York). These names are either resolved directly on to the target object, or on to another object (with the name unchanged) for final resolution, i.e. $\beta(v,n) = (u,n,n_{lower})$. This represents the current object stating that, although it does not know the object in question, it knows another object which will be able to resolve the name correctly. Implicit routing will not be considered further here, since it requires a non-homogeneous resolution: the name does not have nametokens removed, and there is no guarantee that it is simple.

Comments about the Definitions

The above definitions are very subtle; it is obvious, and true, that a namer cannot directly support both synonyms and unique names. Where unique names are supported $names(u,v,n) \wedge names(u,v,n')$ implies that $n = n'$ must always be the case, which contradicts the requirements for synonyms. Likewise both relative names and absolute names cannot be supported: if absolute names are supported then $names(u,v,n) \wedge names(u',v',n)$ implies $v = v'$ at all times. However, relative names require that the same precondition implies, at least at some points, that $v \neq v'$ (by replacing n for n' in the definition), so both styles cannot be supported.

It is, however, not possible to equate the definitions in other ways. A unique name merely means that there is a single name from one source object to one destination object: it does not imply that the same name is used by all source objects to refer to the same destination object: that definition is for absolute names. Likewise relative names specify that two different source objects use different names to refer to the same object, or the same name to refer to different objects: it does not say that the same source object can refer to the same destination with different names: that is the definition of synonyms. Also, a namer may support both synonyms and absolute names: there may be more than one name referring to the destination object; and it can support both relative and unique names: every object knows other objects by a unique name, but this unique name may be re-used by other source objects to refer to differing destination objects.

---

† In fact the Router described in the previous chapter represent an implicit routing name space. The names are the processor addresses, and the Router either delivers to the named processor, or passes the request on to another processor that will be able to deliver the message.

In practice, however, name spaces support both absolute and relative names. This apparent contradiction is generally handled by a switch of some description, which is represented by the function *closure*. This does *not* result in both being supported, but rather provides a mechanism by which the two forms can be switched between – it can be viewed as a mechanism for supporting two name spaces. By allowing the starting object to be selected based on the name, it is possible to provide both forms of naming by inspecting the name. The most common approach is to use an initial 'component' of the name (where the name is a string, then the first character is commonly used) to specify that the name is absolute and therefore to choose a common starting object. The presence of this component, or character, gives absolute names, and its absence generates relative names; in UNIX this is done by the presence or absence of a leading `/`. In mailing systems the starting object, i.e. that returned by *closure*(), is often determined by the number of components present in the name: ''user'' means the local user, ''user@machine'' means a user at the local site, but on a differing machine, and so on.

4.2 Issues in Naming

There are several areas of concern in the field of naming which are not covered by the above theory. The reason is that most deal with issues of implementation, such as methods and speed of name resolution, or the style the names should take, rather than the meaning, interpretation and reasoning about names that the naming theory described above gives.

4.2.1 Name Resolution Mechanisms

The name resolution mechanism alluded to in the theory, has the nametoken used to index into a table, either by comparison, hashing or some other mechanism. This table entry then provides the lower order name needed for the resolution to continue (or the failure of this indicates that the resolution has failed). This, however, is not the only way, and several other mechanisms exist, especially where other considerations such as reliability become important. They are also solutions where the namer does not support a routing name space, or cannot know the lower order name to which it must map the name it is resolving.

One mechanism, as used by the V system, involves broadcasts or multicasts. In this, when a name is to be resolved, the namer broadcasts or multicasts the name and awaits a reply. All services listen to messages of this kind and when they receive a lookup which contains their name they reply with their 'address' or lower order name. The namer then accepts this reply and uses it to generate the lower order name to complete the resolution. This resolution mechanism relies on there being support for broadcast and multicast messages. Severe problems exist with scaling this mechanism (as detailed in chapter 3) as the network can quickly become swamped by broadcasts trying to resolve names. The solution is to use extensive caching: when a namer receives a reply it adds it to its cache table, and before sending out a resolve message it checks its cache to see if it already knows the name. If it does know it, then it avoids sending the resolve message, and instead uses the information in the cache. Even when extensive caching is used the size of the system is still limited to approximately one hundred machines using a standard ethernet network[29]. Hence the solution to the problems of broadcasting is to adopt a name resolution mechanism closer to that alluded to in the naming theory.

The Amoeba system has encountered similar problems when connecting many Amoeba systems together using wide area networks. Firstly wide area networks do not directly support broadcasting and this is expensive when simulated by sending messages to each site. The solution to this uses a mechanism called publishing[76] which involves names registering themselves with a special (name) server which contacts the other Amoeba systems and 'publishes' the information about the new name to these systems. This can be viewed as a special namer which is distributed over many systems to allow a limited combination of the name spaces.

A mechanism, which provides characteristics similar to those supported by the broadcast mechanisms, for name location in meshes has been described by Mullender[60], a simpler version of which was originally designed for the Stony Brook Microcomputer Network[39]. This involves announcing a name along one axis, and searching for a name by querying down the other axis. Figure 4.3 shows an example of this mechanism. The processor labelled 2, has, running on it, a task which wishes to provide a named service. This processor therefore sends this name out along the x-axis, as represented by the dashed line, and all processors receiving this message then take note of it. At some later time a task on processor 1 wishes to use this service, and so a lookup request is sent along the y-axis, as represented by the dotted line. The processor labelled with the asterisk (*) recognises the name, and returns the location of the service. In a torus, the network used by the Stony Brook system, matters are simplified as messages for name addition or resolution need to be sent only in one direction and when received back represents either that all processors know the name, in the case of name addition, or it contains the result of the name resolution.

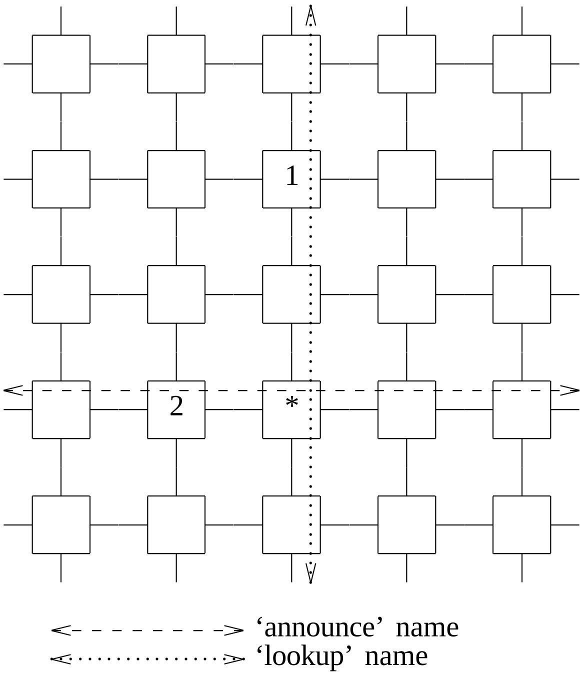


Figure 4.3: Stony Brook Name Location Mechanism for Meshes

4.2.2 Caching and Prefix Tables

In many cases caching[87] has been used to improve the performance of the name resolution mechanism, or to remove or reduce the overhead of extensive IMC, as with the broadcast mechanisms described above. These performance improvements have become necessary since the resolution mechanisms have been limited in speed by the time taken to communicate with other parts of the resolution mechanism, or to acquire the necessary information from backing store, or in the high load that communication they perform places on the network. The action of caching simply involves remembering names that have been resolved and the low level name(s) which the namer, or service, returned. When a new lookup request occurs, the local list of known names is consulted before resolving the request in the ordinary fashion.

A version of this mechanism involves prefix-tables[92] in which only part, instead of all of, the name is remembered. The mechanism works by comparing the names being resolved with those in the prefix table, and the longest match is then used. For example if a prefix table contains the values `/a`, `/a/b`, `/a/b/c/d`, and `/h/c`, and the name `/a/b/e` was being resolved, this would match with `/a/b`, to which would be sent the name `e`, whereas `/a/b/c/d/q` would result in `q` being sent to `/a/b/c/d`. In practice the name cached corresponds to the root of a set of names implemented by a given service, thus in the example the names given as cached would correspond to services, rather than some name within a service.

The area of caching has not been included in the theory, however this could be done by one of two methods. Firstly closure could be redefined so that it encompasses the actions of the caching mechanism. In this the definition would then become $closure: N \rightarrow N \times V \times N_{lower}$, i.e. the closure mechanism would modify the higher order name, and return a lower order name and corresponding object. The other method of dealing with caching involves treating the cache as a namer. This is less clean since the behaviour of the cache, and the names it can resolve, are determined by the previous actions of the task(s) to which it is associated.

4.2.3 Attribute-based Names

Peterson[69] has proposed a naming system in which the individual names do not correspond to a specific service, but rather to the type of service or the functions that the service provides. To compare with the telephone book, an attribute-based name is one which occurs in the yellow pages (e.g. ‘‘plumber’’) rather than one which occurs in the white pages (e.g. ‘‘Jo Bloggs and Daughter’’). Thus with attribute based naming you would request ‘‘postscript laser printer’’ rather than, say, ‘‘apple0’’ and the namer would return the closest match.

The behaviour is similar to that described in the naming theory: the name is matched against a list of known services, but there is, unfortunately, a subtle difference. The resolution mechanism is no longer deterministic as there is now, potentially, a one to many mapping between the yellow-page style names and the services and no way to determine which will be selected. Therefore, it would be appropriate to consider attribute based naming as an alternative, albeit non-deterministic, name resolution strategy.

4.2.4 Scaling

The namer, especially one which supports absolute names, is a potential location for a bottleneck. The reason is that, when a system grows and gains more tasks, these tasks are very likely to require the services of the namer to locate other services they may need. This means that as a system grows the namer will become more heavily loaded, and unless it can be scaled with the system it may become the limiting factor in the speed of the system. Matters are made even worse when absolute names are provided since this requires that there is a single point through which all the name resolutions must pass, unless all objects know all others.

There are many solutions, of varying complexity, to this problem. One method involves caching, which has been described above, to reduce the number of resolutions which must be handled by the namer. Another solution is to replicate specific namer tasks, and to send requests to any of these namer tasks, however replicated namers have extensive problems with maintaining consistency, which may restrict their use. It is also the case that some of the services which the namer identifies may also become heavily used, therefore requiring their own replication, which in turn requires support from the namer to share the load between them. Many of these problems have occurred in the area of replicated filesystems, and solutions for these problems have been proposed which should be applicable to this area[72].

The other place in which the namer can limit the scaling of a system occurs when the namer must support many connections. If the underlying IMC mechanism is exclusively one-to-one then the namer may end up with very large numbers of IMC connections it must listen to. It is also possible that the IMC mechanism will place constraints on the maximum number of connections and this, too, will limit scaling. The solution involves the use both of many to one connections so that the namer need listen at only a single point, or limited number of points, for requests. Secondly allowing tasks in general to maintain and manipulate their IMC connections will allow greater flexibility over the number of connections that a task or namer can support – this is best supported by capabilities. The advantages of capabilities are that they are entirely controlled by the task, which can modify, duplicate or send them to other tasks as specified by the capability[61].

4.2.5 Location Transparency

One of the important features of naming is to provide location independence – that is the service to which a name refers may move, change or be replaced but the name will still remain the same. Needham states[64]

> If we wish to have the flexibility to move the representation of an object from one place to another and still have it satisfactorily shared between computations, then the names used by these computations must be, or resolve into, a form of name that is location independent and which can itself be resolved to find the current physical location information.

In essence this means the name used must be location independent and then transformed into the physical location of the service to which it refers when required. This dynamic transformation, or binding, is the service that the namer provides, and hence it supports location independence – the higher order name used by the computations can be transformed by the namer into any lower order name which represents the location of the service.

In practice each translation through which a name goes represents another layer of location hiding and hence location transparency. Also, if name resolution only occurs at the start of a computation, the service which that computation uses is fixed to the location returned by the namer until such time as the computation ends. If, however, the name, or some form of it, is resolved on each communication with the service, then the service may move during the lifetime of the computation.[†]

4.3 Wisdom Naming

If the definitions given in the section on naming theory are used then *Wisdom* can be said to have several namers and name spaces. The Guardian provides a flat name space with its mapping of the tasks channels to network addresses and the Router provides a non-homogeneous implicit-routing name space with the location-dependent names being the task addresses. However, we will now only be concerned here with what has been called the Namer module, the component of *Wisdom* that provides a method of identifying services, locating and accessing them.

4.3.1 Overview of The Wisdom Namer

The Namer used in *Wisdom* is an amalgamation of ideas, both original and from other systems, which have been combined to achieve the objectives listed at the start of this chapter.

The Namer runs as a collection of ordinary user tasks; it has neither special privileges, nor abilities. This has been used in many systems: the LOCUS filesystem[72] is so implemented, the namers in PULSE[44] and MINIX[85] also use the same approach. The advantage of this approach is that it allows greater flexibility in the namer tasks, and allows users to build their own namers, without necessarily interfering with the namer tasks that implement the remainder of the name space.

Each namer task in *Wisdom* receives a name, typically represented as a string, and matches that string to those it knows about (effectively it performs a table lookup). This results in an IMC channel down which it sends the remainder of the request. The concept of evaluating an name, and sending the remainder (the unresolved part) down some communication channel comes from the eighth edition UNIX work on streams[77].

[†] This has implications for dynamic load balancing in which tasks may move during their lifetime: the IMC connections must provide location transparency so that messages will still arrive at the new location. The solution adopted to this in DEMOS/MP[58] is to have ‘stubs’ which are left where the end of an IMC channel used to be. The stub indicates where the new end of the channel is, so the IMC connections are no longer location dependent.

Each namer task itself provides no structure, all it implements is a flat name space that performs the mapping described above. This has the advantage of making the individual namers simple, easy to analyse, and allows a direct mapping between the structure of the namers and that of the naming theory given above.

Many namer tasks are combined together to provide a structure to the name space. This is done by one namer, the destination, connecting itself to another namer, the source. When the source receives a name for the destination, it removes the part of the name which corresponds to the destination, and sends the remainder to the destination. Thus a structured name space is built up.

This approach gives highly parallel, distributed name resolution, which is necessary for the requisite scaling of the Namer. It should be noted that the above is a conceptual view. Realistically some of the namer tasks could be expected to be combined together to reduce the parallelism to more manageable quantities, and to reduce the number of messages sent.

The above design results in a decentralised, parallel Namer. This differs from single centralised systems such as ordinary file systems or the naming service in the Cambridge Distributed System[63]. It also differs from systems such as Grapevine[16] in which the many namers know all the names, in effect multiple central name servers. The closest are systems such as V[25] in which the various components of the name space know of only a part of the name space, though V also uses broadcast location and caching, and is capable of surviving failures – issues which have not been addressed in the *Wisdom* Namer. The Namer is decentralised as there is, in the description, no single component namer task which either controls the others, or knows all the names. Since it is comprised of many components it is also parallel as these components are able to resolve names independently of one another.

4.3.2 Analysis of The Wisdom Namer Using Naming Theory

Each namer task, as described earlier, implements a single named object as described by the theory. The combination of these tasks, the Namer, implements the naming network, G, with the leaf nodes supplied by other tasks. All the names in the naming network, both of the component namer tasks themselves and the tasks which have announced themselves to the namer tasks, represent services which user tasks may wish to access. These user tasks are potentially unnamed[†] and hence not objects in the name space; to gain access to the naming network they contact an appropriate namer and requesting it to perform the name resolution on their behalf.

In the above theory it was assumed that all objects were named, however in *Wisdom* this is not the case, as exemplified by tasks. This difference can easily be described by modifying the definition of $closure$ to the following $closure : N \rightarrow V \times N$. Thus any task that needs the services of a namer does so by using the closure function, which selects the appropriate namer (V), and will pass on to it the name. In practice this is implemented by closure selecting one out of a set of connections to different namer tasks passed to the child by its parent when the child was created. This allows both relative and absolute names to be implemented, given that a namer connection to some absolute point (the root) is passed to the child: in a UNIX style name space, names beginning with / would be passed to the root, and the others to another namer which represents the tasks current directory.

Each conceptual namer task implements β, the transition function: it accepts a name (n), which together with itself (v, an object), form the inputs to the definition of β, the outputs are the new object v' to which the namer sends the remainder of the name n' over the channel n_{lower} it has identified as being associated with the name. The implementation of β in *Wisdom* is homogeneous – removes leftmost component of the name, i.e. $n = leftmost(n) \| n'$.

[†] Note that need not be the case – in some UNIX systems all tasks are named from a directory (or object) called `/proc`, or in the case of *Wisdom* they could be named through their parents.

The advantage of the homogeneous approach is that it can be shown that the name resolution process must terminate. A name must be finite, and each pass through a namer results in the name being shortened, hence eventually the name is completely resolved.

When the namers are combined, by passing a name through a sequence of them the name is resolved. This is equivalent to an application of Φ which is built up of repeated applications of β, which is implemented by the namers. The namers, through which the request passes, represent the named objects which comprise the name, and so represent the path through the name space which the name described.

4.3.3 Properties of Wisdom Namer

The *Wisdom* Namer, by means of its design and the definitions given above can be shown to support, or be made to support, several properties. When a name is resolved it is sent to a name server which resolves it based upon the names it knows. Since differing tasks can have differing name servers, the implementation directly supports relative names, since the closures are different, i.e. it guarantees that $closure(u,n) \neq closure(u',n')$ can occur.

The name space supported is homogeneous. Each component namer removes a token before passing on the remainder of the name for further resolution. This is equivalent to either cases (i) or (ii) in the definition of homogeneity; in the current implementation it matches with case (i). By supporting homogeneity the process of name resolution will terminate. This approach also allows the expansion of the name space in a uniform manner: it can expand by the addition of more leaf namers which support more names, and these name can then be accessed using existing mechanisms and syntax, with different or longer names. This expansion does not affect any of the already existing names.

The Namer supports synonyms: an object requiring a synonym simply requests the appropriate namer to recognise it with the given name, hence the object is referred to by two different names which is the definition for synonyms to exist. The implication of this is that it cannot support unique names; the reason is that when names are added there is no attempt made by the namers to check that the object referred does not already have a name. The consequence of allowing synonyms is that the resulting name network forms an arbitrary graph, rather than a network.

The namer tasks can be said to implement a source-routing scheme since they translate the high order names into channels. However, this differs greatly from source routing as provided by the UUCP scheme in which each component name maps on to a physical connection. By comparison, the component names in the Namer are mapped to channels that represent logical, and not physical, connections and so may span zero or more physical boundaries.

Since the namer supports relative names the only way to support absolute names is to make *closure* return a fixed object, often called the root of a name space, when absolute names are needed. This does, however, not provide a real absolute name space as that would require all names to be absolute (the $\forall$ in the definition of absolute names dictates this). Since the philosophy of *Wisdom* requires that all the connections that a task has to the outside world come from its parent, this means that the connection to the root, and hence the ability to support absolute naming, rely on the parent passing the correct connections. This gives parents greater control over the services that their children may access, e.g. by passing to the child a root which supports only limited services (this is equivalent to the UNIX `chroot()` system call).

The naming mechanism supports location independence since it dynamically maps names on to channels. The channels represent the location of a task or service (they are mapped by Guardians on to the network address of the task), but since they are only generated by the process of name resolution, the location of an object is irrelevant until that point. The names themselves do not represent the location of the objects, despite the fact that the names space is said to be source-routing. The name specifies a sequence of objects which should be contacted to find the location of the object, rather than directly specify the object itself. However, once the name has been resolved the location of the object has been determined, so the result of name resolution is the conversion of a location independent name into a name which specifies a fixed location.

The only way in which an object may be accessed is via the Namer – there is no equivalent of a fixed network address which can be used in a program to avoid the cost of name lookup. This results in a potential overhead, since all services must be found by the Namer before they are used, but this could be overcome by implementing a caching scheme (including the addresses in the program can be viewed as an extreme form of caching). The advantage of forcing all object accesses to go, at least initially, through the Namer, is that it gives greater flexibility of the system. To quote Needham again[64]:

> The Cambridge File Servers had ring addresses 1B and 6C from their inception and retained these until they were abandoned. It is extremely likely that many programs which made use of them did not include reconsultation of the name server in their outermost retry loop, which strictly they should have done; if someone had changed these numbers at three o'clock one morning, and updated the name server accordingly, it may confidently be expected that parts of the distributed world would have been found not to be working by nine.

In a system which is potentially subject to frequent reconfiguration, which is the case with the scalability of *Wisdom*, abolishing addresses results in a system that will not suffer failures because components have been moved.

4.4 Summary

The Namer described above can be said to have either achieved its goals, or have easy solutions to them which have already been described:

- *Location Transparency*: By providing a dynamic binding between abstract names, which do not contain any location information, and objects, which may reside anywhere in the system, the Namer supports location transparency of its services.

- *Homogeneity*: By the definition given in the theory the Namer supports homogeneity, with the simple syntax defined by a sequency of ASCII strings separated by, currently, the character `/`.

- *Extensibility*: The Namer is extensible since it allows more namers to be added, thereby increasing the network. Since namers are treated as ordinary tasks, and vice-versa, namers can be added anywhere that tasks can, i.e. to any namer, and tasks can be used to provide naming services, such as access to a remote machine's file systems.

- *Distributed and Parallel*: As the Namer is comprised of many component namers it is obviously parallel, and since there is no special centre to the namer (the root namer, if it exists, is treated in the same way as any other namer) it is distributed.

- *Scalability*: This is the main area in which the namer, under excessive load, may fail to meet its goals. Firstly a centralised root, needed to support absolute namers, represents a bottleneck. There are several potential solutions to this – caching and replication. However, these have not been investigated in *Wisdom* due to the comparatively small size of the prototype system. Secondly the namer relies on its Guardian to act as the interface between itself, the Router and the rest of the system. However, the guardian provides only a limited number of IMC connections. This could be overcome in several simple ways: the use of capabilities for IMC mechanism, or allowing the Namer direct access to the Router.

The design of the Namer has avoided the need to use broadcast or multicast messages, and hence overcome that particular scaling problem. When combined with the characteristics of message delivery, as shown in the previous chapter (chapter 3), it allows messages to be sent between tasks without necessarily interfering with other messages in the system, i.e. many name resolutions may proceed in parallel, without interfering with one another, or with other computations that may be being performed.

The Namer ideally needs many connections to other parts of the system, and this is best dealt with by giving it direct control over its IMC connections. The simplest and most attractive mechanism for this is the use of capabilities since they allow the Namer to give the same connection to many tasks, or to give out restricted accesses, as well as giving the Namer control over the number of IMC connections it is willing to support. In retrospect capabilities should have been used by the Guardian for the greater flexibility they would provide to both the Namer, and to tasks in general.

In conclusion, the Namer achieves most of its objectives and of those that have not been achieved most can be overcome by the methods described above. The failure to support IMC using a mechanism akin to capabilities was a mistake, and one that would limit the scalability of a system, and one that will have to be rectified before fuller studies can be made of scaling the Namer.

Chapter 5

Load Balancing

We that have done and thought,
That have thought and done,
Must ramble, and thin out
Like milk spilt on a stone

Spilt Milk by W.B. Yeats

The term ‘load balancing’ should be considered something of a misnomer, since the objective of a load balancing policy should really be the optimisation of some performance criteria – these may include such as CPU utilisation, throughput, turnaround time, waiting time and response time.[†] Performance optimisation may *not* entail an evenly balanced load among all the processors, but rather one in which other factors, such as interprocess communication, are taken into account – thus it may result in possible clustering of some tasks on one processor. Unfortunately, little or no information may be available about these factors and even if they are known the cost of collecting and employing them in the decision making process may impose considerable overheads, as well as greatly complicating this process. For these reasons load balancing, the act of sharing tasks evenly between processors, can be regarded as a basis for optimising the performance metric. Hence the aim of the load balancing described here is to maintain approximately the same load on all the processors. This results in the ready queues of the processors being kept a short as possible, which is reasonably close to minimising the waiting time metric (assuming that the tasks all take an equal amount of processing time).

In *Wisdom*, the programmer creates many tasks, without being concerned with the underlying hardware (and its parallelism). It is the objective of the Load Balancer, therefore, to share out the tasks between the available processors so as to make maximal (or near maximal) use of these processors. This objective can be broken down into various goals, and they are listed in section 5.1.

It is assumed that there will be, effectively, no knowledge of the interconnection requirements of the various tasks. This assumption may appear invalid since there are points at which the interconnections are known, for example in a command line which contains many programs in a pipeline, or from within an application program which creates a farm of processes. However, the information present here is often insufficient – no details are known of other connections that might be made (e.g. to servers), or of the traffic over the connections. Collections of parallel modules might be built up by dynamic connections to servers, in which case no useful information may be present.

[†] These come from Peterson and Silberschatz[67]

Even if all the information necessary is available, or a suitable subset of it, making use of it requires mapping the topology of the program on to the topology of the network, and this has been shown to be computationally very expensive, specifically finding an optimal mapping is NP-complete[40]. Although it is possible to use heuristics to produce mappings, they too are complex, and will only achieve near optimal solutions. The methods described below also produce sub-optimal solutions, but the mechanisms they use are simpler and cheaper, and as the results given later show they do produce very acceptable solutions.

Much work has been done on load balancing and related areas. The focus of each study has varied from attempts to obtain near optimal performance from distributed systems to using load balancing as a method of achieving fault tolerance and deadline scheduling[74]. There is some similarity between certain areas of load balancing and queueing theory, specifically multiple queues and multiple servers (the M/M/m servers). The classic text on this subject is Kleinrock's Queuing Systems[47]. Of the work on load balancing itself, the paper by Wang and Morris[89] provides a very detailed comparison of different load balancing methods together with evaluations of their relative advantages. Ferrari[38] considers more complex load indices for load balancing, which are applicable when the cost of migrating a task is high, and Eager et al.[34] contrast the two main types of load balancing: sender-initiated and receiver-initiated, by means of a detailed study of each.

5.1 Goals of Load Balancing

The goals of a load balancing policy are listed below in order of importance. There is no point in maintaining an even load if it means that the processors can perform no useful work.

- *low processing overhead*: Performing load balancing (collecting all relevant information, making decisions and moving tasks) should be performed as quickly and as cheaply as possible. Failure to keep these overheads low could result in the system being unable to perform useful work, or performing less work than it might otherwise be able to.

- *low message overhead*: Load balancing should not take up more of the system bandwidth than is necessary as this would starve existing processes of the communication resource, and prevent the system form performing useful work.

- *low remote communications*: When cooperating tasks are placed on differing processors by the Load Balancer, they should be kept reasonably close together so that the cost of communication between them remains low. In practice, this means balancing to adjacent processors.

- *High processor utilisation*: Processors should not be allowed to idle, for lengthy periods. If this does occur then, clearly, processing power is being wasted.

- *Even load distribution*: All processors should have roughly the same amount of 'work' to perform. This is a stricter form of the previous condition which helps ensure that the previous condition is kept by dividing up 'spare' work between the other processors to ensure that when one processor finishes its task, it has more work to execute.

The first two goals are standard operating system ones, that is, to minimise the overheads of executing the system. These are considered more important than the remaining three since, even if these three goals are achieved, the cost of balancing may become so high that the system is unable to perform useful work. Keeping the processing overheads low is considered more important than keeping the messaging costs low since the processor can still perform useful work whilst awaiting (or sending) messages. This is not true, however, if it is busy deciding on where to balance the load. Keeping the remote communication low is considered the next most significant goal since large amounts of remote communication between tasks will not only slow the balanced tasks down, but also remove communication bandwidth from other tasks, potentially slowing them down as well.[†] The next goal is that of keeping all the processors busy, which, placed as it is, comes to mean, keeping as many processors busy as is feasible under the restraints imposed by the higher order goals. The reason for this being less important than minimising remote communication is that, with the relatively high costs of communication, spreading the tasks over all the processors might result in a far slower system than perhaps would have been the case had some been clustered on a single, or small number of, processors. The final goal is to keep the load as even as possible – as is implied by the term load balancing – and the low rating of this goal justifies the reasons for considering load balancing as a partial misnomer for this module.

The ordering of the goals represents a weighting of importance, from high to low. The aim of a load balancing policy should be to achieve each goal with a minimal effect on more important goals. It is possible to achieve the first three by performing no load balancing at all, yet this would result in the last two being never met.

5.2 Components of Load Balancing

Any dynamic load balancing policy can be considered to comprise of two components: the metric and the site selection. The metric is a measure of the load on a given processor, and this component includes the methods for its collection from other processors with which balancing might take place. The site selection is the comparison of all the load metrics and the decision on which processor to run a task.

5.2.1 Load Metric

The load metric is the measure of the load on a given processor. This metric can vary greatly from random numbers to complex calculations involving the processor load, available memory, spare input/output bandwidth and previous loads. There have been many surveys and comparisons of the various metrics, the most important of which have already been mentioned. In systems with heterogeneous processors, the metric has also included the relative speeds of the differing processors (assuming that these are object code compatible).

The load metrics are collected in one of two ways: either there is a continual flow of these values, or they are requested when needed. These two methods are referred to as bidding and update since, in the former, the information flow represents a bid for a task, and in the latter it represents an update to the load metric. The choice between them depends on the system and the facilities provided. For example, in a system which continually sends messages between processors (e.g. to say that they are still running), the load can be piggy-backed on them at little or no extra cost. Equally requesting bids from all machines on a large network could quickly swamp the source processor, and cause the network to become overloaded by other processors attempting to return bids.

The metrics must also be kept up to date, which can be a problem if they are only updated at regular intervals. The reason is that if a node announces a low load, then many tasks may arrive for it to execute before the system is informed of the change in its load, thus resulting in the reception of too many tasks, and the system consequently becoming unbalanced. In order to overcome this, very frequent updates of the load are required, but this, in turn, places a high load on the communication bandwidth, which is undesirable and may also introduce conspired thrashing (repeated migration – see section 5.4.2). This is the primary reason that load balancing techniques do not, generally, use methods involving a continual flow of updates, and why those that do are not very efficient and require very accurate load metrics which also try to predict future loads[94].

[†] Certain amounts of remote communication are acceptable if the system has been designed to allow communication between tasks with minimal impact upon other tasks. Details of this are to be found in chapter 3.

Even if the load metric is transmitted only when requested, this can result in a high load on the communication system just when it is least wanted: when there are many tasks in the system, and potentially much communication. To overcome this systems have been developed where the load metric has only three possible values: low, medium or high[65]. When the metric has the value low, the node is willing to receive tasks; when the metric is medium, the node neither wishes to receive nor offload tasks; and, when high, the node wishes to offload tasks. In these systems each node transmits a message *only* to say that its load metric has changed state. The processors then remember the states of the other processors and use this to determine whether or not to balance. A disadvantage of this method is that, when all processors are in the 'high' state, they will not share tasks, which may result in greatly varying waiting times (the time which a runnable task spends waiting for the processor) caused by the differences in numbers of tasks on the processors in the 'high' state.

5.2.2 Site Selection

Site selection is the mechanism that choses a processor to run the task being balanced. The policies are generally simple, and vary depending on the facilities supported. In bidding systems, it is common to use the first reply which has a lower load than the current node if the communication delay is uncertain or unreliable. If, however, the load is available quickly (either because the node has its own copy or is able to obtain the load metrics in a short, reliable, time) the usual method is to select the best processor, i.e. the one with the lowest load metric.

If all the processors are on a broadcast bus (or share memory) then the cost of transferring the task does not vary depending on which two processors are being balanced. If, however, the processors are connected by point to point links, then the load on the links and the relative speed of the links might well play a part in the decision of the destination of the balanced task (though these factors may also be included in the load metric). Also of concern here may be processors that are 'far' away (e.g. requiring many hops to reach), and whether it is worth balancing tasks to them, and the effect this will have on IMC.[†]

Consequently, although processors of a lower load may exist in the network, the cost of transferring tasks to these processors may be unacceptable, not only due to the time it takes to transfer a task, but also in the load placed on the communication system in transferring the (potentially large) task. This problem can be overcome by the introduction of *watermarks*. These are the numbers by which the loads of the relevant processors must differ before the site selection policy considers it worthwhile moving tasks. Generally these numbers are constants, tuned to the speed of the communication network, but they could equally well be variable, depending on the network load and the cost of transferring a particular task (i.e. it varies with the size of the task), and the speeds of processors or communication links.

5.3 Types of Load Balancing

The ways in which tasks are transferred is often used to classify load balancing schemes. Two broad classes can be identified: sender-initiated and receiver-initiated. These terms refer to which processor causes the transfer of a task: in sender-initiated, it is the processor where the task currently resides; in receiver-initiated, it is the processor which is to execute the task. The work by Eager, Lazowska and Zahorjan[34] contrasts the two, using simulations, and can be consulted if a detailed comparison is of interest. In a similar manner the way in which the load metrics are collected can be used to differentiate between differing load balancing policies: in this instance the classes are bidding and update.

[†] The extent to which these points are considered will depend upon the form and characteristics of the communications subnet. If it can support communications over long distance at much the same cost as nearby (or local) communications, then these two concerns will be of little importance.

Load balancing itself can be performed in one of two ways: dynamic or static. In a dynamic load balancing scheme the processor decides where to place tasks depending on current loads (i.e. it uses load metrics measuring the current state of nodes). In contrast, static load balancing uses pre-determined methods to place the tasks on processors.

Examples of static load balancing policies are to place all tasks created on processor A on processor B (the so called deterministic policies), or place half the tasks on B, a quarter on C and a quarter on D (the so-called probabilistic methods). These policies, the basis of early load balancing systems, are very simple, since there is no need to collect or maintain information. Static load balancing works well only when the demands on the system are known in advance, such as may be the case in certain control environments. In a general purpose environment this information is not necessarily available, therefore the dynamic policies can perform far better, and for relatively little increase in complexity. *Wisdom* uses dynamic load balancing for this reason; static load balancing will not be considered further in this thesis.

5.3.1 Receiver Initiated

In a receiver initiated policy, any node with little, or no, work to perform actively ‘looks’ for work by requesting it from other processors. The disadvantage of this is that a network of lightly loaded, or unloaded, processors will have a high overhead due to the fact that all the processors will be asking one another for work. However, once the average load becomes high the overheads generated by this mechanism drop. Hence, this type of load balancing is best suited either to a system which is under heavy load or to one which supports very cheap communication.

5.3.2 Sender Initiated

In a sender initiated policy, the node which has the task, or a request for a task, decides whether or not the task should be transferred. The advantage of a sender initiated policy is that the load on the system (to support the necessary communication) is only present if there are tasks to run. This is also, unfortunately, a disadvantage as at the moment when there are many tasks runnable, there is the greatest overhead. This type of load balancing, therefore, can only be thought to be an advantage where the average load of the processors is low, or where the cost of communication is low compared to that of transferring and creating tasks.

In the architecture under investigation, it is expected that there will be many processors and that the average load will be low (each processor having a small number of tasks). Given these characteristics a sender initiated load balancing policy should perform better that a receiver initiated policy. For this reason, the discussion in the remainder of this chapter will assumed sender initiated policies.

5.3.3 Bidding and Update

As has been already mentioned bidding and update represent two differing ways of collecting load metrics. In bidding policies the bids that a processor makes represent its load. The format the bid takes depends on whether or not the load balancing is sender or receiver initiated. In a sender initiated policy the bids are made in reply to requests, otherwise they represent unsolicited attempts to locate work. Update policies are ones in which the load is transmitted when it changes; and the terminology is also used to cover policies in which there is a continual flow of information (in these the values are updated based on time changes rather than metric changes). Mention of their relative merits has already been made, in section 5.2.1.

5.4 Dangers of Load Balancing

Several factors must be borne in mind when designing or implementing a load balancing policy, some of which can cause a worse performance metric for a limited number of tasks, i.e. they can become unfair. One common omission is failing to take into account the cost of transferring a task, or assuming it to be a constant. Also, the action of transferring a task to another node could load both processors, thereby destroying the benefit of moving the task. This cost, that of moving the task, is sometimes included in the balancing algorithm or the calculation of the load metric so as to minimise this when possible, specifically via the watermark value.

5.4.1 Unbalanced Nodes

One of the more ironic problems that can occur with load balancing policies is that they may generate worse performance metrics by balancing. This may be caused by the overhead in migrating a task or in deciding where to place it, or if the processors become badly unbalanced. The processors can become unbalanced by the following means:

- Outdated Information
- Mobbing
- Single Migration

The first (outdated information) can result in tasks being transferred to a highly loaded processor as other processors are not yet aware of this fact. This problem can be overcome by policies which request the load metric only when needed, or by update policies which correct their information only when their states change. Increasing the rate of information in update policies (especially those based on time changes) tends to make matters worse. There are two main reasons for this; firstly, the processors have a greater overhead in trying to maintain this increased level of communication. Secondly, the processors are more likely to transfer tasks with the high communication rate, consequently sudden, temporary changes of a single processor's load can spread through the system causing excessive task transfers, rather than being smoothed by the passage of time.

Mobbing, a processor being swamped by work, can occur when a node announces a low (or high) load and many other processors suddenly overload (or underload) it. Bidding policies can partially solve this since they are, largely, a one-to-one form of communication.

It is conceivable that, with single migration, a node could readily end up with several long running processes, leaving other processors with short running processes, causing a load imbalance after a short period of time. The simplest solution to this is to allow the processes to migrate many times: unfortunately this encourages conspired thrashing.

5.4.2 Conspired Thrashing

Conspired thrashing (sometimes referred to as *woggling*) is akin to the thrashing which can occur in conventional mono-processor systems (it is the situation in which the system spends all its time deciding which task to run, and preparing it to run, but without ever actually running any tasks). In a system with load balancing, conspired thrashing is where tasks are moved between the processors without any useful work being performed. This can manifest itself in two forms: firstly, the entire system may perform no work since it spends its time moving tasks, or secondly one, or a few, tasks never get the chance to execute since they are forever being moved between processors, i.e. the suffer from starvation.

There are two ways to overcome this problem. Firstly all task can be required to execute for a short time before being considered for balancing. This automatically guarantees that tasks will spend some time executing, but if the cost of balancing, and the time it takes, is high, then the task will suffer from high waiting times. Alternatively tasks may be moved only once, which guarantees they will run, and reduces the waiting time if the load is low on the destination processor.

5.4.3 Communication Overheads

The cost of preparing the task for transmission, the transmission time, and the cost in decoding the transmitted process into a runnable form may be so high as not only to offset the migration advantage but also to place a high load on the processors concerned. These costs must be incurred in multiple migration – creation time migration avoids this, or more accurately reduces their occurrences. Unfortunately, as has already been mentioned, single, creation-time migration is not always acceptable.

There are many examples of these overheads; Zatti[94] uses a collection of UNIX 4.2 BSD systems linked via ethernet running TCP/IP, in which the migration mechanisms have overheads varying from 4.9 seconds to 16 seconds. A setup of two PERQ 2's running Unix United over ethernet[1] gave overheads of 7 seconds for cpu bound processes and varying large overheads, of a similar order, for other types of tasks. A faster communication system should dramatically reduce these overheads and greatly improve performance.

The communication overheads derive not only from the cost of moving the task but also from any communications the task may make. The cost of moving the task can be roughly calculated from its size, but the amount of communication a task will make is often unknown. Based on the idea that the task will have some knowledge of the amount of communication that it will make, Barak[10] has developed a system in which the processes themselves control whether or not they will migrate. Although there are times when certain tasks are definitely not to be migrated (such as those providing services on a node), requiring tasks to perform their own migration is not acceptable as these operations are then no longer transparent to the tasks. The best solution would be to allow a privileged request that would prevent a task from being migrated.

In Helios[51], a commercial parallel operating system for the transputer, load balancing is achieved via the Component Distribution Language[41] (CDL) and the Task Force Manager (TFM). The CDL is used to describe the interconnections of the components of the program, which the CDL compiler then generates code which is used by the TFM to map the components on to the available processors. The TFM has knowledge of the interconnections between the tasks and between the available processors and so can try to produce a near-optimal mapping of tasks to processors. Unfortunately, an optimal solution to this mapping is NP-complete, and it is, therefore, not feasible to determine optimal mappings, so heuristics are used instead. The TFM, therefore, effectively provides a static load balancing mechanism generating near optimal placement for a known network using the information provided in the CDL description of the program interconnections.

5.5 Simulated Load Balancing

The majority of the research in the area of load balancing has assumed, either explicitly or implicitly, the presence of broadcast connections or a complete network, i.e. all the tasks can be moved directly to any of the processors. This is not the case with the processor meshes that *Wisdom* uses. The remainder of the research has dealt with irregular networks which do not exhibit the characteristics of *Wisdom*. Therefore, to evaluate some of the above approaches when applied to a mesh of processors, some simple simulations of load balancing where undertaken. These simulations were used to compare several different methods of deciding whether or not to balance the load, and where to place the tasks if balancing was undertaken. In total three different load metrics, three different methods of comparing the loads to decide if load balancing was necessary (watermarks), and two different site selection policies were simulated in various combinations.

The model used was a 2-D array, each array element representing a processing node, and the value in the array at that point being the load of the corresponding processing node. The simulation repeatedly passes through the array. At each point in the array it stops and performs the balancing mechanisms under study. This is done by acquiring the loads of the neighbouring processors, using one of the three calculation methods, comparing them using one of the three comparisons and if it decides to balance the load it does it via one of the two site selection algorithms. Each pass over the array can be thought of as a compute cycle followed by an attempt to balance the load.

Obviously this model is very naive: there is no concept of global time and global synchronisation in a real system. It assumes that all the communication channels are free whenever needed, and that there is an immediate response to requests. No consideration is given to either communications done by the processes themselves, nor to the overheads introduced by transferring the tasks, nor to any idle processes currently blocked. In practice these are not unfair assumptions to make – given the high speed routing hardware suggested in chapter 3, communications become nearly free, very fast and unlikely to be held up waiting for other communications. The assumption of a global time is used simply to generate a point at which the load balancing may take place: whether or not it is used affects only the complexity of the model.

Several assumptions were made about the design of the model; alternatively they may be considered as axioms that were taken, and not to be placed under study in the model.

- A node will always ensure, if at all possible, that it has a task to execute. That is to say that if it knows it has three incoming tasks, one of its own and a neighbour with none it will still not migrate its task.
- Communication is only possible between immediate neighbours. To travel farther afield the communication must be forwarded; thus, it is not possible to balance tasks to other than its immediate neighbours.
- It is only possible to transfer one task over one link at a time, and this is always completed in one time slice.
- No attempts at simulating fairness were made.
- The models only consider sender-initiated task transfers.
- No consideration is given to memory requirements, communication matters or background loads (such as that created by devices).

Three statistics were collected: load imbalance, task transfers and system utilisation. Load imbalance is defined as:

$$\sum_{p_i} \left(\frac{\left| \left\lfloor L(p_i) - J/N \right\rfloor \right|}{2} \right) \quad \textit{where } L(p_i) = \textit{the number of tasks at } p_i$$

J = total number of tasks in the system
N = total number of processors in the system

It is effectively a count of the processes that are not correctly balanced: i.e., the number of processes that should be moved to obtain an optimal balancing. Task transfers are merely a count of the tasks in transit; a measure which is of importance since, the higher it is, the less communication bandwidth remains to be used, and the more time is lost in the movement of tasks (in practice is is representative of the load balancing overheads). System utilisation is a count of the processors performing useful work: it is highly desirable that this is maintained close to one hundred percent for as much of the time as is possible.

5.5.1 Results of Simulations

The three policies under study, which together make up the load balancing policy, are the load calculation, the load comparison and the site selection policies. The various forms of each that were studied are as follows.

Load Calculation

1) Current count of tasks
2) Current count of tasks, modified by any incoming or outgoing tasks
3) Current count of tasks, modified by any incoming or outgoing tasks, and averaged over the last n timeslices.

Load Comparison (Watermarks)

1) $>_1$, i.e. transfer if the interrogated processor has more than one task fewer than the source processor.
2) $>$, i.e. transfer if the interrogated processor has fewer tasks than the source processor.
3) $\geq$, i.e transfer if the interrogated processor has the same number or fewer tasks than the source processor.†

Site Selection

1) Round Robin. Interrogate the processor in strict order. As soon as a suitable processor is found transfer to it, regardless of whether or not more optimal processor might exist.
2) Optimal Site Selection. Interrogate all the neighbours, select the one with the lowest load, and transfer to it if acceptable. Repeat this until either all the neighbours are receiving tasks, or further transfers are unacceptable.

The scenario being modelled is one in which there is an initial impulse load, which proceeds to create more tasks for a time, these tasks spend time ‘‘computing’’ and then finally start to die.

Tables 5.1 to 5.3 contain the results of 17 simulation runs, each of which contained a different combination of the load calculation, load comparison and site selection policies described above. Every run returned three results – the load imbalance, the number of tasks transferred and a measure of the idle time of the system. Certain possible combinations of the methods were not run since the other results indicated that they would be unacceptably expensive. The various values that were obtained for each of the measures have been normalised by dividing all values by the smallest, so that the smaller the number the better the performance, with the value one being the best obtained. This enables easy comparison of the various metrics.

An interesting fact to emerge from simulations that is not shown in the results presented is that tasks could become ‘trapped’ at highly loaded processors. The picture (figure 5.4) below illustrates the point: the numbers represent the load at the given processor, and it is assumed a watermark of 2 is being used. The tasks on the heavily loaded node (with the load 4) are unable to escape to the unloaded processors (those labeled 0). They cannot move because of the watermark, but, even without one, a similar effect will occur if conspired thrashing is to be prevented. To overcome this problem of trapped tasks some groups have used a load metric that includes some component of that on adjacent processors. This mechanism will work as with the correct combination of the loads it is possible to reduce the load metric for a processor surrounded by unloaded processors so that tasks may move (e.g. basing the load metric on the current task count added to the average of the neighbouring processors task count). Fortunately, for tasks to become trapped in this manner the system, and its tasks, must be static and not one in which tasks are continually being created and destroyed, as is the case with *Wisdom*.

---

† This rather strange comparison was included in the simulations as it was thought that it might overcome boundaries past which the other policies would refuse to transfer tasks. This is returned to later.

Site Selection Policy 1			
Load Calculation Method	Load Comparison Method		
	1	2	3
1	1.62	5.85	7.16
2	1.14	4.89	10.21

Site Selection Policy 2				
Load Calculation Method		Load Comparison Method		
		1	2	3
2		1.06	3.92	7.66
3	3 (integer)	1.00	2.73	-
	5 (integer)	1.14	2.76	-
	3 (floating)	1.52	4.41	-
	5 (floating)	2.18	4.97	-

Figure 5.1: Task Transfers

Site Selection Policy 1			
Load Calculation Method	Load Comparison Method		
	1	2	3
1	1.75	3.93	4.63
2	1.48	1.36	1.98

Site Selection Policy 2				
Load Calculation Method		Load Comparison Method		
		1	2	3
2		1.46	1.00	1.11
3	3 (integer)	2.52	1.62	-
	5 (integer)	2.23	1.62	-
	3 (floating)	1.89	1.72	-
	5 (floating)	3.14	3.89	-

Figure 5.2: Load Imbalance results

5.5.2 Conclusions from the Simulation Runs

The conclusions are easy to draw, and are mostly obvious and intuitive, even without the simulations.

- Loads should take into account arriving and departing work.
- Optimal site selection should be used
- Averaging the load over time is not a beneficial approach
- The watermark should be adjusted to bring about the highest transfer rate that the system will support, subject to a minimum value of one.

Site Selection Policy 1			
Load Calculation Method	Load Comparison Method		
	1	2	3
1	1.00	1.05	1.12
2	1.00	1.00	1.11

Site Selection Policy 2				
Load Calculation Method		Load Comparison Method		
		1	2	3
2		1.06	1.02	1.02
3	3 (integer)	1.40	1.09	-
	5 (integer)	1.72	1.34	-
	3 (floating)	1.24	1.05	-
	5 (floating)	1.40	1.09	-

Figure 5.3: System Idle

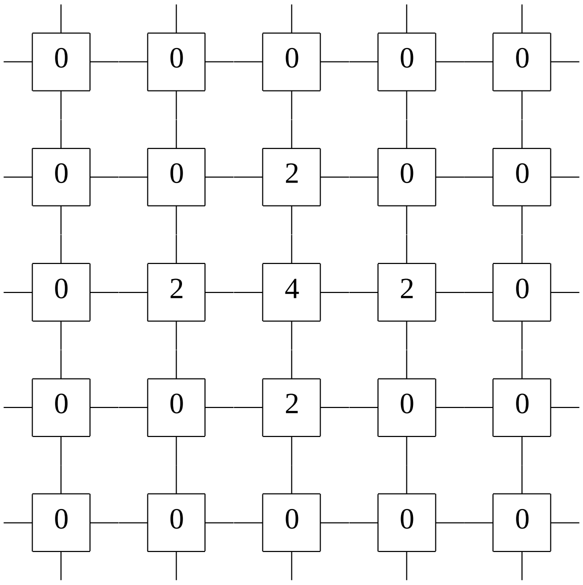


Figure 5.4: Trapped Tasks caused by Watermarks

The first three of points are obvious from the tables, and represent a choice of the second site selection policy, and the second load calculation method. Consulting the relevant lines in the tables shows that the higher watermark (load comparison method one) has a far lower task transfer rate, and although it does increase the load imbalance, makes very little difference on the system idle metric. These are the reasons for the final point.

5.6 Wisdom and Load Balancing

The various characteristics of the Load Balancer for *Wisdom* were determined by the results on the simulations and of the experience described in the many papers on load balancing mentioned earlier in this chapter, and in chapter 1. Also affecting these choices were the restrictions placed upon the system by the available hardware. The following are the chosen characteristics of the Load Balancer:

- Task counts, adjusted by arriving and departing tasks, are used as the load metric.
- Optimal site selection is used.
- A watermark of one is used.
- Only creation-time load balancing occurs.

The load metric was chosen to be a count of the tasks present on any given node, including tasks that were arriving, but not those that were departing. This had been shown to be perfectly adequate by the simulations and agrees with work performed by other groups. This count consists of all the tasks present at a node, but does not remove those that are currently idle. This is since at some point these idle tasks will start running; however, running tasks cannot be moved.

The site selection selection policy uses comparisons with a watermark value of one, in order to reduce the number of transfers to an acceptable limit. Optimal site selection is used because it is known that the load metrics will be returned almost immediately as the communication connections are reliable and fast.

The load balancing system built can be described as a bidding, sender-initiated policy with single migration. By using a bidding policy, problems associated with maintaining correct metrics are avoided; sender-initiated balancing is both easier to implement than receiver-initiated and with the relatively cheap communications it should perform as well as the alternatives. Single migration, although a hardware-imposed restriction,† avoids problems of conspired thrashing and guarantees that a task will, eventually, execute.

5.7 Results from Prototype Implementation

Once the Load Balancer had been constructed, with the properties described above, it was used to run several test programs of varying characteristics. The following figures (5.7 to 5.11) represent the execution time of an increasing number of tasks. Each graph shows the behaviour with only a single processor (the linear trace, marked with Δ) which can be viewed as the ‘control’ test, and the effect of increasing the number of processors which are used to execute the tasks. The execution times are shown for each task, or pair of tasks (averaged over five runs). The network used for these tests is shown in figure 5.5. All tasks were created on the node labeled 1, and in a test using n processors, all processors labelled $\leq n$ in figure 5.5 are potential destinations for the tasks: this was achieved by setting internal flags for the Load Balancer which disabled balancing to certain processors.

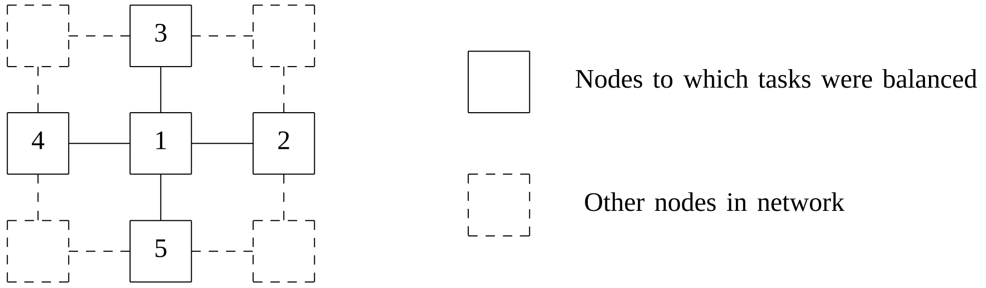


Figure 5.5: Network Used in Load Balancing Tests

---

† The transputer, the processor used, does not support memory management, and this makes the task of moving running task (specifically their stacks) effectively impossible.

Δ	1 processor in test	×	4 processors in test
□	2 processors in test	•	5 processors in test
+	3 processors in test		

Figure 5.6: Legend for Results Graphs 5.7 to 5.11

The first of these tests (figure 5.7) simulates pairs of tasks which spend their time exclusively communicating. This shows the expected advantage of running a single pair on a single processor, however it does also show that when the load becomes high (in this case when there are 4 or more pairs of tasks) the load balancing exhibits a notably improved performance over the single node version. The reason for these improvements is that each communication involves a small, but significant amount of computation by the Router to deliver the message, and when the load is high the computation produced by this benefits from the distribution to differing processors that the load balancing provides. In the region below this high load, the attempts to share the load do not, always, produce improved results since there are less communications, and hence less computations and so less load to balance. Also the actual communication time, the time to transfer the message between the processors, is still dominant. It should also be noted that the additional processors in the network (i.e. those drawn with dashed lines in figure 5.5) will, where the destinations are appropriate, be able to carry some of the communication load when three or more processors were in use, increasing the benefits of balancing by adding more processors.

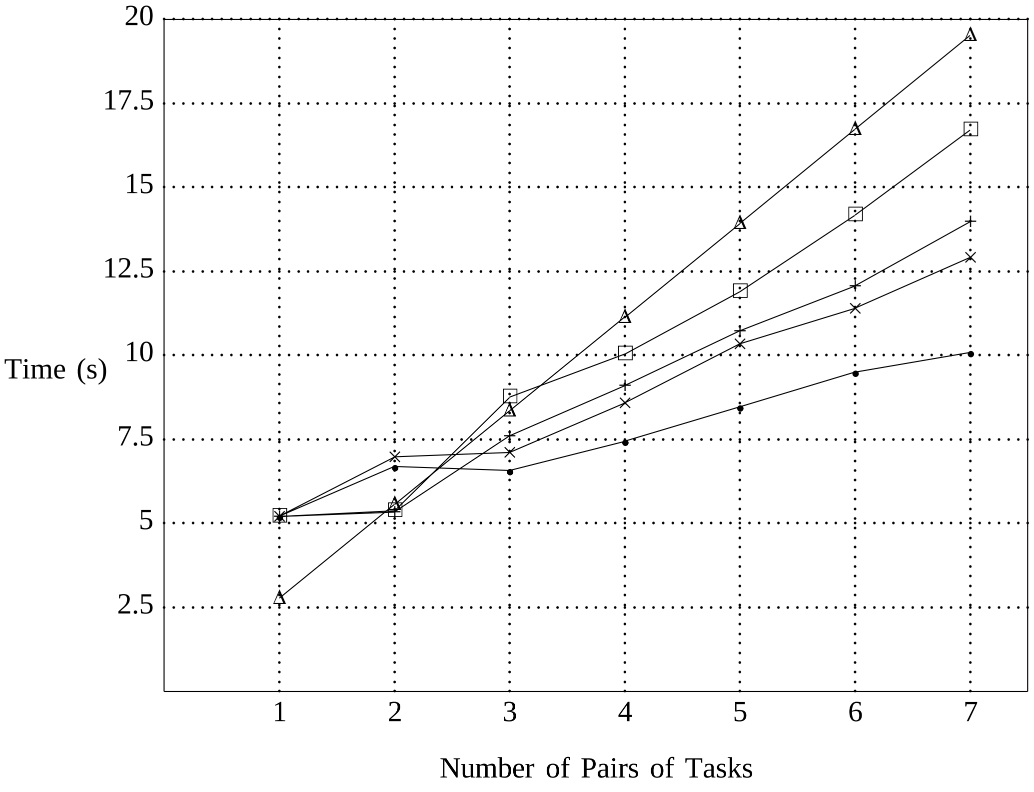


Figure 5.7: Communication Bound Tasks

The following test (figure 5.8) used a task that spent approximately 10% of its time computing, the remainder being spent in communication. The increase in computation time that this represents is slight, but gives a small, though noticeable, improvement in the behaviour of the Load Balancer: although the single task on a single node is still far faster, the differences with two tasks are small, and the performance improves from then on.

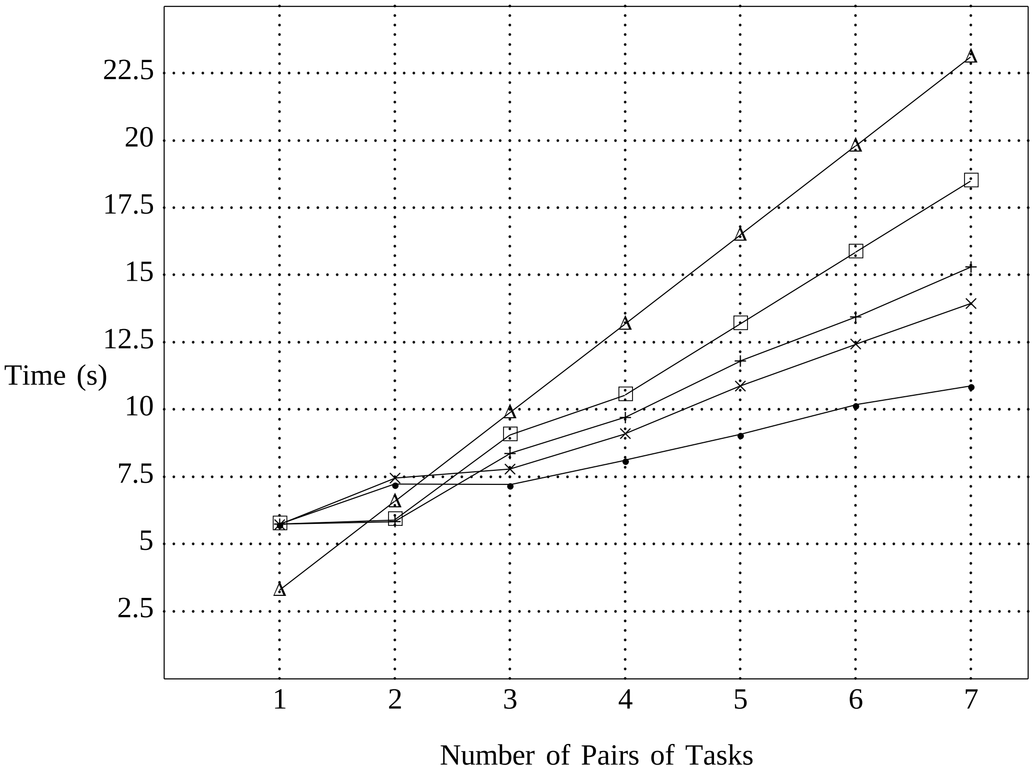


Figure 5.8: Mixed Tasks (10% compute, 90% communicate)

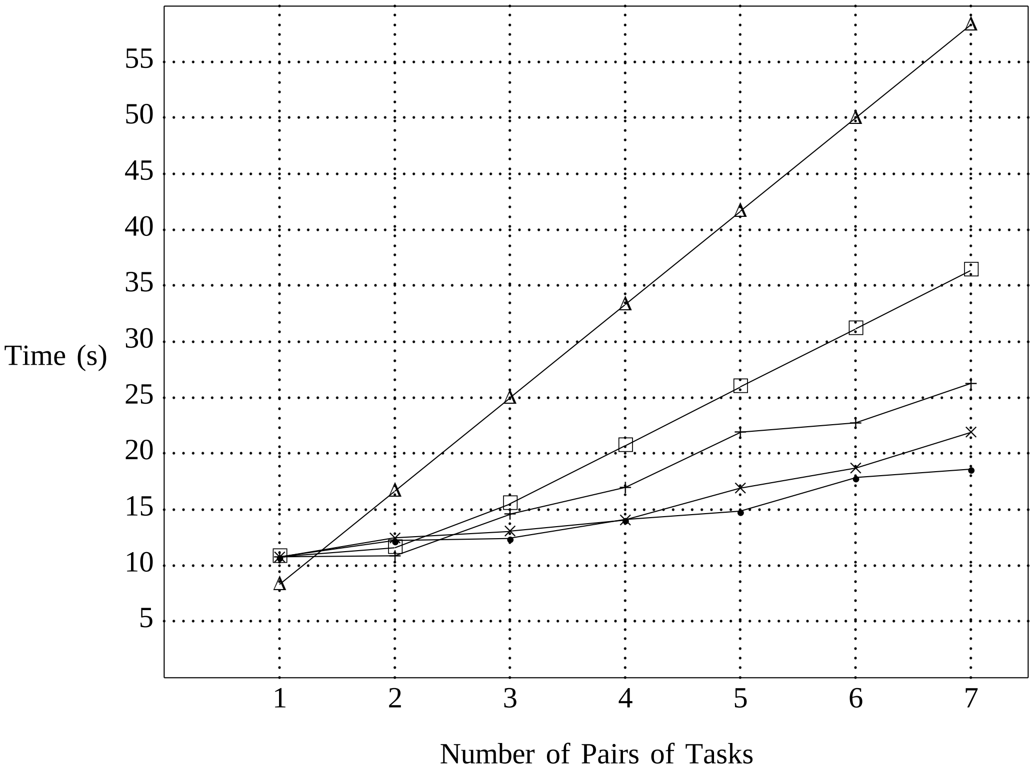


Figure 5.9: Mixed Tasks (50% compute, 50% communicate)

The third test (figure 5.9) used a task that spent half its time communicating, and half its time computing. The amount of computation now being performed is sufficient to ensure that load balancing shows a significant improvement, however there is still sufficient communication to prevent the graph from approaching an ‘ideal’ performance – that which would be generated by the optimal placement of tasks on the processors.

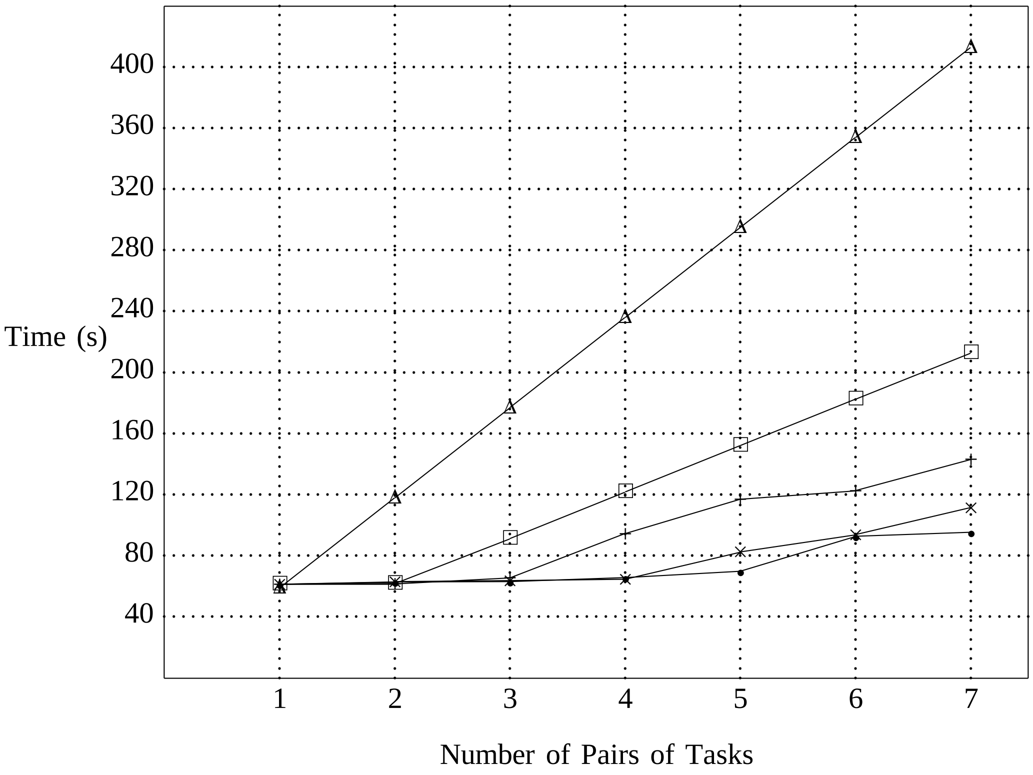


Figure 5.10: Mixed Tasks (90% compute, 10% communicate)

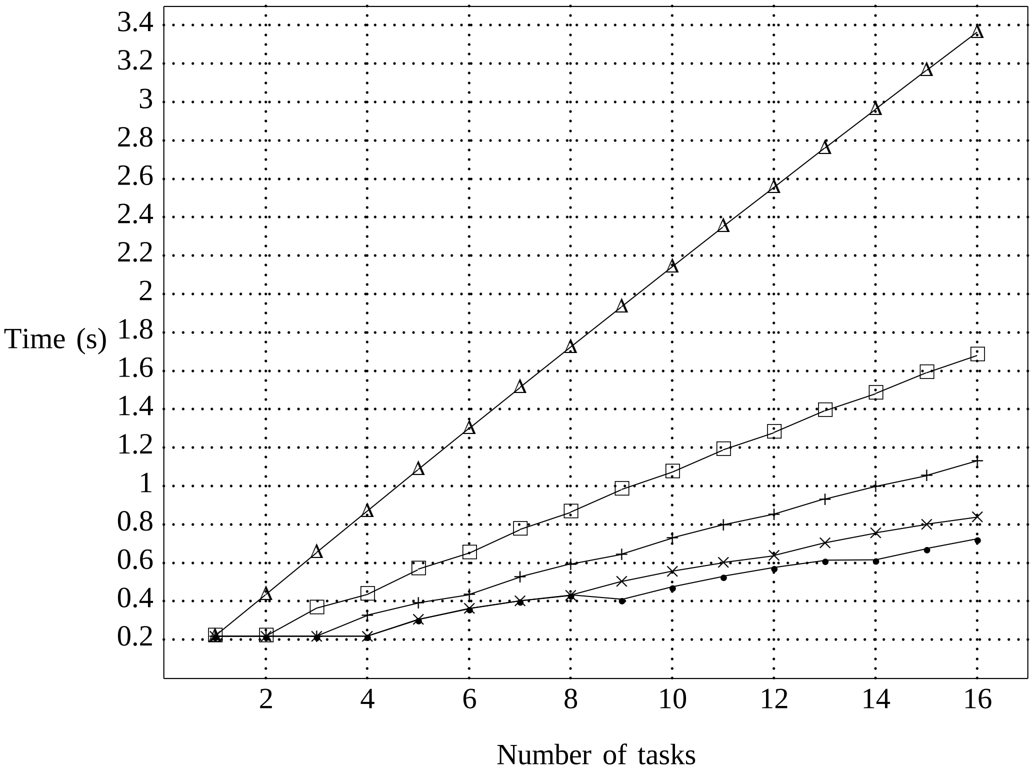
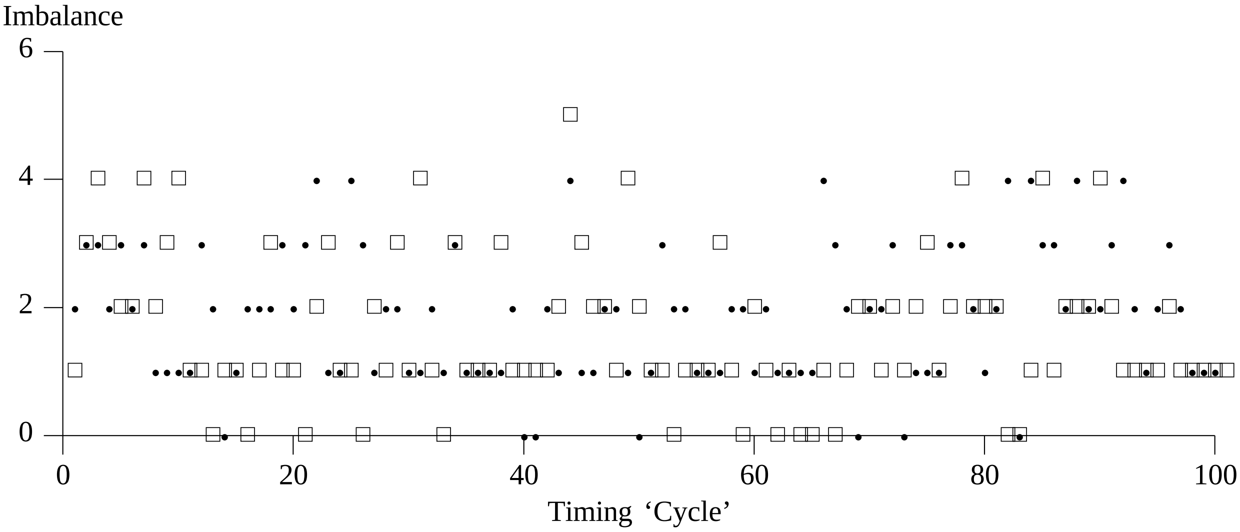


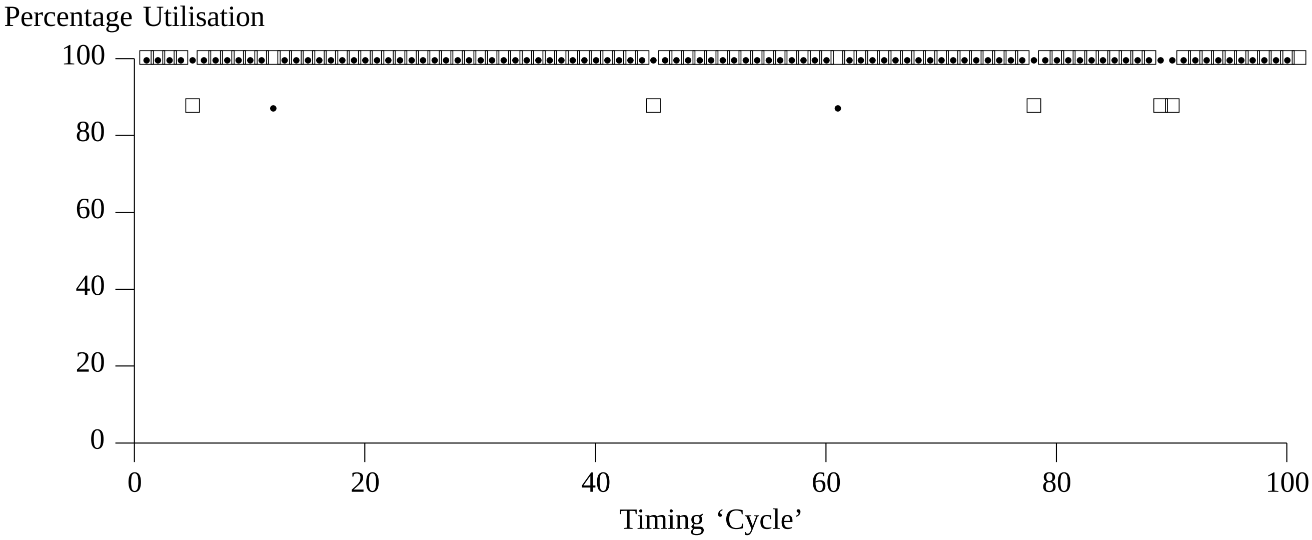
Figure 5.11: Compute Bound Tasks

The penultimate timing test (figure 5.10) used tasks which spent 90% of its time computing, and the remainder communicating. The computation now dominates to an extent that it is nearly ‘ideal’ performance.

The final timing test (figure 5.11) used tasks that only perform computation, and no communication. These give what is, effectively, the ‘ideal’ performance, given that the tasks are indivisible units.



(a) Load Imbalance



(b) System Utilisation

□ Simulated Load Balancing • Actual Load Balancing

Figure 5.12: Comparison of Simulation and Implementation

The final test used the two important metrics used in the simulation, the system utilisation and the load imbalance to study the performance the system. For this test a self-reproducing task, with the following characteristics, was used: random chances of dying, computing, communicating with a special parent, or creating a new task. These chances started out at 10%, 50%, 20% and 20% respectively when the task is created, and are then slowly altered until there is a 50% chance of dying and 50% chance of executing. Every task records its start and end times, and its location in the network. Limits were included to prevent more than 30 of these tasks existing in the network at any given time; this number was rapidly approached, and then maintained. These statistics are reported at its death to the parent and used to calculate the metrics that are shown in figure 5.12.

Also shown with these are the results obtained from a version of the simulator which had been modified to represent a three-by-three array of processors with the characteristics of Load Balancer, although communication was not simulated, and load balancing was assumed to be instantaneous. The results taken represent a ‘chunk’ of about three minutes execution of the system, divided in one hundred even divisions, and a sequence of 100 balancing ‘cycles’ of the simulator, after it had been running for 100 cycles. The system utilisation statistic shows near optimal results for virtually all the time, and compares well with the simulator. The load imbalance factor averages out at 1.84 for the system, and 1.61 for the simulator, and given the maintained maximum of 30 tasks, represents approximately 6% of tasks being misplaced.

5.8 Summary

In a system that hides the specific details of its parallel hardware, yet still expects the programmer to take advantage of this hardware, automatic load balancing must take place at some place. This can be from within a compiler which ‘knows’ the layout of the available hardware (such as vectorising compilers), through static load balancing policies to highly dynamic ones. The approach taken by *Wisdom* is to use a dynamic load balancing policy, but one which is built out of very simple components.

The Load Balancer makes no assumptions about communication patterns, nor does it try to determine these, though this is partially catered for by only creating tasks on neighbouring processors. The tasks are balanced, or moved, only when they are created, thereafter they remain at the same node. This avoids problems of tasks never executing, and instead always being moved, although it may introduce load imbalances where long running tasks are concerned. The processor on which to execute the task is chosen based on very simple metrics, rather than the more complex, more expensive, though more accurate measures that can be used. The system uses a sender-initiated bidding policy, one which only generates work and communication when there are tasks to be moved.

Various tests have been run using this Load Balancer and the results have been very favourable. The Balancer behaved well, and produced near ideal results for compute, and near-compute, bound tasks as well as giving very satisfactory results on communicate intensive tasks. The final test run aimed to represent a dynamic, communicating load, and the system was studied based on the metrics developed to evaluate the simulations. This produced near ideal results again, given that it is a sub-optimal approach (optimal load balancing being unacceptably expensive), which matched closely with those obtained from the simulator.

An important factor omitted from the numbers collected in the above results is that they were all generated from a single program, regardless of the number of processors that where present in that part of the test. Since the program was unaware of the hardware topology upon which it was running (due to the interconnection transparency supplied by the router and the parallelism transparency supplied by the load balancer) exactly the same binaries were used. This exhibits an important feature of transparency, and shows its success.

Chapter 6

Evaluation of Wisdom

> *Life isn't like that. Evolution has no long term goal. There is no long-distance target, no final perfection to serve as a criterion for selection, although human vanity cherishes the absurd notion that our species is the final goal of evolution. In real life, the criterion for selection is always short-term, either simple survival, or, more generally, reproductive success.*
>
> The Blind Watchmaker, by Richard Dawkins

Evaluating *Wisdom* – deciding whether or not the ideas and the design are valid and useful, and to what extent – is not a simple matter. The architecture that *Wisdom* makes use of is very different from those used in the majority of other operating systems, so a comparison with these other systems is very problematic. The reasons are that most other operating systems, save for a few designed around point to point networks, make use of a shared communications media, be it bus, ring or ethernet as shown in chapter 1. This means that these systems do not have the scalable and non-interfering communications characteristics of *Wisdom* which results in programs, generally, not being comprised of parallel components. The closest systems to *Wisdom* are those based around processor pools, such as the CDS and the Sequent machines. These still differ since they share a common communication media which limits the scalability (a ring in the case of the former and a shared bus in the latter), and in that they allocate a single processor to a user or program, rather than assigning a group of processors to a program or user.

To explain the problems of systems with a shared communication media remember that these have a fixed bandwidth. Adding more machines or processors to the network does not increase this bandwidth and so these additions must reduce any single machine's share. It may even reduce the overall bandwidth – these systems often send 'essential' messages between themselves which further depletes the bandwidth. If programs are written as many communicating components that are placed on as many machines as are available, then it is probable that it will not scale well because of the reduction in bandwidth each component will experience.

Wisdom has been designed with scalability as a primary objective, and with the view of it being a single, central service. By comparison, conventional systems are either not scalable, or are made of many, largely independent machines. Consequently the meaning of scaling differs between conventional operating systems and *Wisdom*. In *Wisdom* scalability means the ability to add more processors to speed up both the whole system and individual programs; whereas in the other systems it means the ability to add more machines. It is true that both conventional systems and *Wisdom* seek to support more users by scaling, however in *Wisdom* scaling will benefit existing users, or can be used solely to improve the performance of the system. To illustrate this difference consider a single program – the greatest speed at which it can run is determined, in a conventional system, by the speed of a single processor. However, in *Wisdom*, this single program can run faster as more processors are added to the system (if it has been correctly designed).

Although the architecture of *Wisdom* is not new, the uses to which it has been put are mostly variations on the theme of compute engines. In these, a single program, specially prepared to execute on the particular hardware, is loaded and run, but without the support (or overhead!) normally given by an operating system – i.e. dedicated hardware runs single, tailored programs. Consequently it is not possible to evaluate *Wisdom* by comparison with these architectures as they have none of the functionality *Wisdom* provides, such as flexibility and scalability.

The only way in which comparisons with systems of a more conventional systems can take place is at a higher level: by comparison of the services that are provided. This is equivalent to asking if *Wisdom* provides the services required of it, i.e. is it a sufficient base for a general purpose operating system. This is the first of the questions developed below upon which the evaluation is based, therefore there is no need to duplicate the answer by comparing *Wisdom* with more conventional systems.

In chapter 2, three degrees of transparency are given which were derived from the ANSA[5] work, and are necessary as to conceal the consequences of distribution. The objective of the components of *Wisdom* is to support these transparencies. Therefore, one aspect of the evaluation will be to compare the designed and implemented modules of *Wisdom* with the degrees of transparency, thereby showing whether or not *Wisdom* is capable of hiding the details of the hardware and the distribution of parallel components on it. The question that this thesis set out to answer was if meshes of processors could be used to provide a scalable general purpose operating system. The above transparency deals with the provision of an abstraction sufficient to implement a general purpose operating system. Once the above has been answered, there remains the question of whether or not *Wisdom* is scalable. Consequently, this chapter evaluates *Wisdom* by answering two distinct questions:

- Do the basic components provide a sufficient base from which a general system can be built?
- Are the basic components scalable?

The following two sections will deal with these questions in turn by studying the behaviour of, and the results obtained from, each of the basic modules in *Wisdom*.

Although most operatings systems cannot be directly compared with *Wisdom*, there are some that have been based on point to point networks of communicating processors and hence have characteristics similar to *Wisdom*. Such systems include Helios[51], PARX[21, 49], transIDRIS[46] and Trollius which are all based on the transputer. Helios is among the better documented, and so the final part of the evaluation of *Wisdom* will be a comparison with Helios.

The above evaluations are aimed at testing the concepts and ideas behind *Wisdom* and its components, rather than the prototype implementation. Therefore, when the term *Wisdom* is used, it is not the implementation that is being spoken about (although the results and characteristics of the prototype has generated are used), but instead the ideas, design and chosen mechanisms that are being referred to. Consequently, although the system was built using a slightly modified version of occam2[62] running on a network of transputers, this chapter, as far as possible, attempts to put aside the restrictions and advantages of both the language and processor, and instead concentrates on the concepts under assessment.

6.1 Sufficiency of Base

The sufficiency of the components which comprise *Wisdom* will be decided by comparison with the transparency model that was described in chapter 2. That model required the following three degrees of transparency:

- *Parallelism Transparency*: This requires that there appear to be as many processors as the program requires, that these processors are located and allocated in a manner transparent to the program, and that if more processors are added to the machine, the program can utilise them if appropriate. In *Wisdom* this is implemented by the Load Balancing module, and hence this module must support the transparent location of processors, and the execution of tasks on them.

- *Interconnection Transparency*: this requires that all processors appear fully connected with all other processors, i.e. the nodes and their interconnections form a complete network (as shown in figure 3.3(d)). This part of the model is implemented by the Router. This module must provide a mechanism for delivering messages to any processor. However, this degree of transparency requires that the physical processor still be specified – it merely hides the details of how to get to a specific processor.

- *Location Transparency*: any task can communicate with any other willing task via a mechanism which does not specify the location of the other task. This degree of transparency is handled by the Namer, and hence this module must be able to convert location independent names into the destination for messages, i.e. the locations of the tasks.

6.1.1 The Router and Interconnection Transparency

The Router module supports a complete network as required for interconnection transparency. This module, like the Load Balancer, must have a component task on all the nodes of the system to operate correctly. The service the Router provides is one of unreliable datagrams (though during the use of the prototype message loss has only occurred when tasks are not expecting messages rather than due to the Router destroying messages in transit), which is guaranteed to avoid deadlock.

The mechanisms used by the Router cannot directly support either broadcast or multicast messages. There are mechanisms to support broadcast messages, such as flood messages which visit every processor. The mechanism that supports flood messages involves sending a copy of the message to all neighbouring processors. When a processor receives such a message it consults lists of those messages it has already seen. If it has already seen the message then it ignores it, otherwise the processor sends a copy to all of its neighbours. However these mechanisms have not been implemented as there is not, as yet, a need for them.

The Router supports a complete network, and in that respect provides interconnection transparency, with the exception of facilities for broadcast messages. It may be argued that the support of an unreliable service does not support interconnection transparency, but as individual interconnections could be unreliable this is not the view taken. However, the latencies in message delivery (approximately 1ms, using the current routing module, for each processor through which the message must travel) do affect the transparency since the delays can become noticeable to the tasks, especially in a large network. This can be largely overcome by the use of special purpose hardware (a communications coprocessor),[†] which performs the work currently undertaken by the Router, as this will reduce latencies sufficiently for them to be undetectable, or nearly so. Thus the Router provides interconnection transparency on small

---

[†] Providing communications support by using a second processor at each node which is dedicated to message routing will remove the problems of the load message routing places on the processors running programs, but it will not speed up message delivery.

networks (such as the prototype), but the larger the network the more desirable routing hardware becomes to enable the necessary transparency.

6.1.2 The Namer and Location Transparency

The Namer module, with some help from the guardians, provides location transparency in *Wisdom* by mapping names on to services (or tasks). The names are location independent since the location of the task does not affect its name: any task in *Wisdom* can register itself with the Namer using any name it wishes (subject to such restrictions as avoiding duplication, mechanisms for which were touched on in the introduction to chapter 4). The Namer performs this by matching any name to a channel, or location dependent IMC name, and passing requests on to the service (or task) via this channel. The Namer module itself is comprised of an arbitrary number of component tasks, each of which implements one or more ‘directories’. The names implemented by the entire Namer module are viewed as being hierarchical, and comprise of a simple syntax: component names are made of ASCII strings, and are combined together with a special character.

The name space supported is very flexible: any task can register itself with the Namer, and therefore support further name resolutions (in effect become part of the Namer). Consequently, the addition of any new services or facilities, such as printers, remote file systems or network interfaces can occur without affecting the existing system in any way, or requiring any changes to be made.

The tasks use this module by sending the name of the service they require to the Namer together with an IMC channel on which they receive a channel (IMC connection) from the service. The channel represents the address of the service. However, the Guardians perform a mapping of IMC channels on to addresses, so that the details of the address are hidden from the task. Since the channels correspond to an address the service must remain at the same location once the name has been resolved. The extent to which this is a disadvantage depends on the characteristics of the tasks: if they are liable to move then this can become a serious drawback, which can only be overcome by repeated name resolution, or by an IMC mechanism which supports location transparency (typically this is done by providing a forwarding mechanism, as is done in DEMOS/MP) − both of which are costly in time, memory and communication bandwidth.

The Namer module provides complete location transparency, since the names contain no location information, and the only way to gain access to a service is via the location transparent names implemented by the Namer (i.e. there is no way to specify a location − it must be found using the Namer). The only limit on the transparency is that the IMC connections which are used to communicate with the service represent fixed addresses. Thus, although the name contains no location information, the service must remain at the same location, once it has registered itself with the Namer.[†] This limitation is not as great as it may seem since a task cannot change locations as the current load balancing module does not move tasks during their execution.

6.1.3 The Load Balancer and Parallelism Transparency

The load balancing module supports parallelism transparency by performing all the necessary work to locate the least loaded processor[‡] and to arrange for the task to be run on that processor on behalf of that task. However, two main constraints have been placed[††] on the action of the Load Balancer: firstly it does not move running tasks due to the large overheads of so doing (several people have shown that the cost of performing runtime migration is higher than the benefits it gives, as discussed in chapter 5) and,

---

[†] It is possible for the service to deregister itself from the Namer, to move and then reregister with the Namer, given that it has no other connections.

[‡] Other performance criteria are possible for the choice of the processor such as CPU utilisation, response time, waiting time or throughput as was mentioned in chapter 5

[††] This has been done for implementations reasons, and to take account of the results obtained by other researchers, rather than being a fundamental facet of load balancing

secondly, it only balances to directly connected processors to minimise the communication overheads. The result is that the Load Balancer behaves well for dynamic loads – ones in which there is a high turnover of tasks.

When the load balancing module is given a dynamic load, one that comes from a variety of different and changing locations, then it maintains a high processor utilisation, and finds the least loaded processor from those that it interrogates. However, if the load (i.e. the tasks) are all generated from a single fixed node, then the processors that are used, and hence the parallelism transparency and processor utilisation, are limited by the neighbourhood (the number of processors with which direct load balancing may take place) of the load balancer. This was borne out by the simulation runs, and by the final test (figure 5.12) which gave results very close to those of the simulator, and a low imbalance figure. The module finds, correctly, the lowest loaded processor from the neighbourhood of processors that it inspects (currently only those directly connected to it), as is evidenced by the results presented in chapter 5, and shown in the graph of compute bound processes, figure 5.11. Unfortunately, it performs less well on communication bound tasks, however the higher the number of tasks, or the lower the amount of communication necessary, the better it performs. The reason for these improvements are that every communication represents a certain amount of computation, and so there is an increase in the computation which can be shared between the available processors.

In conclusion the actions of the Load Balancer are transparent to the task – it sees the same interface regardless of the eventual placement of the task it is creating. The Load Balancer will place tasks on any of the processors it is connected to, and the choice is made based on the least loaded processor. Thus all processors in the neighbourhood (currently those directly connected) of the source processor may be transparently used and a maximal amount of parallelism is maintained, subject to the size of the neighbourhood. When more processors are added, they may also have tasks placed on them, again transparently to these tasks. Hence processors added to the system will be used by tasks without these tasks needing to be altered (given that they have sufficient parallelism). Therefore the Load Balancer maintains the required degree of parallelism transparency, within the limitations imposed by balancing only to directly connected processors.

6.2 Scalability of Base

In the previous three sections the components of *Wisdom* where examined to see if they are capable of supporting the necessary transparency to make the hardware appear flexible enough to provide a general purpose computing system. This section answers the second of the evaluation questions by examining how each of the components scale, and hence how *Wisdom* as a whole scales.

6.2.1 The Router and Scalability

Since a mesh is the basis of the communication system, it is possible for messages between different pairs of nodes not to interfere with one another, in direct contrast to the behaviour that is exhibited by broadcast systems. Where multiple paths exist between nodes the behaviour improves, and the chosen method will perform adaptive routing (i.e. select the unused, or least used route) at least to a limited extent. In cases where the paths of the messages must overlap for part of their route, the interference is limited only by the bandwidth of the common connection. All of these points have been shown by the results obtained from the Router, and have been presented in chapter 3, in figures 3.7, 3.10 and 3.11. These are important since they show that introducing more processors into the network need not affect communications in other parts of the network.

Unfortunately, there are some other effects of the Router which will limit scalability. Although the delays are reasonable for short journeys (of the order of a few milliseconds), when the mesh of processors becomes large the delay will eventually increase to be noticeable: with a ten by ten array the maximum delay is approximately an almost acceptable 25ms, but with a network of one hundred by one hundred processors (10,000 processors!) the maximum delay is somewhere near a fifth of a second. The result is that for a scalable system hardware routing is essential both to remove the load on the processors and to speed up message delivery (bandwidths of 360Mbyte/s, and latencies of about a few hundreds of nanoseconds are expected from some of the self-timed wormhole routing chips).

The Router provides good, non-interfering communications so that localised communication is not affected by messages in other parts of the network. This means that increasing the size of the network will not slow down existing communications, and if the programs can now be spread better over the network they speed up. The apparently slow speeds for messages are only for messages that must traverse the entire network. This is not usually the case since communicating components will be created close together by the actions of the Load Balancer. Thus the Router is scalable.

6.2.2 The Namer and Scalability

The approach used by the Namer has several advantages, over and above those given by the location transparency it supports. It is scalable in that it is divided up into components – each constituent task in the Namer is capable of working independently of the others, and hence as many name resolutions *may* take place as there are components in the Namer. Avoiding broadcast methods for name resolution has allowed the Namer to be scaled since it is not limited by the fixed bandwidth; this also avoids overloading processors with many broadcast messages.

Unfortunately, there are disadvantages; it is possible that the Namer, or more accurately any of its constituent tasks that locates frequently used services, may become a bottleneck. In the current system (of nine processors) the Namer is limited by the Router: it can handle the messages as quickly as they are delivered to it. However, if hardware routing, or faster software is provided, there will be more name resolution messages arriving, and it is at least possible that it may become a bottleneck. Whether or not this will occur, and to what extent is uncertain: the prototype is too small (and suffers from other restrictions) to enable accurate predictions on this.[†]

6.2.3 Load Balancing and Scalability

The advantages of the method used for the current load balancing module are that it is simple, fast and places a low overhead on the processor and communications medium: these points have been borne out by the results given in the load balancing chapter (chapter 5). The Load Balancer is comprised of a task that must execute on each processor; it interrogates neighbouring Load Balancers for the loads on their processes and places tasks based on this information. Other than asking for neighbouring loads the Load Balancer tasks act independently. These represent important scaling facets – increasing the number of processors in the network will not increase the load at any single point, but spreads it out over the new processors.

This method does, unfortunately, have some disadvantages: since the balancing of tasks takes place only with the immediate neighbours that a processor has, this mechanism is of little use where all child tasks are generated from a single parent, and so limits the transparent scalability of programs (though not the number of users). There are ways to overcome this by using knowledge of the load balancing mechanism, for example consider the creation of a pipeline of tasks by a parent task (such as a command interpreter). Rather than having the parent create all the tasks and then connect them together, it could create a new task, and send to that the list of commands to be executed. This new task would then

---

[†] Rough calculations give the time for the resolution of a single nametoken, and the associated time to process messages (receiving them from the guardian, and then returning them to the guardian) at approximately 0.5ms, meaning that a single node with no other load present should be capable of handling approximately two thousand resolutions per second. Assuming each task requires two name resolutions on average (input and output ‘files’), and all resolutions have to pass through a single namer task, then the Namer should be capable of supporting about one thousand new tasks per second. Assuming a ten by ten network, that would allow a single node to produce a new task every 100ms, which seems reasonable for a ‘flat-out’ behaviour. More accurate calculations are difficult due to the background load introduced by the Router that would be reduced or removed if hardware routing where supplied. Likewise, much of the speed is dependent upon the details of the implementation, and therefore could be subject to changes.

create, locally, the first of the commands in the pipeline, proceed to create another copy of itself, which is automatically balanced, to which it would then send the remainder of the pipeline, and a connection from the first command. This ‘recursive’ mechanism would worm its way through the network placing the components on adjacent processors as it goes, making use of as many processors as possible. This does, however, reduce the transparency of load balancing.

Another method of improving the Load Balancer would be to increase the neighbourhood of processors to which load balancing could take place, i.e. balance to processors other than those immediately connected. This has the advantage of increasing the transparent scalability of programs. The current system has a neighbourhood of five processors, including the source; the next highest neighbourhood, for a regular mesh, is thirteen (the processors labelled two as well as one in figure 6.1), and therefore could not be investigated as only nine processors were available. The benefits this accrues will depend on the cost of communication as well as the cost associated with the load metric, site selection and actual task placement, and so must be studied with care to ensure that this alteration does, in fact, result in an improved performance. Specifically, if two communication bound tasks can be placed four processors apart, then unless the communications are noticeably faster than currently implemented, the tasks will execute more slowly and place a higher load on the intervening processors than had been the case with the previous method.

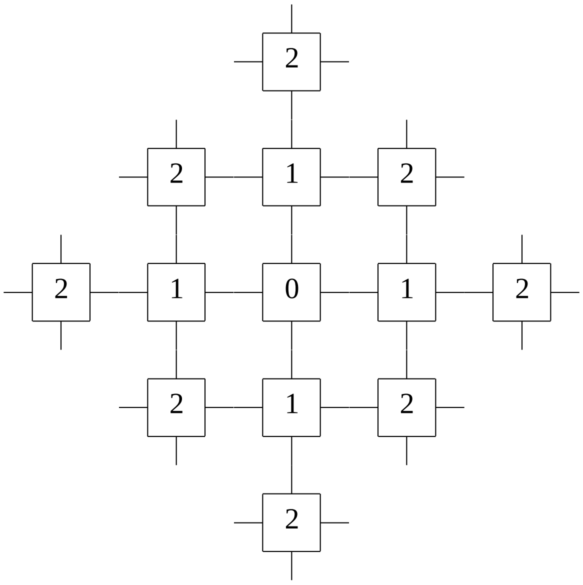


Figure 6.1: Neighbourhoods for load balancing in Meshes

6.3 Comparison with Helios

Helios differs from *Wisdom* since it aims to support a flexible topology, rather than the fixed mesh which has been the subject of this research. A consequence is that many of the design decisions have been different from those taken here, and Helios is therefore considered as the basis of a good comparison, since it allows evaluation against a system which has the same objectives, but has used a different implementation route. The disadvantage of comparisons with Helios is that there are no figures available to indicate the speeds of which it is capable, so benchmark comparisons cannot be carried out (but then, benchmarks are often of little use, either being tailored to exhibit the best parts of the system under study, or the system has been tailored to given an unrealistic results for benchmarks).

The structure of Helios has been described earlier, in chapter 1, but an outline follows. The processors on a Helios machine either run the Helios nucleus, or may be dedicated to programs, i.e. have no operating system support on them, but these latter nodes must be leaves in the network map. The nucleus consists of a kernel, several libraries, a processor manager and loader. The kernel supports message passing, memory management and process creation and scheduling with a hardware independent interface. The various libraries provide a POSIX compatible system. The processor manager supplies some task creation and deletion support, handles signal delivery, environment enquiries, name table management and server location. The loader interprets the load images, and binds to the necessary libraries. Load balancing is provided by the Component Distribution Language (CDL) compiler and Task Force Manger (TFM) which run as ordinary tasks.

The basic facilities Helios provides are the same as those supported by *Wisdom*: a message delivery mechanism, a naming mechanism and a Load Balancer. In addition it supports a file system and both terminals and X servers, which are implemented as user tasks, as well as the usual complement of utilities. These various items (the ‘additions’) do not comprise part of the comparison as the evaluation is studying the support for these tools, and not the tools themselves.

The message delivery mechanism of Helios uses capabilities, each of which has a corresponding port. A port either points to a task to which the message is to be delivered (local ports), or to a port on a neighbouring transputer to which the message should be forwarded (surrogate ports). Hence many ports are combined in a chain (one pointing to the next) over which messages are sent. Although these ports appear to support a virtual circuit, message delivery is not reliable, as the system reserves the right to destroy messages. Routines exist in the nucleus libraries which support reliable delivery. When a port is sent in a message, the necessary chain of linked ports is created by message delivery software. This means that a more flexible topology can be supported, but that messages may not take optimal, or adaptive routes and that resources are tied up maintaining the ports. As the chain of ports is created during message delivery (there being no mechanism to create a chain of ports between two specified processors), the delivery mechanism must be able to find the port in the message to create this chain. Consequently, ports can only be sent in known parts of the message. In comparison with *Wisdom*, Helios does support a more flexible network topology, but it cannot route messages in an adaptive manner, nor can the IMC channels occur anywhere in the message, and Helios requires more resources (a table of ports for message delivery).

The naming mechanism in Helios provides a hierarchical name space. Each processor maintains its own cached name table of known names, which is consulted when a name is being resolved. If the name is not locally known it performs a broadcast search for the name by flooding the network. This causes every processor to consult its nametables, and any processor which knows the name replies with a port which enables access to the service (currently only the first reply is used). These actions are handled by the processor manager and libraries of the Nucleus. The claim is that the broadcast flood search is acceptably fast, and will scale, though though there is no evidence to support this. *Wisdom* does not support caching, and requires that all the component namer tasks cooperate to allow a name to be resolved. To offset this, broadcast messages are not used (i.e. the network is not flooded) to locate a name which reduces the communication overhead of service location. Since services are located through the Namer, rather than by a reply to broadcast message, a greater degree of trust can be placed in the result of name resolution. Finally *Wisdom* does not require that every processor supports a namer task.

Helios supports load balancing via two user programs: the CDL compiler and the TFM. The CDL is a simple language, which provides a method of describing the components of a calculation, the requirements they have (e.g. processor type, or memory needs) and their interconnections. The TFM uses heuristics and a map of the network on which the program is run to allocate the component tasks to available processors, taking into account the interconnections of the tasks. This allows parallel programs to be described in a way ignorant of the underlying hardware and have it mapped on to the hardware at runtime. *Wisdom*, when compared with Helios, places tasks in a manner that may make less efficient use of communications (*Wisdom* ignores communication interconnections), and, unless a program makes special allowance, it will not use all the processors in a network, unlike Helios. However, the *Wisdom* method is simpler and computationally cheaper; it also takes into account the current load on the processors in the system. The Load Balancer also has the advantage of not requiring a network map, which is the case with Helios.

Both *Wisdom* and Helios provide a similar base set of functions, designed to operate on a point to point network without support for broadcast. However, Helios is designed to use a far more flexible network (processors may be inserted or connected anywhere), which introduces complexities into the Router, meaning that it takes more resources than does the mechanism used by *Wisdom*, and requires a map of the network to be provided for the TFM (the load balancer) to operate. Likewise the name resolution mechanism can place a heavier load on the communications network than the *Wisdom* Namer. Finally the load balancing in Helios does provide a better solution than that of *Wisdom* (Helios may place tasks anywhere, but *Wisdom* can only place on adjacent nodes), though this is neither as flexible nor able to take into account current loads. Thus the model and mechanisms chosen provide all the support that Helios does, in a manner that rarely performs worse, and will generally perform better.

6.4 Summary

The first of the evaluation questions has shown that the three principle components of *Wisdom* – naming, routing and load balancing – provide the necessary degrees of transparency, and the necessary support, to allow a mesh of processors to be used as a general purpose computer environment. Scaling, the major reason for investigating processor meshes, seems feasible for up to hundreds of processors given a few improvements to the current prototype, and appropriately spaced task creation points, or and increase in the load balancing neighbourhood. It should be possible to go beyond these limits given suitable routing hardware and the application of the various improvement methods mentioned in previous chapters, and thereby build usable networks of thousands of processors – numbers of the order that had originally been hoped for. However, a computer built from about one hundred processors, running at the speeds at which current processors are capable of, does represent a sizeable processing resource, which *Wisdom* has shown can be usable and flexible.

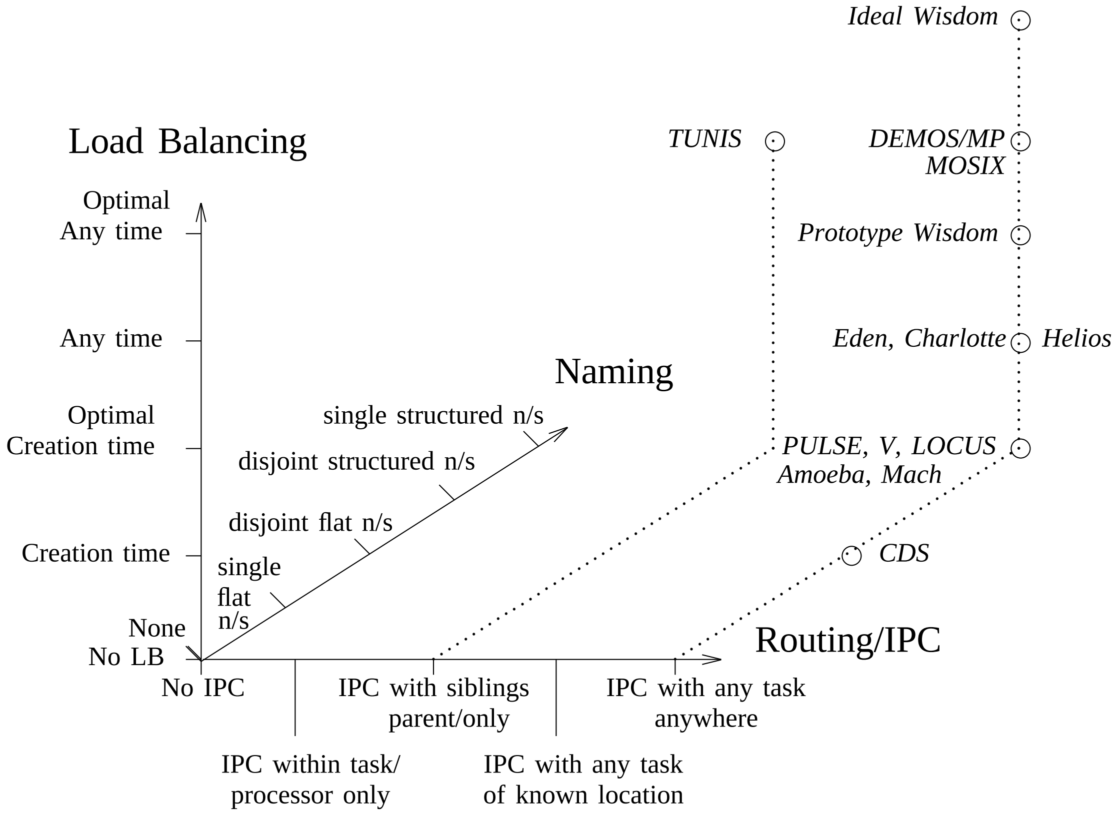


Figure 6.2: Realised *Wisdom* Transparency in the context of other systems

These results show that meshes are suitable for the scalable general purpose environment envisaged *given suitable support*. This suitable support involves the use, and availability, of suitable hardware at attractive prices (including routing hardware). Knowledge of both the environment and the component modules must also be applied: the Load Balancer behaves well provided that the tasks profile is dynamic and their creation points are flexible and spread out. In practice this is akin to saying that each user should be connected to the system (either by physical terminal lines or by IMC channels) at points away from other users. Before this system can be used, however, other components must be implemented – most notably a file system and command interpreter. These are *not* considered as essential foundation components since they utilise the modules provided, and do not provide a service which must be used by all others.

In chapter 1 various operating systems were compared with each other on the basis of the transparency they provided. The next chapter, chapter 2, showed how it was hoped *Wisdom* would compare with other operating systems using the same transparency comparisons – this represented the target for *Wisdom*, or how the ideal version would compare. Figure 6.2 shows how the prototype version of *Wisdom* compares with the other systems. When viewing this figure there are two factors to be borne in mind, firstly that research has indicated that moving further along the axis labeled load balancing is not necessarily beneficial. Secondly this figure takes no account of the scalability aspects that *Wisdom*, unlike other operating systems shown, exhibits.

Chapter 7

Further Work and Conclusion

> *Little story now concluded, but history of world unfinished. Lovely ladies, kind gentlemen: go home to ponder. What was true at beginning remains true. Play make man think. Thought make man wise. And Wisdom make life endurable.*
>
> *Marlon Brando* giving the last lines in the film *The Teahouse of August Moon*

This thesis has examined the issues in, requirements of, and problems with, building a scalable operating system for a network of point to point connected processors. The lessons of research in the field of distributed operating systems, and other specialist work in appropriate areas were used to develop a model which would provide the necessary support for a usable scalable system. This model consists of three degrees of transparency: interconnection transparency, location transparency and parallelism transparency. Each of these degrees of transparency is supported by one of the basic elements of *Wisdom*: naming, load balancing and routing. These three components – the foundations of a scalable operating system – were examined, and various ways of implementing them were considered, and the most appropriate chosen, with the relevant reasons and alternative choices being given in the appropriate chapters. Finally, in the previous chapter, these three components were evaluated against the objectives of the system.

The results given have shown that it is possible to build a scalable general purpose operating system using a mesh of processors. The three components have been shown to provide a sufficient foundation on which all other necessary services can be built, and which can, in turn, exploit the power of a scalable system. The Namer represents such a service, despite being one of the basic elements – it ignores the underlying hardware and its interconnections. There are potential problems and deficiencies with both the hardware available and the implementation of *Wisdom* which appear to restrict the scalability of the prototype. However, these areas have been identified and solutions proposed which either overcome or avoid the bottlenecks, and are feasible to implement.

7.1 Further Work

Like almost all research, the work described here has produced as many questions, or areas deserving future investigation, as it answers. Some of these relate directly to the work, such as scaling, others to the problems of implementing the work, and finally areas needed to make use of this work, such as environments to encourage the use of parallelism. *Wisdom* has, fortunately, reached a stage from which it can be used as a base for such studies, so it is hoped that it will be both the object of future research and a tool with which to perform this work. The following sections cover the most important areas of future work.

7.1.1 Scaling

The prototype has been built on a three by three network, and performs well at this size. However, one of the main areas of investigation was on the scalability of the system, and so its behaviour on larger networks should be examined, and means of improving the transparent scalability of programs. Also, the prototype has not been used to solve ‘real’ problems, only test cases have been run on it: this does not constitute a real test of the system, so ideally work on scaling should involve using real applications to solve real problems, and to see how *Wisdom* performs.

The network so far evaluated has been limited to a square array, but there is no reason why, whilst still maintaining a mesh, that this must be the only shape that can be used. Other shapes, variations of rectangles and tori, should be examined to decide the optimal solutions. These shapes are likely to be influenced by the number and placement of fixed resources, such as disks, and so these, too, should be considered in deciding the optimal shape. The network has also be limited to a two dimensional mesh, but the various components of *Wisdom* are designed so that increasing the mesh to three dimensions would make little difference, and this might be worth investigating. However, this will introduce physical construction problems as in larger networks processors could become inaccessible in the middle of a three dimensional computer, or the wiring, to maintain accessibility could become tortuous and lengthy, potentially slowing down communications.

There are also aspects of the modules as implemented that may become limiting factors in scaling the system, for example the Namer, and the neighbourhood size of the Load Balancer. Both of these should be examined to decide if they are limiting factors, and which of many methods described can be used to overcome these limits (if they can be overcome), as well as studying the effects these alterations have on other areas to see if they may introduce more bottlenecks or size limits.

7.1.2 Communication Mechanisms

The most immediate problem, and the one which will have the most impact on the performance of the system, as well as its scalability, is the speed of communication. It is true that the routing system has not been optimised for speed as much as have some systems (such as Amoeba), but it is very unlikely that anything more than a single order of magnitude improvement in speed will be obtainable. This, although far faster than some other systems, is still very slow compared with the almost instantaneous transmission of information that is afforded by information sharing from within a single task, or the high speeds attainable between tasks within a single processor.

Hardware has been designed and built which provides transfer times of a microsecond or less. These systems also remove from the processors the overheads of message routing. The effect of such hardware on the behaviour of the system will be great: bottlenecks, or throttles, caused by the communication system will no longer be present, and so certain assumptions may no longer be needed, or new ones may be introduced. Faster communications will also have implications for the scaling of such components as the Load Balancer – it may well become acceptable to place tasks at far distant locations without unacceptable performance degradation, and hence greater neighbourhoods could then be considered.

7.1.3 Improved or Ideal Hardware

The chosen processor (the transputer), although better in many ways than conventional processors, has several drawbacks when applied to the environment envisaged here. It has become clear, through the experience of implementing *Wisdom*, that certain facilities are necessary for a scalable system, and others would be of great benefit. These areas, and the identification of other areas of potential support, should be investigated. The following lists these facilities:

- Wormhole routing communications coprocessor. This is necessary to attain the communications speeds which allow the efficient use of parallelism, and to remove from the processors the overheads of handling message delivery (in effect this speeds up the processors by two different means: faster communications and removing message processing overheads).

- Access to, and ability to manipulate, process queues. In a general purpose system it is necessary to have total, or near total, control over processes to allow the removal of errant or unnecessary processes. All other requirements can be handled by judicious use of adding processes to process queues, or of communication primitives. Support for multiple priority levels would also be of great use, though not vital (it can be simulated by swapping queues).

- Memory protection with access to this for the communications coprocessor, but not necessarily virtual memory. This is necessary to provide protection against errant processes, or against attack by malicious ones. The communication coprocessor should have access to the necessary information to allow it to perform communications on behalf of processes without having to interrupt the processor.

- Conventional Cache. This is to increase the effective speed of the processor: it is not possible for a fixed high-speed memory to be efficiently shared between several tasks if its loading must be done by the programs.

- Fail-Handle, rather than the pseudo fail-stop or fail-ignore modes available on the transputer. The problem is that an error condition either halts the processor requiring the intervention of a neighbouring processor, or that it is ignored and the processor proceeds on a destructive path. Many problems, in a general purpose operating system, can be handled by the processor itself (memory violations, arithmetic errors or unexpected messages) without requiring the intervention of another processor (though this mechanism is useful, and support for it should remain).

It is, perhaps, interesting to note that most of these are to be provided by the new INMOS transputer, the H1.

7.1.4 Languages

For the envisaged system to be used in such a way as to allow individual programs to benefit from scaling of the hardware, these programs must be written with parallelism built into them. There are a multitude of languages which express parallelism at all different granularities, and with different degrees of explicitness. Languages such as occam allow programmers to express more parallelism than can be handled in *Wisdom* (each statement is a task), whilst others such as Ada allow less parallelism but at an overhead which is sometimes too great to be of use. Investigation is needed into a language which supports message passing, or can use it efficiently, avoids an inherent need for shared memory, and is both easy and attractive to use whilst at the same time allowing some method of analysing the behaviour of its parallel components, such as the rules of transformation that exist for occam[78].

On a similar theme, work is needed to provide distributed parallel debuggers which have a minimal impact on the task being examined (timing bugs may disappear when the debugger is applied, hence making their removal difficult).[†] Improvements will also be needed on the interface to the debugger as it will now have to convey information on many activities which are happening in parallel, as well as displaying information on messages and their transfer.

7.1.5 File System and Other Services

Although there is an extensive body of knowledge on how to build the various components that *Wisdom* needs to provide the full services of an operating system, work is needed on how they may best exploit the nature of a mesh of processors, and provide services to large number of truly parallel tasks. Most important is that of a filesystem, and work is underway both here at the University of York[8], and at Hewlett-Packards Palo Alto research labs[93]. It would seem that the most important part of this work will be the study of efficient caching, since it is envisaged that there will always be more processors than fileservers, and so the file servers will be potential bottlenecks unless good caching algorithms are available.

7.1.6 Component Failure Survival

So far this thesis has ignored the issues of reliability and availability. However, in a system comprised of tens, hundreds or even thousands of processors it would be reasonable to hope that the failure of a single processor would not cause the entire machine to fail. Thus work should be done on modifying *Wisdom* so that it can survive component failures. In practice, the design of *Wisdom* will greatly simplify this: only the Router and Load Balancer need have this support as all other parts of the system are unaware of specific processors.

7.2 Conclusions

Wisdom has been shown to be the foundation of a usable scalable parallel operating system, and hence that meshes of processors can be used to provide a general purpose computing environment. *Wisdom* is not the only system which is based on a scalable point-to-point network (others using the transputer include Helios, Trillium and PARX). All these systems operate successfully, though, generally, they also suffer from the lack of specialised hardware as described above.

In the introduction, it was mentioned that one branch of multicomputers has evolved into the floating point coprocessors that are now available for many processors; the suggestion of specialised routing hardware represents a very similar development. It is possible that other modules may also be rendered into specialised hardware or coprocessors – the design of *Wisdom* would allow this to happen with a minimal adverse impact.

One issue that has not been tackled is the uses to which *Wisdom* might be put. Despite the trend for workstations on desks there is the need for powerful centralised computers to perform detailed simulations, large compilations or format long documents. Workstations can be efficiently handled as intelligent graphics terminals connected to a centralised computer using efficient 'graphics' protocols, such as the blit primitives[70]. By including much of the workstation's processing power in the central resource it can be used whenever needed by whoever needs it, rather than only when the workstation is switched on, and generally only by the owner of the workstation.[††] It would seem, therefore, that a *Wisdom*-like central machine would be useful for providing high computation power for workstations, or as a system to which many graphics terminals could be connected which then provide workstation like facilities.

[†] During the development of *Wisdom* there have been several occasions when bugs have appeared which proceeded to disappear as soon as debugging traces were inserted. The eventual, and unsatisfactory, method of debugging was to introduce errors which cause the program to halt, and track down the bugs by moving these halt points, running the program(s) from scratch each time they where moved, and hence the location of the bug was tracked down.

[††] Some work has been done on harnessing the power tied up in idle workstations[59], and although it does find much useful processing power, much is still lost through moving work between machines, or machines simply having been switched off.

References

1. Tahir Afzal, ‘‘Load Balancing Unix United’’, SRM/406, Computing Laboratory, University of Newcastle-upon-Tyne, UK (October 1985).

2. N. Alon, A. Barak and U. Manber, ‘‘On Disseminating Information Reliably Without Broadcast’’, *Proc 7th International Conference on Distributed Computing Systems,* Berlin, pp. 74-81 (September, 1987).

3. R. Alonso, *The Design of Load Balancing Systems For Local Area Network Based Distributions*, U.C. Berkeley Publications (Fall 1983).

4. G. Anderson and E. D. Jensen, ‘‘Computer Interconnection Structures: Taxonomy, Characteristic and Examples’’, *ACM Computing Surveys* **7**(4), pp. 197-213 (December 1975).

5. Advanced Systems Network Architecture, *ANSA Reference Manual Release 0.03 (Draft),* ANSA, 24 Hills Road, Cambridge, UK (June 1987).

6. Yeshayahu Artsy, Hung-Yang Chang and Raphael Finkel, ‘‘Interprocess Communication in Charlotte’’, *IEEE Software* **4**(1), pp. 22-28 (January 1987).

7. Y Artsy, H Y Chang and R Finkel, ‘‘Processes Migrate in Charlotte’’, #655, University of Wisconsin-Madison (Aug 1986).

8. P. B. Austin, *Sage: A Parallel Scalable File System (2nd Year January Report)*, Dept. of Computer Science, University of York (10 January 1990).

9. A. Barak and G.O. Paradise, ‘‘MOS - A Load Balancing UNIX’’, *Proc. Autumn 86 EUUG Conference,* Manchester (UK), pp. 273-280, EUUG (September 1986).

10. A. Barak and A. Shiloh, ‘‘A Distributed Load-balancing Policy for a Multicomputer’’, *SOFTWARE - Practice and Experience* **15**(9), pp. 901-914 (September 1985).

11. A. Barak and R. Wheeler, ‘‘MOSIX: An Integrated Multiprocessor UNIX’’, *Proc of the Winter 1989 USENIX Conference,* San Diego, CA, pp. 101-112 (February, 1989).

12. Robert Baron, Richard Rashid, Ellen Siegel, Avadis Tevanian and Michael Young, ‘‘Mach-1: An Operating Environment for Large-Scale Multiprocessor Applications’’, *IEEE Software* **2**(4), pp. 65-67 (July 1985).

13. F. Baskett, J. Howard and J. Montague, ‘‘Task Communication in DEMOS’’, *Proceedings of the Sixth ACM Symposium on Operating Systems Principles*, pp. 23-32 (November 1977).

14. B.Bershad, ‘‘Load balancing with Maitre d’’’, 86/276, EECS computer science division, University of California, Berkeley, CA 94720 (Dec 1985).

15. B. Beck and B. Kasten, ‘‘VLSI Assist in Building a Multiprocessor UNIX System’’, *USENIX Association Conference Proceedings*, Portland, Oregon, pp. 255-276 (Summer 1985).

16. Andrew D Birrell, Roy Levin, Roger Needham and Michael D. Schreder, ‘‘Grapevine: An Exercise in Distributed Computing’’, *CACM* **25**(4), pp. 260-274, Xerox Palo Alto Research Center (April 1982).

17. A P Black, E D Lazowska, J D Noe and J Sanislo, ‘‘The Eden Project: A Final Report’’, 86-11-01, University of Washington, Seattle, Dept. of Computer Science.

18. C. Boldyreff, ‘‘Draft Proposed ANSI/ISO C Standard and POSIX Standards Developments’’, *European UNIX User Group News Letter* **8**(1), pp. 48-50 (Spring 88).

19. C. Boldyreff, ‘‘The POSIX Standard and Its Future Development’’, *European UNIX User Group News Letter* **8**(4), pp. 45-46 (Winter 88).

20. C. Boldyreff, ‘‘POSIX Standarisation Developments’’, *European UNIX User Group News Letter* **8**(3), pp. 34-35 (Autumn 88).

21. J. Briat, M. Favre, D. Fort, N. Gonzalez-Valenzuela, Y. Langue, T. Muntean and Ph. Waille, ‘‘PARX: A Parallel Operating System for Transputer Based Machines’’, *Proceedings OUG-10, Applying Transputer based parallel machines*, Enschede, Netherlands (3-5th April).

22. D.R. Brownbridge, L.F. Marshall and B. Randell, ‘‘The Newcastle Connection or UNIXes of the World Unite!’’, *Software – Practice and Experience* **12**, pp. 1147-1162 (1982).

23. D.R. Cheriton, ‘‘An Experiment Using Registers for Fast Message-Based Interprocess Communication’’, *Operting Systems Review* **18**(4), pp. 12-20 (October 1984).

24. D. R. Cheriton, M. A. Malcolm, L. S. Melen and G. R. Sager, ‘‘Thoth, a Portable Real-Time Operating System’’, *Communications of the ACM* **22**(2), pp. 105-114 (February 1979).

25. David R. Cheriton and Timothy P. Mann, ‘‘A Decentralised Naming Facility’’, STAN-CS-86-1098, Department of Computer Science, Stanford University (February 1986).

26. D. R. Cheriton, *The Thoth System: Multi-Process Structuring and Portability*, North Holland (1982). Operating and Programming Systems Series

27. David. R. Cheriton, ‘‘The V Distributed System’’, *Communications of the ACM* **31**(3), pp. 314-333 (March 1988).

28. D.R. Cheriton, ‘‘The V. Kernel: A Software Base for Distributed Systems’’, *IEEE Software* **1**(2), pp. 19-43 (April 1984).

29. D. Cheriton and W. Zwaenepoel, ‘‘The Distributed V Kernel and its Performance for Diskless Workstations’’, *Proceedings of the Ninth ACM Symposium on Operating System Principles*, Bretton Wood, New Hampshire, pp. 128-139 (October 1983).

30. E. Chow, H. Madan and J. Peterson, ‘‘A Real-Time Adaptive Routing Network for the Hypercube Computer’’, *Proceedings of the IEEE Real-Time Systems Symposium*, San Jose, California (1-3 December 1987).

31. George F. Coulouris and Jean Dollimore, *Distributed Systems: Concepts and Design*, Addison-Wesley, London (1988).

32. Daley, R. C. and Dennis, J. B., ‘‘Virtual Memory, Processes, and Sharing in MULTICS’’, *Communications of the ACM* **11**(5), pp. 306-312 (May 1968).

33. William J. Dally, Linda Chao, Andrew Chien, Soha Hassoun, Waldemar Horwat, Jon Kaplan, Paul Song, Brian Totty and Scott Wills, ‘‘Architecture of a Message-Driven Processor’’, *Proceedings of the 14th International Symposium on Computer Architecture*, pp. 189-205, The Computer Society of the IEEE (June, 1987).

34. D L Eager, E D Lazowska and J Zahorjan, ‘‘A Comparison of Receiver-Initiated and Sender-Initiated Dynamic Load Sharing’’, 85-04-01, University of Washington, Seattle, Dept of Computer Science, Seatle (April 1985).

35. P. H. Enslow, ‘‘What is a ‘Distributed’ Processing System?’’, *Computer* **11**(1), pp. 13-21 (January 1978).

36. P.A. Ewens, R.C. Holt, M.J. Funkenhauser and D.R. Blythe, ‘‘The Tunis Report: Design of a UNIX-Compatible Operating System’’, CSRI-176, Computer Systems Research Institute, Univeristy of Toronto, Canada (January 1986).

37. P. Ewens, D. R. Blythe, M. Funkenhauser and R. C. Holt, ‘‘Tunis - A Distributed Multiprocessor Operating System’’, *USENIX Association Conference Proceedings*, Portland, Oregon, pp. 247-254 (Summer 1985).

38. D. Ferrari, ‘‘A study of load indices for load balancing schemes’’, 86/262, EECS computer science division, University of California, Berkeley, CA 94720 (Oct 1985).

39. D. Gelernter and A.J. Bernstein, ‘‘Distributed Communication via a Global Buffer’’, *Proc. 1st ACM SIGACT-SIGOPS Symp. on Princ. of Distributed Computing*, pp. 10-18 (1982).

40. R. L. Graham, E. L. Lawler, J. K. Lenstra and A. H. J. Rinnooy Kan, ‘‘Optimisation and Approximation in Deterministic Sequencing and Scheduling: A Survey’’, *Ann. of Discrete Maths.* **5**, pp. 287-326 (1979).

41. C.H.R. Grimsdale, ‘‘Using CDL’’, 15, Perihelion Software, Shepton Mallet, Somerset, England (October 1988).

42. Andrew Hopper, Steven Temple and Robin Williamson, *Local Area Network Design*, Addison-Wesley, Cambridge, England (1986).

43. C. Jesshope and D. Nicole, ‘‘Supernode Architecture Evaluation’’, *Report Esprit Project P1085* **003**, Southampton University (22 May, 1986).

44. D. Keeffe, G.M. Tomlinson, I.C. Wand and A.J. Wellings, *PULSE: An Ada-based Distributed Operating System*, Academic Press, London (1985).

45. Robert M. Keller and Frank C.H. Lin, ‘‘Simulated Performance of a Reduction-Based Multiprocessor’’, *Computer* **17**(7), pp. 70-82 (July 1984).

46. P. J. King, ‘‘Implementing a POSIX Compatible Operating System on a Multi-Transputer Supercomputer’’, *European UNIX Systems User Group Autumn Conference*, Cascais, Portugal, pp. 39-51 (October 1988).

47. Leonard Kleinrock, *Queueing Systems: Volume I: Theory*, John Wiley and Sons, Los Angeles (1975).

48. R Korry, ‘‘A Load Sharing Algorithm for a Workstation Environment’’, 86-10-03, University of Washington, Seattle, Dept. of Computer Science (Oct 1986).

49. Y. Langue and T. Muntean, ‘‘PARX: A UNIX-like Operating System for Transputer-based Parallel Supercomputers’’, *USENIX Proceedings: Workshop on UNIX and Supercomputers*, Pittsburgh, PA. (September 26-27, 1988).

50. Keith A. Lantz, Klaus D. Gradischnig, Jerome A. Feldman and Richard F. Rashid, ‘‘Rochester’s Intelligent Gateway’’, *IEEE Computer* **15**(10), pp. 54-68 (October 1982).

51. Perihelion Software Ltd, *The Helios Operating System*, Prentice Hall, London, UK (1989).

52. David May, *Occam 2 Language Definition*, INMOS Limited (February 13, 1987).

53. D. May and R. Shepherd, ‘‘OCCAM and the Transputer’’, *Proc IFIP Workshop on Hardware Supported Implemention of Concurrent Languages in Distributed Systems*, Univ. of Bristol, UK (March 26-28, 1984).

54. David May and Roger Shepherd, ‘‘The Transputer Implementation of Occam’’, Technical Note 21, Inmos (February 1987).

55. C. Mead and L. Conway, *An Introduction to VLSI systems*, Addison-Wesley (1980).

56. P.M. Merlin and P.J. Schweitzer, ‘‘Deadlock Avoidance – Store-and-Forward Deadlock’’, *IEEE Transactions on Communications* **COM-28**(3), pp. 345-354 (March 1980).

57. R.M. Metcalfe and D.R. Boggs, ‘‘Ethernet: Distributed Packet Switching for Local Computer Networks’’, *Communications of the ACM* **19**(7), pp. 395- (July 1976).

58. Barton P. Miller, David L. Presotto and Michael L. Powell, ‘‘DEMOS/MP: The Development of a Distributed Operating System’’, *Software – Practice and Experience* **17**(4), pp. 277-90 (April 1987).

59. M.J.Litzkow, M.Livny and M.W.Mutka, ‘‘Condor - A Hunter of Idle Workstations’’, #730, University of Wisconsin-Madison (Dec 1987).

60. Sape J. Mullender, ‘‘Principles of Distributed Operating Systems Design’’, Doctorate Thesis, Vrije University, Amsterdam, Netherlands (31st October 1985).

61. S. J. Mullender and A. S. Tanenbaum, ‘‘The Design of a Capability-Based Operating System’’, *BCS Computer Journal* **29**(4), pp. 289-299 (August 1986).

62. K.A. Murray and A.J. Wellings, ‘‘Issues in the Design and Implementation of a Distributed Operating System for a Network of Transputers’’, *Microprocessing and Microprogramming, Proceeding of EUROMICRO ‘88*, Zurich **1988**(24), pp. 169-78 (29th August-2nd September 1988).

63. R.M. Needham and A.J. Herbert, *The Cambridge Distributed Computing System*, Addison-Wesley Publishing Company, Cambridge (1982).

64. R.M. Needham, ‘‘Names’’, *An Advanced Course on Distributed Computing*, Fingerlakes, Ithaca, pp. 95–108, ACM Press (Preprint) (July 1989).

65. L.M. Ni, Chong-Wei Xu and T.B. Gendreau, ‘‘A Distributed Drafting Algorithm for Load Balancing’’, *IEEE Transactions on Software Engineering* **SE-11**(10), pp. 1153-61 (October 1985).

66. Derek C. Oppen and Yogen K. Dalal, ‘‘The Clearinghouse: A Decentralized Agent for Locating Named Objects in a Distributed Environment’’, *ACM Transactions on Office Information Systems* **1**(3), pp. 230-53, Xerox Office Systems Division.

67. James. L. Peterson and Abraham Silberschatz, *Operating Systems Concepts*, Addison Wesley, Austin, Texas (1985).

68. Larry L. Peterson and Douglas E. Comer, ‘‘Understanding Naming in Distributed Systems’’, *Distributed Computing* **1989**(3), pp. 51-60 (1989).

69. Larry L. Peterson, ‘‘The Profile Naming Service’’, *Transactions on Computer Systems* **6**(4), pp. 341-364, ACM (November 1988).

70. R. Pike, ‘‘The Blit: A Multiplexed Graphics Terminal’’, *AT & T Bell Laboratories Technical Journal* **63**(8 part 2), pp. 1607-1632 (October 1984).

71. R. Pike and P. J. Weinberger, ‘‘The Hideous Name’’, *USENIX Association Conference Proceedings*, Portland, Oregon, pp. 563-568 (Summer 1985).

72. Gerald J. Popek and Bruce J. Walker, *The LOCUS Distributed System Architecture*, MIT Press, Cambridge, Massachusetts (1985).

73. Michael L. Powell and Barton P. Miller, ‘‘Process Migration in DEMOS/MP’’, *Operating Systems Review Special Issue, Proceedings of the 9th ACM Symposium on Operating Systems Principles.*(5), pp. 110-119 (10-13th October 1983).

74. Krithivasan Ramamritham and John A. Stankovic, ‘‘Dynamic Task Scheduling in Hard Real-Time Distributed Systems’’, *IEEE Software* **1**(3), pp. 65-75 (July 1984).

75. R. F. Rashid and G. G. Robertson, ‘‘Accent: A communication oriented network operating system kernel’’, CMU-CS-81-123, Department of Computer Science, Carnegie-Mellon University (30 Apr 1981).

76. R. van Renesse, J. M. van Staveren, J. Hall, M. Turnbull, B. Janssen, J. Jansen, S. J. Mullender, D. Holden, A. Bastable, T. Fallmyr, D. Johansen, K. S. Mullender and W. Zimmer, ‘‘MANDIS/Amoeba: A Widely Dispersed Object-Oriented Operating System’’, *Proc. of the EUTECO 88 Conf.*, Vienna, Austria, pp. 823–831, North-Holland (APR 1988).

77. D.M. Ritchie, ‘‘A Stream Input-Output System’’, *AT&T Bell Laboratories Technical Journal* **63**(8), pp. 1897-1910 (October 1984).

78. A.W. Roscoe and C.A.R. Hoare, ‘‘The Laws of occam Programming’’, PRG-53, Programming Research Group, Oxford University Computing Laboratory (February 1986).

79. C. H. Russell and P. J. Waterman, ‘‘Variations on UNIX for Parallel-Processing Computers’’, *Communications of the ACM* **30**(12), pp. 1048-1055 (Dec 87).

80. Michael D. Schroeder, Andrew D. Birrell and Roger M. Needham, ‘‘Experience with Grapevine: The Growth of a Distributed System’’, *ACM Transactions on Computer Systems* **2**(1), pp. 3-23 (February 1984).

81. Michael Schroeder, ‘‘Dec’s Autonet: A Lecture at Fingerlakes’’, *An Advanced Course on Distributed Computing*, DEC SRC (July 1989).

82. J.F. Shoch, ‘‘Inter-network Naming, Addressing, and Routing’’, *COMPCON 78*, pp. 72-79 (Spring 1978).

83. Andrew S. Tanenbaum, *Computer Networks*, Prentice Hall International (1981).

84. A. S. Tanenbaum and S. Mullender, ‘‘An Overview of the Amoeba Distributed Operating System’’, *ACM OSR* **15**(3), pp. 51-64 (July 1981).

85. Andrew S. Tanenbaum, *Operating Systems: Design and Implementation*, Prentice-Hall International, Amsterdam, The Netherlands (1987).

86. Andrew S. Tanenbaum and Robbert van Renesse, ‘‘Distributed Operating Systems’’, *ACM Computing Surveys* **17**(4), pp. 419-470 (December 1985).

87. D. B. Terry, ‘‘Distributed name servers: naming & caching in large distributed computing environments’’, 85/228, EECS computer science division, University of California, Berkeley, CA 94720 (Mar 1985).

88. M. R. Thompson, R. D. Sansom, M. B. Jones and R.F.Rashid, ‘‘Sesame: The Spice File System’’, CMU-CS-85-172, Department of Computer Science, Carnegie-Mellon University (Dec 1985).

89. Yung-Terng Wang and Robert J. T. Morris, ‘‘Load Sharing in Distributed Systems’’, *IEEE Transactions on Computing* **C-34**(3), pp. 204-17 (March 1985).

90. Richard W. Watson, ‘‘Identifiers (naming) is distributed systems’’, *Distributed Systems – Architecture and Implementation*, Munich, pp. 191-210, Springer-Verlag (March, 1980).

91. William Dally and Charles Seitz, ‘‘The Torus Routing Chip’’, *Distributed Computing* **1**(4), pp. 187-96 (1986).

92. Brent Welch and John Ousterhout, ‘‘Prefix Tables: A Simple Mechanism for Locating Files in a Distributed System’’, UCB/CSD 86/261, Computer Science Division, University of California, Berkeley (October 1985).

93. John Wilkes, ‘‘DataMesh — scope and objectives: a commentary’’, HPL-DSD-89-44, Hewlett-Packard Company (19 July 1989).

94. Stefano Zatti, ‘‘A Multivariable Information Scheme To Balance The Load in a Distributed System’’, UCB/CSD 85/234, Computer Science Division, EECS, University of California, Berkeley (May 1985).

Contents

Figures

’rs

Wisdom: The Foundation of a Scalable Parallel Operating System

Kevin A. Murray

Submitted for the degree of Doctor of Philosophy

University of York

Department of Computer Science

16th February 1990