The Intermediate Machine Approach
To Distributed Computing System Design

by

GORDON PETER SIMON

B.Sc., The University of British Columbia, 1981

A THESIS SUBMITTED IN PARTIAL FULFILLMENT

OF THE REQUIREMENTS FOR THE DEGREE OF

MASTERS OF SCIENCE

in

THE FACULTY OF GRADUATE STUDIES
Department of Computer Science

We accept this thesis as conforming
to the required standard

THE UNIVERSITY OF BRITISH COLUMBIA

April 1986

© Gordon Peter Simon, 1986
In presenting this thesis in partial fulfilment of the requirements for an advanced degree at the University of British Columbia, I agree that the Library shall make it freely available for reference and study. I further agree that permission for extensive copying of this thesis for scholarly purposes may be granted by the head of my department or by his or her representatives. It is understood that copying or publication of this thesis for financial gain shall not be allowed without my written permission.

Department of Computer Science

The University of British Columbia
2075 Wesbrook Place
Vancouver, Canada
V6T 1W5

Date Apr 30 /1986
Abstract

This thesis proposes that an intermediate machine be viewed as the software base of a distributed operating system. In this role the services provided to the operating system by the intermediate machine are similar to those provided by a security kernel. Such an intermediate machine software base differs from the traditional security kernel in that it additionally provides an interpreted instruction set. The advantage of this approach is that software extension of a machine architecture - whereby multi-processing and device abstraction are provided - is implemented in a homogeneous manner across all nodes of the distributed computer system.

The objective of the thesis is to determine the feasibility of an intermediate machine approach to distributed computer system design. Through investigation and experimentation, an intermediate machine based distributed computer system is developed and evaluated. This paper describes the system and its evaluation. As well, the merits of an intermediate machine software base are considered and the approach is contrasted with popular contemporary distributed system designs. Since it is expected that an interpretative intermediate machine would slow execution of system software, the focus of the project is in exploring this weakness in an effort to determine its extent. These efforts are to manifest in suggestions for a workable design of an intermediate machine based distributed computer.
# Table of Contents

Abstract ........................................... ii  
List of Figures ..................................... iv  
Acknowledgement ................................... v  

1.0 Introduction ................................... 1  
   1.1 Motivation ................................... 1  
   1.2 Objectives ................................. 3  
   1.3 Overview ................................. 4  

2.0 Background ................................... 6  
   2.1 Reference Architecture .................. 6  
   2.2 Intermediate Machine ..................... 9  
   2.3 Distributed Systems ....................... 11  

3.0 The Intermediate Machine Solution ............ 21  
   3.1 Approach ............................... 21  
   3.2 Design .................................. 24  
   3.3 Justification ............................. 26  

4.0 Description of the Implementation ............ 30  
   4.1 SLIM Distributed Computing System ........ 30  
   4.2 SLIM Machine ............................ 40  
   4.3 SLIM Distributed Operating System ....... 48  

5.0 Results and Evaluation ....................... 53  

6.0 Conclusions .................................. 64  

Bibliography .................................... 70
List of Figures

Figure 1. Distributed computing machine example .................. 3
Figure 2. The architecture of a layer .............................. 7
Figure 3. Computer System Reference Model ........................ 8
Figure 4. Richard's schematic diagram of a portable compiler .......... 10
Figure 5. Computer system configurations .......................... 12
Figure 6. A V domain of local network-connected machines .......... 15
Figure 7. Constrained application domain configuration ............ 18
Figure 8. Intermediate Machine based Computer System ............... 24
Figure 9. Architecture of the SLIM Distributed Computing System ...... 31
Figure 10. Organization of files on the SLIM bootstrap diskette .... 33
Figure 11. Process architecture of the SLIM Operating System ........ 35
Figure 12. Monitor output after machinestrap of SLIM ............... 36
Figure 13. Bootstrap sessions for nodes of the SLIM Distributed System 37
Figure 14. Multiple window output at node console .................. 39
Figure 15. Uni-process and multi-process architectures ............... 41
Figure 16. Architecture of the uni-process SLIM intermediate machine . 42
Figure 17. Simplified view of the multi-process SLIM architecture .... 47
Figure 18. Code module organization of the SLIM Distributed Operating System ......................................................... 49
Figure 19. Transport protocol between clients and servers ............ 51
Figure 20. Ideal configuration of an intermediate machine system .... 56
Figure 21. SLIM Distributed System configuration .................... 58
Acknowledgements

I thank my supervisor, Dr. Sam Chanson, for his advise and guidance on this thesis and Dr. Son Vuong for reading the final draft. As well, thanks to my former supervisor, Dr. John Peck, whose assistance during development of a multi-process SLIM intermediate machine is greatly appreciated. The support and encouragement of Dr. Paul Gilmore and Dr. Harvey Abramson are also gratefully acknowledged.

The financial support from Sydney Development Corporation is acknowledged and my thanks go to the people at Sydney Development Corporation for making a fellowship grant possible. Specifically I thank David Sutcliffe, Graeme Balcom and T.M. Williams.

Special thanks to my close friends Paul Okada, Chun Tam and Robert Eppich for their assistance. Paul parted with his computing equipment for many months in order to provide me with a distributed hardware environment. The most important acknowledgement goes to my family and my fiance, Caroline Cheng, whose encouragement and support will take a lifetime to return.
1.0 INTRODUCTION

This paper reflects the efforts of a study in Distributed Computing Systems Architecture. Of particular interest is the design of a distributed operating system across a heterogeneous network of processors and peripherals. The scope of this paper does not permit a complete and comprehensive treatment of all issues associated with development of such distributed computing systems; instead the author endeavors only to dissertate a technique for their design and implementation. The title of this thesis provides the reader some insight into the substance of the technique since its basis is the intermediate machine. The purpose in writing this report is to propose and defend details of the Intermediate Machine approach.

In the design of an operating system software developers clearly have identified the need for a software base. Most often it is to the kernel of the operating system that the developer has relegated this responsibility. Such a base provides primitive functions in response to the shortcomings of computer hardware. However, an obvious conflict stems from the function required of the kernel by the operating system and the contrary need for minimal complexity and size of this operating system base. Also this conflict is further magnified in a distributed environment. This thesis submits that the intermediate machine can offer an analogous service with some significant improvements as an alternative to the kernel in the role of a software base for a distributed operating system.

The design of a distributed computing system should begin with a determination on the functional requirements of the distributed operating environment. Identification of these requirements is followed by the development of specifications for a distributed operating system which implements the operating environment desired. Often the selection of hardware is performed
at this time as well. At this point, there is a difference between the traditional approach to distributed computing system design and the approach proposed in this thesis.

It is suggested before a developer begins the implementation of a distributed operating system that an intermediate machine be constructed on top of the hardware. In addition to providing the reliable, secure and functional software base required by an operating system, it furnishes an idealistic, machine independent, homogeneous machine interface. A distributed operating system is then implemented on this intermediate machine software base. Because of this interface the operating system is less impacted by the limitations resulting from constraints of hardware, heterogeneity, security and reliability.

Alternate approaches to distributed computer system design which differ from the approach described above in that none incorporate the intermediate machine, are reviewed in the second chapter of thesis. The systems considered, EDEN [Lazowska et al. 81], V-System [Cheriton 84], RIG [Feldman et al. 82] and LOCUS [Walker et al. 83], provide a variety of techniques for managing hardware heterogeneity. The EDEN system eliminates heterogeneity by explicitly limiting the hardware configuration to a set of homogeneous nodes. The V-System tries to establish homogeneity by providing a uniform kernel interface at all hardware nodes but full homogeneity is not possible when different processors are involved. Neither RIG nor LOCUS focus on issues of heterogeneity at or below the kernel but both attempt in different ways to provide a homogeneous services environment. RIG abstracts hardware to a logical level whereby a heterogeneous system is represented by a collection of logical resources managed by servers which communicate by means of uniform systemwide messaging. In LOCUS a heterogeneous network is abstracted by a systemwide distributed filesystem although the operating system recognizes a heterogeneous situation as such and, as well, implements typing when dealing with hardware constraints. For example, the operating system implements an instance of "/bin/who" for each different processor architecture in the network, although the filesystem provides the user with only a single filename entry. The context of the user's machine type is the mechanism provided by LOCUS to resolve this heterogeneity in a manner by which the user can remain unaware of the various load module entries.

The central theme in this work is the development of the intermediate machine as the reliable, secure and functional base of a distributed operating system. Although there exists literature on topics related to some aspects of this theme, few authors, if any, have exploited this view of the intermediate machine. Why and what, in reference to the research evincing this thesis, take form as
motivation and objectives in two sections of this chapter. The how or method of the approach procured through this research is the subject of the remaining chapters. An overview of the thesis is provided by the final section.

1.1 MOTIVATION

Imagine a Distributed Computing Machine comprised of several microprocessor based workstations networked to a mainframe or perhaps mini-computer with its associated quality printer and storage peripherals. The potential of this configuration is realized only when it supports a Distributed Operating System [Watson 81]. The heterogeneous nature of such a Distributed Computing Machine while enhancing the function and service provided by a Distributed Computing System presents development and implementation challenges not encountered in a homogeneous architecture.

![Figure 1. Distributed computing machine example.](image)

In this thesis, the "system" of interest henceforth, is a heterogeneous Distributed Computing Machine supporting a unified Distributed Operating System. An opportunity to contribute to research of such systems is the primary motivation of this work.
To a user accessing our system by way of local or perhaps remote workstation, the benefits of distributed computing may emerge in the form of: extended resource sharing, relative performance increase, extended communication, enhanced system integrity, reduced costs, and increased system availability. The power of this system lies in the developer's ability to provide these services through a consistent, global interface. This sentiment surfaces frequently across Distributed Operating System literature and research.

"LOCUS is a Unix compatible, distributed operating system in operational use at UCLA on a set of 17 VAX/750's connected by a standard Ethernet. The system supports a very high degree of network transparency, i.e. it makes the network of machines appear to users and programs as a single computer; machine boundaries are completely hidden during normal operation." [Walker et al. 83]

"The distributed systems considered in our project can be described loosely as organizations of highly autonomous information processing modules, called nodes, which cooperate in a manner that produces an image of a coherent system of a certain defined level." [Svobodova 79].

Attaining this global environment across a homogeneous hardware architecture is a much simpler problem than implementation across a heterogeneous hardware architecture. So much so that in some Distributed System implementations such as the EDEN system [Lazowska et al. 81], the heterogeneous issue is ignored. The intermediate machine approach, in the form presented by this thesis, reduces the heterogeneous problem to a homogeneous one.

The perhaps as yet unexploited relationship between the "intermediate machine" concept and distributed systems architecture provided further motivation for this thesis research. The work held an opportunity to apply "intermediate machine" technology to distributed system design and implementation. It is the intention of this thesis to motivate the reader to similarly employ the "intermediate machine" approach in further distributed system research and development.

1.2 OBJECTIVES

The main objective of the research was to investigate the feasibility for application of the intermediate machine in an approach to distributed system design. This objective is premised in the potential of the intermediate machine as the appropriate software foundation to a networked computing environment. A positive result will suggest that the intermediate machine contributes to the design of a Distributed Operating System implemented across a heterogeneous network.
A second objective, although in some ways a prerequisite to the first, was to determine the requirements or expectations imposed by a Distributed Operating System on an intermediate machine. This objective forces a functional specification of the intermediate machine and establishes its relationship with the hardware and operating system. A result of progress in these efforts may and, in fact, is expected to influence the purpose of an operating system kernel. Consequently, it may be necessary to accompany the resolution to this objective with some recommendations in respect to the function and form of a kernel.

With the usefulness of experimentation in mind an attempt was made to construct an experimental Distributed Computing System [Berg 82]. Under the constraints of a Master of Science thesis a complete study of the intermediate machine and its impact on the heterogeneous problem is not possible. Instead, the focus of attention is on the developments and difficulties manifested in the sample, albeit primitive, Distributed Computing System implemented for purposes of this thesis. Although theoretical justification of the intermediate machine as a solution to the heterogeneous problem is presented in this paper, its extensibility to a configuration more diverse than the described target hardware will be left largely to intuition and further research.

1.3 OVERVIEW

The first chapter of this thesis presented the motivation behind, the objectives for, as well as a brief overview of the Intermediate Machine approach to distributed computing system development. The second chapter outlines a background perspective of the research in this thesis and in this effort develops criteria by which it can be compared to similar work elsewhere. Chapter two also contributes a model of heterogeneity which, in reference to distributed computing systems, provides a means for comparison of this quality across distributed system architectures. Chapter three attempts a generic posture in a comprehensive description and justification of the intermediate machine solution. Chapter four which manifests the intermediate machine solution in the SLIM exemplar presents a description of the experimental distributed computing system implemented for this thesis. Significant results and an evaluation of them, in response to the objectives and criteria of earlier chapters are provided in chapter five. The last chapter, chapter six, summarizes and concludes this thesis.
2.0 BACKGROUND

The *Distributed Systems Architecture Model* offered by Watson [Watson 81] provides a useful framework for discussion of background and research material relating to this thesis. This chapter presents a rudimentary form of the model extended to describe heterogeneity of a computer system. A model of heterogeneity serves to emphasize the contributions of the Intermediate Machine approach and furthermore provides a mechanism by which the strengths of a distributed operating system, based on this approach, are highlighted when compared to a distributed operating system of another design.

The first part of this chapter describes the reference architecture employed in the definition of a heterogeneity model for computing systems. The model establishes a framework for further discussion in this and later chapters of the system architecture which results from application of the Intermediate Machine approach. The framework also proves useful in contrasting other distributed system implementations against one based on the intermediate machine. As well, a brief section designed to convey the contributions of the intermediate machine concept is presented. The final section, in light of this framework and other criteria, reviews several distributed systems.

2.1 REFERENCE ARCHITECTURE

Within the context of a reference model suitable for the purposes of this thesis, some informal definitions are necessary. A *node* is defined as an "autonomous information processing module" [Svobodova 79] providing objects and operations which can be performed on those objects. This definition extends to both hardware and software constructs since both a microprocessor and a software process - the executing entity created in software - are considered examples of nodes.
A *layer* is a level of abstraction implemented as hardware or software. It adds function to one or more physical nodes in an effort to provide services in the form of one or more logical nodes. Conceptually, the set of logical nodes and the set of physical nodes are disjoint since it is the purpose of the layer to provide modular independence and abstraction. If we arbitrarily select a layer and call it layer \( n \) then layer \( n + 1 \) is a reference to the client layer of layer \( n \). A layer may attempt to make the physical node completely transparent to its client layer (layer \( n + 1 \)) or simply provide an enhanced version of the physical node. A layer may multiplex a set of logical nodes on a single physical node or it may reduce a set of physical nodes to a single logical node. These various transformations depend on the objectives and function of the layer. Note that the logical nodes of a given layer \( n \) form the physical nodes of its client layer (layer \( n + 1 \)). A layer represents the architecture of a machine while nodes represent its interface.

The architecture of a computer system can be described as a collection or series of layer elements. Watson suggests a coarse resolution of four layers to generalize distributed systems in his "Distributed System Architecture Model" [Watson 81]. The four layers of this model are named as follows: "Applications", "Distributed Operating System Services", "Distributed Kernel", and "Hardware". The second and third layers are commonly combined as a single software component termed the "operating system". The superficial layering of Watson's model extends equally well to centralized systems; hence, the variation of his model shown in figure 3 will be used as a framework in this thesis to describe all computer systems.
The example computer system diagramed in figure 3 is representative of a multi-processing architecture. The "Kernel" layer multiplexes a number of software processes on a single processor host. The remaining layers of the operating system refine the primitive kernel services to a level necessary for a user process. An application program executes within this user process node providing an application environment, here termed the "application node".

![Computer System Reference Model](image)

Figure 3. Computer System Reference Model.

A network of nodes is a set of two or more communicating nodes in the same logical layer. Note that in our model, the nodes above and below layer n are transparent to nodes at layer n. Therefore, by definition, it is not possible for a node to directly communicate with other nodes at different layers. A distinction between a network and a network based service is made here. Since the architecture of a service or object is hidden from a node, the networking of the nodes which are part of a network based service, such as a distributed file system, is also hidden.

A set of nodes is strictly homogeneous when all nodes belonging to the set are identical. Two nodes are considered identical if they have corresponding identical objects and associated interfaces. This relationship must also hold true for the processing objects of each node. The term "processing object" refers to the processor component of the node such as a microprocessor at the hardware layer. When not all of the nodes in a set are identical the set will be termed heterogeneous.
As well, the term "strictly homogeneous" applies to a layer when all logical nodes associated with the layer are strictly homogeneous. Unfortunately, a layer is responsible only for the function and services addressed by it not for those addressed by layers above or below. This premise necessitates the term *functionally homogeneous* which will be used to describe a situation where the services addressed by a given layer are provided in a homogeneous form, while the services provided by lower layers are of a heterogeneous form. In other words, two nodes may provide the same function yet are not strictly homogeneous when their interfaces are different. Note also, that a layer can achieve strict homogeneity in spite of any lower layer heterogeneity by superseding all services provided by a lower layer.

### 2.2 INTERMEDIATE MACHINES

The node elements of our model are sometimes referred to as machines; the process nodes of layers above the hardware layer are termed *virtual machines* while the processor nodes of the hardware layer are termed physical or *real machines*. Here the term "virtual machine" is used in a generic sense rather than as a reference to a specific class of virtual machine used by the proponents of operating systems such as IBM's VM [Madnick and Donovan 74]. The layers of this model represent the implementation of the machines. Hardware components implement real machines and software components implement virtual machines.

A further categorization of the virtual machine results in the distinction between an *interpretative virtual machine* and a *native virtual machine*. Keeping in mind that a node or machine is an "autonomous information processing module", interpretative and native refer to the processing component of the virtual machine and how it relates to the processing component of the physical machine. If the processing component of the physical machine manifests itself at the virtual machine level, then the term native virtual machine applies. When a substitute processing component is implemented at the virtual machine level, the term interpretative virtual machine will be used. The substitute processor object of a given logical node, implements its instruction set onto the respective physical node through interpretative means. Often the substitute instruction set is more powerful than the native instruction set because the implementation layer can take advantage of all services provided by the physical node.
The logical nodes associated with the application layer are always interpretative virtual machines. The application program providing the interpretative virtual machine executes within the virtual machine of the services layer which most often is a native virtual machine. In fact, all logical nodes associated with the remaining layers of a computer system, excluding the hardware layer, usually are native virtual machine implementations. The logical node of the hardware layer is, of course, the physical machine.

![The structure of a portable compiler](image)

Figure 4. Richards' schematic diagram of a portable compiler.

The intermediate machine has its origins as a participant in the development of a portable compiler structure. The intent of the developers was to provide a mechanism by which the impact of machine dependence on a compiler could be minimized. Essentially, an intermediate machine is a form of interpretative virtual machine to be used as a target machine by the machine independent phases of a compiler. The final phase, the "Code Generator Phase", then translates the intermediate form to the native virtual machine form of the target system. A computer system may directly support the interpretative virtual machine by means of an interpreter, in which case there is no need for this final phase. Above, figure 4 shows the structure as suggested by M. Richards in an article on "Portable Compilers" [Richards 77].

The intermediate machine concept surfaced as far back as 1958 when a committee of SHARE proposed a solution to "The Problem of Programming Communication with Changing Machines" [Mock et al. 58]. Since that time in an effort to satisfy the requirements of a universal intermediate language, a number of abstract machine definitions have been developed including "Z Code" [Bourne et al. 75], "OCODE" and "INTCODE" [Richards 71], "Janus" [Coleman 74], "P-CODE" [Overgaard 80], "SLIM" [Peck 83, Fox 78], etc. A trend that developed during this research in
intermediate machine design was to consider the high level structure of its user, such as a "ALGOL" or "BCPL" program, and provide an architecture conducive to this structure. Thus, in some instances, a relationship develops whereby the intermediate machine becomes tailored to its client.

2.3 DISTRIBUTED SYSTEMS

For the purpose of presenting examples that provide suitable contrast with the system proposed by this thesis, several current distributed computer system implementations will be described in this section. The distributed systems considered: EDEN [Lazowska et al. 81], V [Cheriton 84], RIG [Feldman et al. 82], and LOCUS [Walker et al. 83]; represent recent research activity in this field. An attempt will be made to relate each system to one of the computer system configurations illustrated in figure 5. The configurations of figure 5 are intended to model the degree of homogeneity possible under the constraints of their implementation.

Firstly, figure 5 categorizes computer system implementations in terms of Federated Computer Systems and distributed Computer Systems. Each of the four configurations shown is an example of a computer network; however, a Distributed Computer System which is characterized by its Distributed Operating System is of a much more integrated structure than a Federated Computer System. The distinction is best stated by Deitel [Deitel 84], who uses the term "Network Operating System" to describe our Federated Computer System.

"In network operating systems, each of the network hosts runs its own non-network operating system. The networking is controlled by user programs that run on the various hosts. This approach is relatively straightforward to implement and has the advantage that it utilizes the software that already exists for the various. It has the disadvantage, however, that it lacks homogeneity.

In a distributed operating system, the individual operating systems of the hosts are discarded; a single homogeneous operating system is implemented for the entire network. This approach requires considerably more effort than a network operating system, but it offers conceptual integrity of design, an attribute that makes it much easier for designers and implementors to understand the network, maintain it, modify it, etc." [Deitel 84]

When the application layer of a computer network is the client of the network facility, a Federated Computer System is formed. A Distributed Computer System is formed when a layer that is a
Figure 5. Computer System Configurations.
member of the operating system is a client of the network facility. For either computer system, the
network object must be perpetuated, and perhaps enhanced functionally, by intervening layers to
the level of the client layer.

Secondly, in figure 5, the Distributed Computer System configurations are further
categorized by the level or degree of homogeneity associated with each implementation. A
heterogeneous collection of machines generally has different facilities available at each, or at least
many, of the machines. A set of logical nodes across all these machines are, by definition,
heterogeneous since the services associated with the nodes are different. However, if the collection
of machines are networked, and the facilities of each machine are made available to all of the logical
nodes, then the set, at least with respect to the services provided, is homogeneous. Unfortunately,
a layer providing such network based services is not responsible for the homogeneity of the logical
nodes supported by other layers. At best, a layer can supersede the services of a lower layer,
making the lower layer services transparent to the client of the superseding layer. Thus, a computer
system achieves a measurable degree of homogeneity depending on the number of layers providing
homogeneous service.

The RIG system [Feldman et al. 82], Rochester's Intelligent Gateway, is described as a
"multiple-machine, multiple-network distributed system". Its implementation is across a
heterogeneous collection of machines "interconnected by networks of varying characteristics".
The Services layer is implemented as a collection of servers managing the set of system resources
(or objects) for which it provides abstract representations, and operations to be performed on the
representations. The application layer is supplied with a systemwide, uniform messaging facility
through which communication with, and control of, network based services can be established.

"Access to resources is network transparent. A client can access a local
resource with the same semantics it uses to access a remote resource. The result
is an environment in which clients can communicate with servers without regard
for the topology of the distributed system as a whole." [Feldman et al. 82]

The kernel layer of RIG is as varied in architecture as its hardware environment. Many kernel
layer services are superseded by RIG servers. A host kernel supporting "the execution
environment (virtual machines) for processes" with basic message-passing, process scheduling,
memory management and device management services is expected to provide a complement of
servers which usually includes: a Process manager, Job manager, Name server, Logger, Time
server, Console handler and Network server. Therefore, a homogeneous kernel layer is not a
requirement of the RIG system, which "has been implemented as both a native operating system (Eclipses) and as guest-level extensions to existing operating systems (on the DEC-10, VAX, and Alto)."

Describing the RIG system in terms of the Distributed Computer System configurations of figure 5, RIG will be referred to as a "Services Level Functionally Homogeneous Distributed Operating System" (figure 5a). Both the Kernel layer and Services layer implement native virtual machines. Each distinct Hardware node results in at least one, though more when multi-processed, distinct Kernel node. When RIG supports a set of N distinct Hardware nodes, the result is a set of N distinct Kernel nodes. Unless the Services layer attempts to conceal the existence of all but one distinct group of Kernel nodes, and the RIG Services layer apparently does not, the set of nodes available at the Application layer will be heterogeneous. Also, note that for a process oriented distributed operating system such as RIG, there are more information processing nodes (processes) available to the Services layer than to the Application layer, since processes are consumed by the implementation of the Services layer itself.

Cheriton's V-System [Cheriton 84], similar to RIG in that it relies on a client-server relationship to establish the operating system environment, provides a "V domain of local network-connected machines". Essentially, a V domain is the set of logical nodes implemented by the Kernel layer through means of an interkernel protocol on top of LAN based workstation hardware.

"The V kernel is referred to as distributed because its facilities are available uniformly and transparently across multiple machines connected by a local network - that is, it provides the appearance of one kernel interface, for the most part successfully masking the existence of multiple machines. This is accomplished by having each machine run a separate, complete copy of the kernel. Remote operations are implemented by communication between different copies of the kernel using the interkernel protocol."

"The V kernel's design was guided in part by an analogy between its function and that of a hardware chassis and backplane. A standard backplane, such as the IEEE P796 bus or S-100 bus, provides a base for building and configuring hardware systems. The backplane provides power and communication and little else. Similarly, the V kernel provides processing power and communication in the form of processes and interprocess messages. In this analogy, a service is made available in a particular configuration by plugging in the software board that implements the desired service. The new service is then available through service-level protocols that use transport-level communications provided by the kernel." [Cheriton 84]
The V kernel, intended to be simple in its function, attempts to establish a homogeneous set of machine interfaces throughout the V system network. Although this limitation on function serves its purpose well, one might expect the kernel to lack efficacy in providing the support ordinarily expected of it. In addition to these minimal kernel and basic communication operations, the V kernel also provides some simple device control primitives. A homogeneous V kernel layer together with a high level language approach to server implementation offers the V-System developers a versatile method of distributed system implementation.

Again referring to figure 5, a distributed V-System is represented by the "Kernel Level Functionally Homogeneous Distributed Operating System" (figure 5b) configuration. This model suggests that the homogeneity of such systems is greater than systems where only Services level homogeneity exists. The V system still does not provide homogeneous nodes at the Kernel or Services layer when architecturally different hardware nodes are employed. Yet a greater quality of distribution is available through such a system, over one which does not attempt to provide homogeneity at both layers. Both the V-System and RIG by virtue of their kernel based server architectures, can minimize the impact of heterogeneity by confining applications to one group of identical processors and utilizing processors different from those in the group for the employment of the Services layer only. Figure 6 which is reproduced from V-System literature [Cheriton 84] shows how this is achieved in the V-system and, further in this chapter, figure 7 models this configuration.

![Figure 6](image_url)

Figure 6. A V domain of local network-connected machines.

The LOCUS system [Walker et al. 83], of special interest to this thesis for its intriguing solutions to problems of heterogeneity, is similar to RIG in that it attempts to establish homogeneity at the Services layer. LOCUS is different from RIG and V in that it does not implement a
Central to the design of LOCUS is a network-wide distributed filesystem which promotes a homogeneous view of programs and data. LOCUS depends heavily on its filesystem to establish distribution and control; whereas, RIG and V make use of servers and message passing for the same purposes. In response to the problems posed by heterogeneous machines, the filesystem supports strong typing of machines and files thereby offering transparency of machine boundaries and the network.

"There are several aspects to the hardware heterogeneity problem, such as number representation, byte ordering, instruction set differences and special hardware constraints (e.g. floating point availability). Strong file typing and conversions during transmission can help some of the problems.

In a LOCUS net containing both a DEC PDP-11/45's and DEC VAX 750's, a user would want to type the same command name on either type of machine and get a similar service. However the load modules of the programs providing that service could not be identical and would thus have to have different globally unique names. To get the proper load modules executed when the user types a command, then, requires using the context of which machine the user is executing on." [Walker et al. 83]

The solution, implemented in LOCUS, to this particular problem is a scheme involving a mechanism termed the "hidden directory". In an example given by Walker, the global representation of a command, for instance /bin/who, would refer to a hidden directory, which would contain a file entry (eg. 45 and vax) representing each load module needed in the network. During an effort to resolve a pathname search, when a process encounters a hidden directory, the user's context, which in this case is the machine type, is consulted to determine the selection from the file entries discovered. These steps are taken by the LOCUS operating system in a manner transparent to the user. When a user on the VAX machine invokes /bin/who, the module to be loaded would be taken from the "vax" entry of the hidden directory.

Because LOCUS effectively deals with system homogeneity at the Services layer, the requirements for homogeneity at the Kernel layer are less than one would expect of a distributed kernel. Although LOCUS is modeled by the "Services Level Functionally Homogeneous Distributed Operating System" (figure 5a) configuration, to the user it might appear as effective as a "Kernel Level Functionally Homogeneous Distributed Operating System" (figure 5b) configuration. Unfortunately, to the LOCUS implementor, the reality of its heterogeneity is very evident, since detailed attention must be expended on each unique machine in the network.
Additionally, the interrelationships between each unique machine must be predicted and addressed. Unfortunately, the same requirements cannot always be asked of application software development and in this respect the LOCUS design appears deficient. Finally, note that since LOCUS is a procedure based operating system, processes are not consumed by the Services layer. Each process provided by a LOCUS kernel achieves existence at the Services layer as well.

The Eden System is described by its architects as an "integrated distributed" computing system. Their objective was to attain distributed computing while maintaining an integrated environment providing "a high degree of sharing and cooperation among its users". In this respect and many others, EDEN is not unlike our other distributed systems; however, its distinction lies in its non-goals. Lazowska et al. [Lazowska et al. 81] state explicitly their intention not to address the heterogeneous problem in the EDEN project.

"We are not concerned with supporting a network of heterogeneous nodes, and with the concomitant problems of communicating abstract values among such nodes. Eden will support one basic node type (variations in configuration will of course be allowed), and 'foreign' machines will be interfaced to the system through such nodes. Eden users can invoke services on foreign machines through an 'object-like' interface, but the relationship will not be symmetric. Machine independence is an explicit goal in the design of the Eden kernel. This task, although challenging, is considerably less difficult than supporting heterogeneous nodes." [Lazowska et al. 81]

Instead, EDEN was implemented across a homogeneous network of "Intel iAPX 432" processor based "node machines" constructed for the project. The above quotation might also be considered a strong argument in favor of the research work of this thesis, since it is not always acceptable to limit a distributed computing system to homogeneous hardware.

The EDEN kernel is small and simple, providing a minimal number of primitives and relying on software outside of the kernel to provide most of the operating system services. The implementors of EDEN developed a four layer operating system software structure: the "EDEN kernel" layer, the "network o.s." layer, the "applications support" layer, and the "human interface" layer. The "human interface" layer provides program level operating system services such as editors and "language subsystems". Distributed services such as file system support are the concern of the "network o.s." and "application support" layers.

Since the physical machines of the EDEN system are assured to be homogeneous, the native virtual machines of the operating system are also homogeneous. Thus, the EDEN project can
loosely be associated with the "Strictly Homogeneous Distributed Computer System" (figure 5c) classification. Of course, each of the systems described so far can achieve a similar status simply by constraining, as EDEN does, the hardware environment to a homogeneous set of machines. This practice, however, is not always acceptable or applicable.

By now, the reader may have surmised that in this thesis homogeneity is considered a leading criterion by which to evaluate the merit of a distributed computing system design. Other applicable criteria are portability, extensibility, function and performance. Of these criteria, homogeneity is viewed by the author as the most relevant to current and future research of distributed computing systems, since its influence extends to issues such as process migration, host migration, and dynamic system configuration and reconfiguration. The UNIX operating system provides good testimony on the importance of software portability and extensibility as these are the qualities on which much of its wide acceptance is based. Function and performance, which often seem to be in conflict, are commonly accepted criteria of computing systems.

![Diagram of Distributed Computer System](Image)

Figure 7. Constrained application domain configuration.

The V-System and RIG, as a result of the client-server architecture, rank well under the stated criteria. The client-server paradigm, a popular model across distributed operating system implementations, appears to deal effectively with portability, extensibility, function and performance. As a consequence of its hardware limitation, the EDEN design appears to be less
adaptable than the V-System or RIG but likely, by integration of distributed control, achieves functional improvements over the others. In spite of implementing an inventive refinement of the Services layer, LOCUS presents essentially the same weaknesses as both RIG and the V-System. Unfortunately, the deficiencies of either approach results in limitations and complications for solutions to problems strongly influenced by homogeneity, such as the aforementioned process and host migration issues. For example, a software application under RIG is constrained to a subordinate domain, supported by alike processors within the total system. An application program coded for a particular machine in the network can migrate only to a similar machine in the network; consequently, a large mainframe is probably of little use to such efforts as load balancing when other machines, in the network such as a set of personal computers, are of a different architecture. Above, figure 7 oversimplifies this situation but suggests that machine nodes B and C provide system services only while machine node A must support all application software.

Perhaps the most appealing feature of the RIG or V-System design is extensibility. The addition of novel or duplicate hardware units is easily managed by the introduction or reproduction of their respective server processes. The administration of these additional hardware units usually presents little problem since, at the Services layer, the units manifest in a homogeneous form. Both the EDEN and LOCUS designs, although LOCUS perhaps less than EDEN, gain extensibility at a cost higher than that of the client-server architecture.

RIG offers to new machines the opportunity of full integration into the system simply by providing a "RIG-compatible kernel and MPP (message-passing protocol) as part of its own operating system or as process-level support services" [Feldman et al. 82]. In that this requirement is not as stringent in RIG as in some other distributed system implementations, notably the V-System, portability of software components is sometimes impeded. For all of these systems, the portability of software can be facilitated by the employment of high-level language programming. However, without adequate provisions for portability at both the Kernel and Services layers, software must often be immoderately modified during its introduction to new hardware.

The attainment of full functionality in most any distributed system will be impeded by the determents of partial homogeneity. Both the LOCUS and EDEN designs explicitly acknowledge the problems of heterogeneity, but neither, although LOCUS makes an inventive attempt, provide comprehensive solutions. Many sophisticated distributed operations, such as process and host migration, are difficult to implement across a heterogeneous environment within a V-System, LOCUS or RIG design. Ravindran points out a critical factor in such an endeavor, which in my
judgement relates to enormous implementation costs unless homogeneity can be established at a sufficiently low layer.

"Since process migration directly manifest in a dynamic movement of the locus execution from one site to another, the implementation of the process abstraction should be site-independent." [Ravindran and Chanson 86b]

EDEN, on the other hand, is able to pursue such expanded function at the expense of hardware diversification; however, the limitations of such a domain can impact system function negatively as well.

The emendation of a distributed environment from a heterogeneous to a homogeneous form, at best, represents minimal overhead. A bad software structure may amplify the impact of this overhead to a point where performance of the system is severely hindered. The EDEN system, of course, avoids this overhead altogether, but for RIG, LOCUS and the V-System, the method by which this emendation is implemented will determine the influence of heterogeneity on system performance. In a message-based multi-process architecture, such as RIG or the V-System, a major factor of performance still is communication efficiency, and, at present, this factor overshadows most other performance issues. Eventually, as improvements in this regard are made, one should expect heterogeneity factors in performance to be the focus of more attention. A further consideration in performance pertains to the layer at which heterogeneity is addressed, as the lowest layer provides less opportunity for such overhead to propagate to other layers and, consequently, grow larger than necessary.

The definition of a crude, yet workable, model of computer system homogeneity, is offered in the first section of this chapter. Although not complete and certainly as with most any model, not universally applicable, it serves as a basis for discussion of much of the material in this thesis. A brief overview of the intermediate machine followed the first section in an attempt to provide the reader with an interest for chapter three. Also, this chapter has described several of the popular distributed computing system implementations and, in a review of each one, the notable design issues in relation to the approach proposed by this thesis were detailed. Criteria suitable for the formal evaluation of a distributed system's design were developed and then related to these implementations in preparation for chapter five. Restated, the criteria proffered by this chapter is homogeneity, extensibility, portability, function and performance.
3.0 THE INTERMEDIATE MACHINE SOLUTION

To date, the intermediate machine has not played a role in the development of distributed computer systems. In existing distributed implementations, the problems resulting from the combination of heterogeneous hardware are addressed by a variety of solutions, yet in many instances these solutions are not as comprehensive a response to the many issues in distributed computing as the intermediate machine. The intermediate machine solution should be viewed as a foundation for the development of a distributed computer system capable of complex operation in a heterogeneous environment. This chapter describes the architecture of a distributed computing system whose design is based on the Intermediate Machine approach.

3.1 APPROACH

Modern software techniques, in an attempt to devise a structure suitable for the objectives of a computer system, encourage refinement of the hardware through the implementation of a supportive software base.

"With respect to operating systems, a number of efforts were made to develop a low level base that would provide basic system functions in a highly reliable way. On top of this skeletal software base, extensive operating system functions and user supports would be built." [Popek and Kline 79]

The evolution of this concept into what is known as the "security kernel" has helped to resolve the purpose and definition of such a base. A software base engineered to the specifications of a security kernel must meet the paramount design criteria prescribed by demands of reliable security, though frequently functional requirements are limited as a result. This conflict is recognized as one of the few weaknesses of the security kernel strategy. Popek further argues that costs of this concession
"can generally be minimized"; however, in a distributed environment, the consequence of such penalties is expected to be more severe.

"There are several considerations that make a kernel based operating system architecture significantly different from a more traditional architecture developed without security as a major design criterion. Reliable security demands that the software upon which security depends be as small and simple as practical. As a result, functions included in operating system bases to enhance performance or increase the convenience of writing software above that base are not needed for security purposes and consequently not included in a kernel. Kernel designs would typically exclude that software despite potentially increased performance overhead or greater non-kernel software complexity." [Popek and Kline 79]

The purpose of an operating system base, succinctly described in reference to the V-System distributed kernel, is to provide "processing power and communications in the form of processes and interprocess messages." [Cheriton 84]. Another example furnishing guidelines for this purpose is the VEREX kernel, whereof is detailed the objectives intended for its base in the following [Lockhart 79].

More specifically, the kernel provides:
1. many inexpensive deterministic processes;
2. interprocess communication via small synchronous messages, and a data transfer facility.
3. operations for process management, which includes: dynamic process creation and destruction, flexible processor scheduling, memory management, and exception handling;
4. a simple system initialization mechanism;
5. device abstraction, including: limited access, controlled DMA, and buffering for efficient handling of rapid interrupts.

As well, the consensus on functions to be excluded from a kernel implementation of the operating system base include a file system and high-level device support. Of course, a precise and universally accepted definition of the goals and form of a software base does not exist, yet agreement on principles of its architecture is maturing.

As evidenced by the examples in this thesis, as well as many others not cited, the conventional method of implementing a software base is in the form of an operating system kernel. A consequence of this approach is the conflict that arises in the dual purpose assumed by the kernel. Such a kernel must oblige, not only, the functional requirements of an operating system, but the further constraints of a software base as well. The security kernel illustrates best, the contrast between these goals since both are realized, though often compromised, in its design.
Also, in support of my claim that this conflict is aggravated in a distributed heterogeneous environment, I will reproduce a list of problems, prepared by Watson [Watson 81], some of which impose new responsibilities on the kernel. These problems result from both heterogeneity and distribution since they are "due to the physical separation and heterogeneity of their components and multiple controlling administrations" [Watson 81]

- Naming (identification), protection, and sharing issues resulting from the heterogeneous systems.
- Translation issues resulting from heterogeneous control and data encodings and representations.
- The need for a message based interprocess communication (IPC) model because memory sharing is not practical between distributed components.
- Distributed service and resource structures.
- New sources of errors and potentially more complex error-recovery.
- Need to maintain consistency among multiple copies of information.
- Synchronization and other control problems resulting from distributed state information and arbitrary message delay.
- Accounting and resource management issues resulting from multiple controlling administrations.
- Efficiency problems resulting from the variable communication delay and bandwidth of between components.
- More complex verification, debugging, measurement problems.

As well, sophisticated distribution controls, such as the "process alias" described by Ravindran and Chanson [Ravindran and Chanson 86a], require implementation in or significant support from the distributed kernel.

Now, consider the intermediate machine in an alternate strategy to the implementation of a software base. If one accepts the premise that the operating system kernel, though perhaps a natural choice, is not the best of the options available for implementing a software base, then the opportunity of the intermediate machine as its replacement must surely warrant attention. This approach, in addition to fulfilling the need for a software base, liberates a developer from having to assign such responsibilities to the operating system kernel and permits unrestricted application of the kernel to distribution issues in the implementation of a distributed computing system.

At first, the introduction of the intermediate machine into the design of a distributed computing system may not seem to be a palatable endeavor. However, in support of this approach, and as a suggestion that it is little different from a technique employing, for a similar purpose, a security kernel, the parallel between this form of software base and the intermediate machine should be pointed out. In common application, a security kernel delivers hardware extension as needed for
multi-process support, device abstraction, and a minimal set of primitive functions, as well as, interprocess communication in some specific instances; these same services are the intended responsibilities of the intermediate machine as well. In this fact, the resemblance of the security kernel to the intermediate machine is revealed and perhaps suggests that such services more naturally associate with the intermediate machine.

3.2 DESIGN

The easiest method of describing the Intermediate Machine approach is to present it in terms of the Computer System Reference Model discussed in the previous chapter. Essentially, the alternative computer system architecture developed by means of the Intermediate Machine approach is much like the earlier described architecture except for the addition of a new layer. The new layer, shown below in figure 8, is called the "Intermediate Machine" layer and is positioned immediately above the Hardware layer.

A considered choice was made not to include the Intermediate Machine layer as part of the Operating System component but instead view the first operating system layer as its client. The advantage of this view is that it suggests independence of the Intermediate Machine layer from the
operating system thus promoting a relationship where an Intermediate Machine may support a number of operating system designs or an operating system supported by a variety of intermediate machines. This advantage would be realized as improved portability across the physical machines and operating systems supporting an intermediate machine layer.

The Intermediate Machine layer implements an interpreter for the emulation of an abstract machine architecture. This interpreter provides a uniform interface for hardware components in an effort to abstract and obscure all hardware dependencies to the benefit of the kernel and supervenient layers. Thus, this new layer provides a hardware independent environment to the operating system and serves as an effective mechanism for dealing with the heterogeneity of computer networks. The Intermediate Machine layer presents a homogeneous network of intermediate machine nodes despite the reality of the physical computer network.

The physical nodes of the Intermediate Machine layer are the Real Machines created in hardware. Its set of logical nodes is a collection of Interpretative Virtual Machines which are multiplexed - or timeshared - on these Real Machines. The logical nodes of the Kernel and Services are Supportive Virtual Machines as they are native only with respect to the Interpretative Virtual Machines of the Intermediate Machine layer. An Application layer logical node is an Interpretative Virtual Machine.

The purpose intended for the Intermediate Machine layer, in conjunction with the assimilation of heterogeneous hardware, is that of hardware extension and abstraction, a responsibility traditionally assumed by the Kernel layer of an Operating System. Although the function definition of a kernel is unique to each operating system, hardware extension seems a common element among them.

"The kernel extends the base machine architecture to provide an environment suitable for supporting message-based, multi-process structured operating systems." [Lockhart 79]

"The kernel is a primitive monitor that implements objects and indivisible operations on these objects as an extension of the basic machine architecture." [Cheriton 79]

"Essentially the monitor is a software extension of the hardware structure, which makes the RC4000 more attractive for multi-programming." [Brinch Hansen 69]
The objective of this hardware extension is to provide an environment conducive to the management of software processes. A need for hardware extension exists because of the general employment intended for computer hardware. Consequently, the first steps in adapting computer hardware to a particular application takes the form of hardware extension. The Intermediate Machine abstracts hardware to a functional level much more acceptable to the layers of the operating system than naked hardware.

At first, the reader may notice a similarity between the architecture of an Intermediate Machine based computer system and IBM's VM/370 system [Madnick and Donovan 74]. However, there are some significant differences. The major difference is that the Intermediate Machine approach implements an abstract machine, unlike VM/370 which "makes a single real machine appear to be several real machines" [Deitel 84]. In other words, the virtual machines of VM/370 are Native Virtual Machines, while the Intermediate Machine Approach implements Interpretative Virtual Machines. Another difference is that a VM/370 virtual machine is a complete machine as it provides access to all real devices provided in hardware. The Intermediate Machine approach, as proposed in this thesis, does not require all virtual machines to be complete in their access to devices.

3.3 JUSTIFICATION

The costs of an Intermediate Machine interpreter are significant and must be measured carefully against its advantage in an effort to justify the Intermediate Machine approach. It is important to recognize that the administration of a heterogeneous environment introduces overhead regardless of the method. In previous chapters we have seen current distributed implementations deal with heterogeneity in various ways and at various layers; however, one is prompted to identify the layer at which optimal performance of a heterogeneous computer network can be achieved. In this thesis it is suggested that dealing with heterogeneity through the introduction of an Intermediate Machine layer, in terms of function against cost, is the optimal solution.

Perhaps by this point overstated, but still one of the more important contributions of this thesis, is that the Intermediate Machine approach provides an opportunity to reduce the heterogeneity of a computer network to a more workable, homogeneous environment. The purpose of a homogeneous environment becomes obvious when one considers issues of host and process
migration. This aspect of the approach alone warrants its research and its benefits are realized by both developer and user of the distributed computer system.

The looming concern and disadvantage is processor performance loss. Processor performance loss in an Intermediate Machine layer is due mainly to the cost of instruction interpretation as many machine cycles are lost through the overhead of decoding each program operation. At best, one could only hope to equal the performance of the bare hardware. Consequently, in a centralized, single processor system the introduction of an interpretative Intermediate Machine layer is frequently discouraged. Only the promise of software portability has enticed software developers to attempt incorporating an Intermediate Machine layer into the design of such a centralized system, the UCSD P-System being a notable example.

It is recognized that a process executing on an intervening interpreter, based on a single processor, will perform poorly as compared to a similar process executing directly on the same hardware. The performance loss is exaggerated when a number of processes, multiplexed onto the same hardware, are considered. However, an opportunity exists to impact this performance loss in a distributed system where a number of processor resources are available to these same processes. If the processor resources are under utilized, they can be made available to these other processes. Furthermore, with the costs of processors currently representing only a small portion of total distributed computer expenditures, a large number of under utilization processors can easily be provided. When many processes are distributed over a number of interpreter resources, the total execution time might even be less than the execution time of similar processes running directly on a single multi-programmed processor.

In addition to sharing under utilized processors, and of equal benefit to a user, is the relative performance increase that can be achieved in a heterogeneous distributed computer system. A user whose processing power is limited to a microprocessor based work station would be advantaged considerably by access to the processor of a powerful mainframe computer in the network. For a programmer, the compilation of a source file, or some similar task, might be expedited if the job could be performed at the mainframe site. Furthermore, in this example, it is desirable that the result of the compilation job, the object code, be executable at any site in the network, or of special interest to the programmer is that the code be executable, at least, at both the mainframe site and the microprocessor based workstation. This relative performance increase, coupled with the performance improvement that can sometimes be achieved when low utilization of processors is possible, counters to some extent the performance loss due to interpretation.
A further advance made possible by the intermediate machine relates to an improvement in system function. For example, as sophistication of distributed computer systems is realized, process migration and host migration operations will form an integral part of distributed computing. Process migration involves physically moving the code and data space of a process from its current processor site to a different processor site as well as duplicating its environment at this new site, but the migration of the process is usually transparent to the user and other processes interacting with the migrating process. Host migration is an operation similar to process migration in that it may be made up of a sequence of process migrations, but involves the relocation or duplication of a machine node to another site in the network.

"Host migration becomes a requirement when machines are to be taken in and out of a network for operational reasons transparent to other hosts in the network. We consider a simple scenario to illustrate the feature. Let H1, H2, .., Hi, .., Hn be hosts on the network, and suppose Hi is to be taken out of the network for maintenance purposes. A new machine Hi' (identical to Hi) has to be introduced into the network, and all executions on Hi are moved into Hi'; then Hi is shutdown and removed." [Ravindran and Chanson 86b]

Whereas, in a heterogeneous environment, the implementation of process migration and host migration operations are likely to be impossible; by the Intermediate Machine approach, since all nodes in the network appear to be an instance of the same intermediate machine regardless of any heterogeneity presented by the Hardware layer, their implementation appears more feasible.

An ideal programming environment liberates the software developer from concerns which are not a part of the application problem. In other words, when possible, issues of machine dependence, hardware deficiency, and software portability do not manifest in an ideal programming environment. The intermediate machine in resolving some of these issues contributes towards the realization of such an ideal programming environment. Machine dependencies for instance, in the software of an operating system, often require attention at each of the many layers, but the intermediate machine centralizes these machine dependencies in one layer by providing a focus for such efforts. Software portability, the value of which is appreciated less in a centralized environment than in a distributed one, can be realized at either the source level or the object level. Source portability is made possible with limited success through the use of a portable high-level language, but across heterogeneous machines, object portability is possible only by means of some such vehicle as the intermediate machine. Presently, however, there are few if any computer system implementations capable of object code portability although its application is evident in process and host migration.
Those readers familiar with the concepts of microprogramming and user microprogrammable machines [Agrawala and Tomlinson 76] might notice the analogy between microcoding a real machine architecture on a user microprogrammable machine and microcoding an intermediate machine on a microprocessor or mainframe processor. The difference in this analogy is that the target of the intermediate machine, a microprocessor or mainframe processor, is much more sophisticated than the user microprogrammable machines in use; hence, the microcoding of an intermediate machine may be considered an elevation of the microprogramming concept. The performance and function that may be expected from an intermediate machine is much greater than usually expected from a real machine because the primitive operations available from the processor are superior to the operations available on a user microprogrammable machine. Furthermore, this new higher level of microprogramming which supports machine independent portable efficient object code improves the prospects of object code portability.

In providing a uniform interface to hardware components of a computer network, the intermediate machine establishes an ideal, homogeneous network representation or model, on which, independent of any future variability in the hardware understructure, operating system software can be developed. The intermediate machine, under such circumstances, serves as a buffer between hardware and software. The intermediate machine also facilitates software extension in the event that new hardware is to be added to the network. Without having to make any changes to software layers above the intermediate machine, a developer can accommodate a new, and perhaps architecturally different, hardware node simply by constructing for it the required intermediate machine. Potentially, this technique, which was the basis of the original interest in intermediate machine technology with respect to compiler software portability, could minimize the time and effort involved in transporting and extending a distributed operating system. As a buffer, the intermediate machine provides the software developer with an effective method for contending with inadequacy and volatility of many hardware products.

The balance of advantages against disadvantages ultimately justify, or at least determine a value for, the design of a computer system. The processor performance loss is identified as the most significant disadvantage of the intermediate machine approach. On the scale, in favor of the approach is offered improved function, relative performance increase, balanced processor utilization, opportunities for software portability and extensibility, as well as hardware extension in support of a multi-process architecture. Because the intermediate machine would serve the purpose of a software base at a cost not much greater than existing forms of a software base, the scale is already biased towards acceptance of the intermediate machine approach.
4.0 DESCRIPTION OF THE IMPLEMENTATION

Perhaps, from the reader's perspective, the practicability of the Intermediate Machine approach is still in question. As a thesis project, an attempt was made to develop an experimental Distributed Computer System using the Intermediate Machine approach. The feasibility of implementing such a Distributed Computer System is the topic of this chapter.

The abstract machine model chosen for the role of intermediate machine in this thesis project has been named the "SLIM machine" [Fox 79]. With reference to this model, the "SLIM Operating System" is, therefore, the name given to the Distributed Operating System that has evolved from the thesis research.

4.1 SLIM DISTRIBUTED COMPUTING SYSTEM

The selection of computer hardware, intermediate machine and distributed operating system are prerequisite to the development of an intermediate machine based Distributed Computing System. The choices that determine the structure of the SLIM Distributed Computing System are presented in this section. Economic factors nearly always guided the selection where choices were available.

Figure 9 helps to visualize the structure of the SLIM Distributed Computing System. The set of physical machines, interconnected by network technology, form the distributed domain. Every physical machine in this domain is made to look like a SLIM machine by the introduction of an Intermediate Machine layer. Thus, a heterogeneous network of physical machines takes on the appearance of a homogeneous network of SLIM nodes. This network of identical SLIM nodes provides the operating system with a simplified view (subsequently called the "Network Model") of
the hardware domain. Hardware extension, to a level necessary for support of client and server processes, also is performed by the Intermediate Machine layer, thus, as illustrated, operating system processes execute directly on each network node.

![SLIM Distributed Computer System](image)

**Figure 9.** Architecture of the SLIM Distributed Computing System.

The hardware base is a networked collection of IBM Personal Computers. Minimally, each hardware node consists of 128Kb RAM, at least a single floppy disk drive, an Arcnet (Datapoint Corporation LAN specification) local area network interface, and, of course, an Intel 8088 microprocessor. Also, each IBM Personal Computer supplies a video display screen (monitor) and keyboard, which are needed when a hardware node is to serve as a workstation. Consequently, all hardware nodes in the computer network configured for the thesis project were directly available to users as workstations. In an alternate configuration some nodes might provide dedicated services such as printing and, therefore, would not be available as workstations.

Although the design of the SLIM Distributed Computing System is intended for a heterogeneous network, as evidenced above by the description of the hardware layer, its implementation is currently limited to a homogeneous one. The reasons for this limitation are both technical and economic. The cost of acquiring additional machines architecturally different from the IBM PC's and then networking them to this IBM PC network configuration were beyond the means of the research.
The selection of an abstract machine model with application as the Intermediate Machine in a Distributed Computer System is guided primarily by the structure of the Distributed Operating System. For this reason, a determination on the form of Distributed Operating System is a postulate in the preference of the abstract machine. Although any one of the many existing Distributed Operating Systems would qualify as a candidate in such a determination, an implementation of this magnitude, within the constraints of this thesis, was not workable. Instead, a rudimentary Distributed Operating System was designed anew with intent simply to exercise the Intermediate Machine approach. The structure of this Distributed Operating System, earlier referred to as the SLIM Operating System (SOS), is similar to that of RIG [Feldman et al. 82], Verex [Cheriton 79, Lockhart 79] and its distributed relative, the V-System [Cheriton 84]. Lockhart provides us with a remark that describes this structure; "A central idea is that of structuring systems as a kernel and a set of cooperating processes . . . ". A more detailed description is provided by Feldman et al.

"The world consists of a collection of resources (or objects), accessible by clients, and managed by servers. A server defines the abstract representation of its resource(s) and the operations on this representation. A resource can only be accessed or manipulated through its server. Because servers are constructed with well-defined interfaces, the implementation details of a resource are of concern only to its server." [Feldman et al. 82]

With the general architecture of the Distributed Operating System established, further decisions on the intermediate machine architecture can be made.

The criteria for selection of an intermediate machine architecture are currently informal and largely intuitive. The choice in this thesis was the SLIM machine model. This choice is based more on SLIM's simplicity and general application rather than any of its specific complexities. The origins of the SLIM machine are in an M. Sc. thesis by Mark C. Fox [Fox 78]. His work laid a foundation for the architecture of the present day SLIM machine. Further work on the SLIM design by Dr. J. Peck et al. [Peck 83] refined and extended Fox's machine leaving us with a comprehensive uni-process architecture. To better meet the requirements of an operating system through the efforts of this thesis research, the uni-process architecture was modified into a multi-process architecture.

The software of both the Intermediate Machine layer and the SLIM Operating System are provided on floppy diskette. One can initiate an instance of the SLIM Distributed Computing System by ensuring placement of this floppy diskette into the bootstrap floppy drive of an IBM Personal Computer during its power up. The SLIM System manifests an existence by first
bootstrapping the Intermediate Machine layer onto the machine and then bootstrapping the SLIM Distributed Operating System onto this initial node. To distinguish the two phases of bootstrap, the first phase, the bootstrapping of the Intermediate Machine layer, is named the "Machinestrap". The second phase, the bootstrapping of the operating system, retains the usual name of "Bootstrap".

The machinestrap is a small segment of code that resides on the first sector of the boot floppy diskette - normally where the operating system bootstrap lives - and is loaded into ram memory by the IBM PC firmware. The first function it performs is to initialize and prepare the IBM PC for installation of intermediate machine software. The integrity of the system diskette is checked and, if all is in order, the intermediate machine software, which is separated into an instruction interpreter module and a BIOS device administration module, is loaded and activated. Failure of the machinestrap to install the intermediate machine results in an error message reported to the screen. Its correct and complete execution results in the IBM PC effectively being transformed into a SLIM node (machine). The remaining nodes of the SLIM Distributed Computing System join the network at their convenience and, in fact, the intention is for the joining and exiting of machines to take place dynamically.

![Figure 10. Organization of files on the SLIM bootstrap diskette.](image)

As a visual aid to better understand the initialization of the SLIM system, figure 10 depicts the files involved in the bootstrap sequence and their arrangement on the floppy diskette. Since
IBM PC DOS served as the program development system in the construction of SLIM software, it proved advantageous to maintain a compatibility, which accounts for the above file layout, between the SLIM file system and the PC DOS file system [Microsoft 82]. The machinestrap code segment (MSTP.BIN), as well as both the instruction interpreter (SLIM.BIN) and the SLIM BIOS (SBIO.BIN), are "microcoded", (note that this is not the traditional sense of the word) for the target machine. The "BIN" suffix appended to each of these files denotes the contents as a binary image for the Intel 8088 processor. In contrast, the SLIM operating system bootstrap (BOOT.I) and any other SLIM operating system software is coded for the intermediate machine. The "I" suffix denotes an interpreter image for the SLIM intermediate machine. At present, the simple machinestrap program requires the files, SLIM.BIN, SBIO.BIN, and BOOT.I, to be of limited size and at specific locations on the floppy diskette, although, at least for the SOS bootstrap file, it should be modified to allow for arbitrary size and location.

The machinestrap (MSTP.BIN), the SLIM interpreter (SLIM.BIN) and the SLIM BIOS (SBIO.BIN) constitute the intermediate machine software. The SLIM.BIN code segment, which is less than 1 KBytes in size, as well as the SBIO.BIN code segment, currently under 5 KBytes, both are loaded permanently into the target processor's main memory (RAM), while the machinestrap (MSTP.BIN) is loaded for execution purposes only. In the IBM Personal Computer, when loaded, this software is distributed sparsely across the first 10 KBytes of memory.

The SLIM Operating System bootstrap and indeed all the SLIM Operating System software, were programmed in the SLIM assembler language. For the purpose of assembling this software to SLIM object code, a SLIM assembler, written in the C language for PC DOS, had to be developed. Although a compiler able to generate SLIM code for the BCPL language [Peck 83] exists, it had not been transported to the PC DOS development system. Hind sight suggests that first transporting a BCPL/SLIM compiler to PC DOS may have reduced efforts in construction of the SLIM Operating System; however, as such work alone constitutes a M.Sc. thesis [Hayter 83], programming facilities had to be limited to the SLIM assembler.

Since the machinestrap performs its duties in a manner unnoticed by the user, bootstrap initialization is no more complicated for the SLIM Operating System than for any other operating system. As hardware initialization and preparation are performed by the intermediate machine layer, the main responsibility left to the bootstrap program is one of loading all requisite process essential to the correct operation of the operating system. This set of process includes most resource and management servers, as well as certain system clients, as depicted below in figure 11.
The collection of processes in figure 11 represents, at best, a minimal configuration and provides only enough services as necessary for loading of other system or application processes. The processes labeled "Filer 1" and "Filer 2" are resource servers providing file support. The combination of a keyboard device and a monitor device represents the resource termed a "console", for which two servers are shown: "Console 1" and "Console 2". The current version of the console server provides rudimentary windowing facilities which makes it possible for a number of client processes to share a single console server. The loader servers, "Loader 1", "Loader 2", and "Loader 3", operate the processor resource of a machine node whereby a remote or local invocation of a process can be serviced.

In that a resource server is anchored to a machine node providing a particular resource, they differ from management servers and system clients. Of the three remaining processes shown in figure 11, two are system clients, and one is a management server. The management server termed the "Operator" is essentially a name server providing a service whereby the physical location or address of a server, and related relevant information, can be associated with its name. Although the Operator can reside at any machine node in the system, the current implementation of the SLIM Operating system requires that it be the first process loaded. As subsequent processes are loaded the Operator is informed and updated. Of course, a practical operating system should offer a
number of system clients, such as text editors and language compilers but, at present, only one system client is available on the SLIM system. The system client referred to can be described as a user shell, of which two - "Shell A" and "Shell B" - are shown above. Each shell provides a user with access to system facilities and, as well, interacts with a loader server on behalf of the user in executing other system clients or application programs.

![Figure 12. Monitor output after machinestrap of SLIM.](image)

A scenario of a typical user session on the SLIM Distributed System concludes this section. We begin by inserting a SLIM system diskette into the boot drive of a machine with qualifications as the initial node of the network. As the machinestrap completes the installation of SLIM software, the console monitor of the machine produces output as shown above in figure 12. This output provides identification and copyright information.

The SLIM Operating System bootstrap makes its presence known by prompting a question mark ('?') below the copyright. Although much of the bootstrap phase can and should be automated, at this point in the system's development, assistance from the user is required. To this end, at the initial node of the system, the user responds to the question mark prompt with the name of the first SLIM process to load. The current design of the SLIM Operating System requires a name server, called the operator, to be the first executing process. Therefore, as shown in diagram
a) SOS Bootstrap at machine 1

b) SOS Bootstrap at machine 2

Figure 13. Bootstrap sessions for nodes of the SLIM Distributed System.
'a' of figure 13, the user enters 'OPERATOR.I" at the machine's console keyboard. The bootstrap then provides the user an opportunity to further load processes essential to the minimal operation of the system. The prompting of the symbol ('#'), to which the first response is a request for the loading of a filer process, reflects this opportunity. Note that typically, a system will support more than one instance of the filer program and, therefore, a mechanism to distinguish multiple filer process from one another is necessary. Hence, in the response, the name of the filer program is followed by a semicolon (';') and then by the user defined name for this instance, which in the example is "FILER1". Additionally, an instance of the loader program, which was named "LOADER1", was also requested in the example of figure 13a. The loading sequence and the bootstrap phase itself is terminated when the user's response is a period symbol ('.').

Also shown in figure 13, diagram b, is the bootstrapping of a second machine in the network. The machinestrap phase of the two machines is identical; however the response to the bootstrap prompts of machine 2 are different from those of machine 1. At the second machine, a user's response to the 'question mark' prompt is simply a null entry ('<CR>'), which leads the bootstrap to assume the existence of the operator. In the example, the processing resource of machine 2 is to be made available to the system; hence, an instance of the loader program is loaded and, in an attempt to distinguish from it the first loader server, is named "LOADER2". As well, a second filer, "FILER2", is also loaded. The next program loaded at machine 2, a console server, is named "CONSOLE1". The shell, which is loaded after the console server, interacts with CONSOLE1 as the user's representative in the system. The console definition parameter, "CONSOLE=CONSOLE1", designates the console server for "SHELLLA". Unfortunately, the example exaggerates the system's true capabilities since, presently, modifications to both the shell and loader programs are necessary for implementation of full parameter control. Currently, an actual demonstration requires the console definition to be implemented in the shell code. Again the bootstrap is terminated by a period symbol.

Although upon request, the bootstrap loads each program immediately, in order to avoid a conflict for the console resource, the execution of each process is delayed until the loading sequence is terminated. Therefore, the final task of the bootstrap is to "originate" each process: an operation whereby a process is authorized for execution. In the scenario described, the floppy diskette drive and the console monitor display activity as their respective servers begin execution.

When the shell process requests, from the console server, the service of an open operation, a double border window is created on the monitor screen. The open operation is followed by a write
request which results in an identification message and prompt being output to the screen. The large window in figure 14 below shows this activity, although the window at the time of this snapshot was no longer active and, consequently, is represented by a single border only. The smaller, double border window represents a second incarnation of the shell. The "SET" command, examples of which are shown in both shell windows, is the mechanism by which a user can select a choice of server facility. The sequence of three commands in the larger shell window form an effort to load the second shell. The two "SET" commands designate "FILER1" and "LOADER2" as the active filer and loader servers, respectively, for the duration of any subsequent commands. The third command is a request to load the shell program, under the name "SHELLB", with a console definition parameter designating "CONSOLE1" as the console server for the shell. Shortly after the return key is entered for this command, the double border window of the large shell changes to a single border and the second shell window appears which in figure 14, is represented by the upper right window. The identification and prompting appears in this new window which leaves the cursor marking an input opportunity from the second shell.

SOS - SLIM OPERATING SYSTEM V1.0

```
# SET FILER=FILER1
# SET LOADER=LOADER2
# SHELL.B;SHELLB CONSOLE=CONSOLE1

SOS - SLIM OPERATING SYSTEM V1.0

```

GS LINEAR PROGRESSION V1.0

```
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18
```

Figure 14. Multiple window output at node console.

A user may again select the first shell window or, as shown in figure 14, proceed with operations on the second shell. Presently, function keys on the console keyboard allow a user to select a window; however, a mouse or similar pointing device might be the device actually preferred for such interaction. In the second window, "FILER2" was set as the filer server and a
program called "COUNT" was loaded. The console server, in response to activity by the count program, opens the third window and displays the output shown in figure 14. As the count program continues to execute, its output is added to the display. In the example of figure 14, however, the second shell window is again made active, the result of which is the suspension of "COUNT" output and a further opportunity for interaction with "SHELLB".

4.2 SLIM MACHINE

The SLIM machine furnishes a model for the specification of the Intermediate Machine layer in the SLIM Distributed System. SLIM adequately meets the requirements imposed on an abstract machine by an operating system and additionally provides an opportunity for adaptation to the character of the operating system. Historically SLIM was intended as a replacement for OCODE of the portable BCPL language compiler. The extent of its portability has been demonstrated by a variety of implementations, in both interpretative and translated forms, across real machine architectures from micros to mainframes.

The history of the SLIM machine lends its architecture some credibility and a list of the machine's intended objectives distinctly identifies its contributions as a vehicle for distributed system development. Peck [Peck 83], referring to SLIM as "an ideal computer", clearly states these objectives.

"... we shall describe an ideal computer which has been designed with several objectives in view. Firstly it should be a machine not unlike many present day computers. Secondly it should be reasonably simple so that it can be used for teaching elements of computing. Thirdly it should be suitable as a target machine for simple high-level languages such as BCPL. Fourthly it should be an aid in the portability of system programs. Fifthly it should be a machine on which it is possible to have a multi-user operating system." [Peck 83]

These objectives were consistent with the needs anticipated of an intermediate machine, and principally of interest was the objective suggesting SLIM could support a multi-user operating system; however, in this respect, its design was vague and incomplete. In fact, the responsibility of researching this aspect of the SLIM architecture was accepted at incipience of this thesis project and, in part, this paper reflects an effort to extend the formal definition of the SLIM design.
Previously, I described SLIM as a uni-process architecture. In contrast and in an attempt to better accommodate the structure of a Distributed Operating System, this thesis produced an enhanced version of SLIM with a multi-process architecture. A uni-process architecture - one without multi-process support - classifies the architecture of most present day micro computers. By definition, such a machine will provide for the execution of, at most, one sequential program. The inclusion of hardware interrupts permits control over an asynchronous hardware process, and thus gives the appearance of a multi-process architecture. However, software, which "extends the base machine architecture", still is needed for such machines to provide a satisfactory multi-process environment. The Verex operating system is an example of software that provides this function. A multi-process architecture will provide for the execution of, at most, N sequential programs where N is an integer greater than one. Note however, that program executions are not necessarily simultaneous. Thus, there is a distinction between multi-process and multi-processor architecture. The relationship between multi-process, multi-processor, as well as uni-process and uni-processor is illustrated below in figure 15.

![Figure 15. Uni-processor and multi-processor architectures.](image)

Most small computing systems are of a "uni-process / uni-processor" architecture. The IBM Personal Computer, with its single Intel 8088 microprocessor supporting the MS DOS or similar operating system, is an example of this architecture. The same IBM Personal Computer supporting the UNIX operating system is an example of the "multi-process / uni-processor" architecture.
### Registers:

<table>
<thead>
<tr>
<th>C</th>
<th>H</th>
<th>A</th>
<th>N</th>
<th>G</th>
<th>E</th>
<th>S</th>
</tr>
</thead>
<tbody>
<tr>
<td>program counter</td>
<td>high point of stack</td>
<td>accumulator</td>
<td>interrupt</td>
<td>global base</td>
<td>environment base</td>
<td>stack limit</td>
</tr>
</tbody>
</table>

### Memory:

... parameters linkage variables workspace ...

---

### Instruction Set:

#### Transfer:

<table>
<thead>
<tr>
<th>L&lt;U&gt;</th>
<th>S&lt;U&gt;</th>
<th>LI&lt;U&gt;</th>
<th>SI&lt;U&gt;</th>
<th>L$&lt;U&gt;</th>
<th>S$&lt;U&gt;</th>
<th>L:&lt;U&gt;</th>
<th>S:&lt;U&gt;</th>
</tr>
</thead>
</table>

#### Arithmetic:

<table>
<thead>
<tr>
<th>+&lt;U&gt;</th>
<th>*/&lt;U&gt;</th>
<th>/&lt;U&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>-&lt;U&gt;</td>
<td>&gt;&gt;&lt;U&gt;</td>
<td>~&lt;U&gt;</td>
</tr>
<tr>
<td>*/&lt;U&gt;</td>
<td>«&lt;U&gt;</td>
<td>-&lt;U&gt;</td>
</tr>
</tbody>
</table>
| */<U> | \/<U> | |}

#### Stack:

| P | PL<U> | M<U> | X |

#### Translocation:

<table>
<thead>
<tr>
<th>C&lt;U&gt;</th>
<th>R</th>
</tr>
</thead>
<tbody>
<tr>
<td>J&lt;U&gt;</td>
<td>?S</td>
</tr>
<tr>
<td>F&lt;U&gt;</td>
<td>?I</td>
</tr>
<tr>
<td>T&lt;U&gt;</td>
<td></td>
</tr>
</tbody>
</table>

#### Process:

<table>
<thead>
<tr>
<th>O</th>
<th>L$&lt;U&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q</td>
<td>S$&lt;U&gt;</td>
</tr>
<tr>
<td>U</td>
<td></td>
</tr>
</tbody>
</table>

#### Relational:

<table>
<thead>
<tr>
<th>=&lt;U&gt;</th>
<th>«&lt;U&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>«=&lt;&gt;U&gt;</td>
<td>&lt;=&lt;U&gt;</td>
</tr>
<tr>
<td>&lt;=&lt;U&gt;</td>
<td>==&lt;U&gt;</td>
</tr>
<tr>
<td>==&lt;U&gt;</td>
<td>=~&lt;U&gt;</td>
</tr>
</tbody>
</table>

---

**Figure 16.** Architecture of the uni-process SLIM intermediate machine.

---

**Fetch-Execute Cycle:**

LET W be the raw operand
TEST environ THEN W := E ELSE
TEST global THEN W := G ELSE
IF H-reference THEN
  \( W := \neg H; H \leftarrow 1 \)
IF indirect THEN W := IW
advance C to next instruction
execute the current instruction
When a single machine makes a number of tightly-coupled processors available, "uni-process / multi-processor" and "multi-process / multi-processor" architectures are possible. An example of the "multi-process / multi-processor" architecture is presented in a thesis, "The Design of a Distributed Kernel for a Multiprocessor System" [Boyle 82], were Boyle mapped Verex onto a multi-processor configuration.

Before a multi-process SLIM description is presented, the architecture of the uni-process SLIM machine will be summarized. A historical and more detailed presentation of uni-process SLIM is available from Peck and elsewhere [Peck 83, Fox 78, Hayter 83]. The reader may refer to figure 16 which highlights the essential elements of the uni-process SLIM architecture: a simple, orthogonal instruction set and minimal, yet complete, register set. Uni-process SLIM consists of some memory, seven special purpose registers, and a fetch-execute mechanism for instruction decoding and execution. Being of a uni-process architecture, SLIM supports the execution of at most one program, which, of course, must reside in the memory of the machine. SLIM is a stack oriented machine which, in fact, is the origin of the SLIM acronym (Stack Language for Intermediate Machines). The SLIM instruction set in most respects is quite ordinary. Its repertoire includes: monadic and dyadic arithmetic operations, conditional and unconditional translocation operations, register-memory transfer operations, process control operations, stack modification operations, and relational operations yielding true or false.

An intriguing acronym, CHANGES, names each of the seven SLIM registers. The program Counter (C register), which is advanced to the next instruction after the current instruction is decoded, marks the execution point within the program code. The stack High point (H register), Environment base (E register), and Stack limit (S register) together maintain the status of the program stack. The H register addresses the top of the stack and is modified either directly or, more often, indirectly by many of the SLIM instructions. The E register identifies the segment of stack which is local to the active program procedure and, as procedures are entered and exited, the E register is indirectly modified. The S register marks the extent of the program stack with respect to the total memory available to the SLIM machine. The exact use of the S register lacks definition, but this register might be used to separate the stack domain from a data or heap domain. The main intent of the S register is to provide a mechanism by which a stack overflow can be detected. The Accumulator (A register) is the object of most SLIM operations. The A register serves as the operand for monadic and dyadic operation; the value found at the top of stack is usually the other operand for dyadic operations. Arithmetic and relational operations leave their result in the A register. The iNterrupt register (N register) is associated with SLIM machine interrupts; however,
this aspect of the SLIM machine, at present, also lacks definition. The Global base (G register) identifies a segment of SLIM memory as a global data region providing local procedures access to shared program data.

The seven CHANGES registers represent the explicit register set of the SLIM machine; however, additionally an implementation of SLIM might make use of an implicit register set. Explicit registers are allied in that their use must be understood by the programmer, while the use of implicit registers are concealed from the programmer. The Work register (W register) is a common member of this implicit register set, but there may be others depending on the implementation. The employment of the W register is associated with SLIM memory addressing facilities as it generally holds, temporarily, the operand value provided by a program instruction.

Like most other machine architectures, SLIM provides a number of memory addressing modes. The SLIM architecture incorporates the addressing mode into the instruction operand, essentially, as a combination of three fields. The operand - an eight, sixteen or thirty-two bit quantity - is a field whose numerical value identifies the object of interest; additionally, there are two fields, the modifier field, and the indirection flag, which alter the extent of this operand value. The various permutations of these three fields result in five addressing modes which, in this thesis, are termed as follows: Immediate, Modified, Indirect, Indirect Modified, Stack.

<table>
<thead>
<tr>
<th>MODE</th>
<th>MODIFIER</th>
<th>EXAMPLE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Immediate</td>
<td>.</td>
<td>+&lt;n&gt;</td>
</tr>
<tr>
<td>Indirect</td>
<td>-</td>
<td>+I&lt;n&gt;</td>
</tr>
<tr>
<td>Modified</td>
<td>E</td>
<td>+E&lt;n&gt;</td>
</tr>
<tr>
<td></td>
<td>G</td>
<td>+G&lt;n&gt;</td>
</tr>
<tr>
<td></td>
<td>C</td>
<td>+@&lt;n&gt;</td>
</tr>
<tr>
<td>Indirect Modified</td>
<td>E</td>
<td>+IE&lt;n&gt;</td>
</tr>
<tr>
<td></td>
<td>G</td>
<td>+IG&lt;n&gt;</td>
</tr>
<tr>
<td></td>
<td>C</td>
<td>+I@&lt;n&gt;</td>
</tr>
<tr>
<td>Stack</td>
<td>-</td>
<td>+H</td>
</tr>
</tbody>
</table>

In the chart above, the modifier field is shown to be a set of four values: the null value, the E value, the G value, and the C value. The three letters in this set refer respectively to a register in CHANGES. For example, when the E modifier is used, the operand value of the instruction is
modified (added) by the contents the E register. The null modifier leaves the raw operand unmodified. The indirection flag determines whether it is the value or the contents of the memory location addressed by the value that is to be placed in the W register.

The SLIM instruction set, in synoptic form, is presented in figure 16. Each instruction starts with a mnemonic representation of the operation followed by, where an operand is possible, the symbol "<W>", which denotes the W register. For example, "L<W>" is the SLIM "load" instruction whereby the value contained in the W register would be transferred to the accumulator (A register). If the actual assembler instruction was of the form "LIE1", the value at the memory location addressed by the contents of the E register plus 1 would be available in the W register for transfer to the A register. Figure 16 logically groups the SLIM instruction set into 6 categories in an attempt to minimize the readers efforts in a basic understanding of the machine's workings. A segment of SLIM assembler and a high-level language analogy in BCPL are presented below.

```
$"WRITENUM"
  @1: D1 LIE-3 <=-10 F@5 LIE-3 /10 Cl@2 D1
  @5: LIE-3 /*10 - +'0 CIG60 D1 R

$"WRITEN"
  @3: D1 LIE-3 >=0 F@6 LIE-3 - Cl@2 D1 J@7
  @6: L' CIG60 D1 LIE-3 Cl@2 D1
  @7: R

  @2: D@1  @8: D@3
```

The two procedures shown above, "WRITEN" and "WRITENUM", cooperate in an effort to produce a character string representation of a binary integer. The "D" instruction shown above is actually an assembler directive requesting storage with a value as provided in the operand. Note that each procedure begins with this directive which, in such a situation, describes the number of parameters expected by the procedure.

```ml
LET writenum(n) BE
  { IF n <= -10 THEN writenum(n/10)
    wrch(-n REM 10 + '0')
  }

AND written(n) BE
  TEST n >0 THEN
    writenum(-n)
  ELSE
    { wrch('-'); writenum(n) }
```
Although it is hoped that a cursory familiarity of SLIM is conveyed by this example, the reader is once again directed to [Peck 83] where a greater background and understanding of the assembler language is of interest.

The modifications to the uni-process SLIM design, with respect to the objectives of this thesis, centered on incorporation of constructs which facilitate process management. The first such modification involved the development of an instruction, the SLIM Originate instruction (O mnemonic), by which a new process could be established dynamically. The second modification, towards administration of SLIM hardware resources, determined the function and operation of the SLIM device load and store instructions (L$ and S$ mnemonic). Further modifications, such as the refinement of the SLIM fetch-execute cycle, and the definition of process blocking, interrupt handling and operation of the SLIM N register completed revisions to the SLIM architecture. The result of these efforts was an abstract machine model with a capability for service as the software base of a multi-process distributed operating system.

Essentially, the multi-process SLIM architecture differs from the uni-process architecture only in that it supports, simultaneously, more than one SLIM processes. When an executing process is blocked, such as by device activity or the expiration of its time quantum, the SLIM processor schedules another. The development of SLIM devices compatible with multi-processing, provided the greatest implementation challenges; however, these complexities are actually hidden from a user of multi-process SLIM. A second notable distinction between uni-process SLIM and multi-process SLIM is the separation of code and data components in the multi-process architecture. Although each process must maintain a separate and complete data segment, as a result of the separation, a code segment can be shared among several processes. As well, in the interest of code sharing, a new member of the explicit register set, the code Base (B register), was introduced.

The organization of a SLIM process and its relation to the multi-process SLIM CPU is illustrated in figure 17. A process is created when two memory segments, one for code and one for data, are allocated and initialized; the allocation of memory segments is performed by manipulation of a SLIM device. The code segment is initialized when the program code is loaded. The more complex initialization of a data segment involves first setting register values into a section of the data segment termed the "Register Section" and then loading program data into a section termed the "Global Section". The Stack Limit (S register) delimits a variable boundary between the Global section and a third section which need not be initialized, the "Stack Section". The Program Counter (C register) is set to the location in the code segment which associates with the first instruction to
be executed. The Global Base (G register) identifies the starting location of constant and variable data. As with uni-process SLIM, the H and E registers are initialized to the high-point of the stack, and the A register is undefined - actually, in the implementation, A is used to pass the process identifier to the process. The Interrupt register (N register) must be set to zero since the setting of a bit in this register to the high state signals the blocking of process execution. The Originate instruction adds the process identifier of this new process to a process queue in the SLIM CPU and the Quit instruction, subsequently, removes the process from the queue.

Figure 17. Simplified view of the multi-process SLIM architecture.

The Register Section of the data segment is used as storage for a process state during the suspension of its activity. When a process begins execution, the register values set into the Register Section during initialization are loaded into the CPU registers. If the process is blocked, the CPU register values are unloaded back into the Register Section so that the state of the next process can be reloaded. A determination on the state of a process, active or blocked, as well as the loading and unloading of registers during the scheduling of processes, is performed by the fetch-execute cycle of the SLIM interpreter.

A process is blocked by the change of an interrupt register bit from the low state (logic 0) to a high state (logic 1). Each bit relates to a class of SLIM device such that a conflict, due to concurrent device activity, is easily resolved. For example, as it is unlikely that an operation on a floppy
The diskette device will complete within the time frame of a typical SLIM instruction, during an invocation the interrupt bit allocated to the floppy diskette device will be set. When the device has fully serviced the operation, the process is enabled for further execution at the next scheduled interval by the resetting of the same interrupt bit. Thus, the SLIM design, by the mechanism described, abstracts the primitive interrupt signal such that, at the Intermediate Machine layer, it manifests as a process being blocked.

Although SLIM supports and defines a mechanism for device access, the device abstraction of the SLIM machine is user defined. The implementation of this access mechanism, which is similar to the software trap facility provided by many machine architectures, is by means of the device load and store instructions (L$ and S$). Each SLIM device is a logical manifestation of a real hardware device or the functional representation of machine resources. For example, the current SLIM architecture provides the logical manifestation of a floppy diskette device (DSK), video screen device (SCR), keyboard device (KYB), and printer device (PRN). In addition, processor memory, the processor itself and interprocess communication services are provided as SLIM device representations. The processor device (PRC) allows control over processor parameters, such as process quantum values. The segment device (SEG) provides processor memory management such that new code and data segments can be allocated or deallocated. In conjunction with the segment device, a memory device (MEM), whereby a process can access memory locations beyond its own working segments, is required. The interprocess communication device (IPC) provides a facility for passing messages between processes; this facility maps onto the physical link (ARCNET LAN) between the machines. Finally, a special error device (ERR) which provides a reliable service during a panic situation is found to be useful.

4.3 SLIM DISTRIBUTED OPERATING SYSTEM

The organization of the SLIM Distributed Operating System is presented in figure 18. In addition to the major layer divisions, the sub-layer divisions are shown and within each sub-layer all significant code modules are identified. All modules above the Intermediate Machine layer are SLIM object modules and each such module is implemented as a separate SLIM process. As is suggested by the multiple instance of the loader and filer servers, certain modules may be replicated throughout the system domain. Figure 18 accurately reflects the current state of the SLIM Operating System and although its structure is rudimentary it is sufficient for simple operations.
Figure 18. Code module organization for the SLIM Distributed Operating System.
The processes of the Services layer are organized into three sub-layers. The bottom sub-layer, the Resource Servers sub-layer, is the set of all processes dependent on machine resources and devices. Management and administrative server processes are grouped at the Management Servers sub-layer. Management servers do not depend on any machine resources and, unlike the resource servers, are able to migrate freely throughout the system domain. The third sub-layer is the System Clients sub-layer, which is the sub-layer at which system command processes are grouped. As well, processes at each sub-layer are intended to operate within the environment provided by the sub-layers and layers beneath them. For example, management servers may take advantage of the services provided by resource servers. A print spooler would likely make use of both printer and filer resource servers in the performance of its function. A system client, such as a list program, would, in turn, make use of a filer resource server and the print spooler.

The kernel of a distributed operating system should provide more services than are actually provided by the Kernel layer of the SLIM Operating System. The boot module shown in figure 18 provides the mechanism by which the SLIM Operating System is initialized. Further development of the Kernel layer might involve the addition of other sub-layers above the Bootstrap layer, which would help to promote distributed function. The implementation of these sub-layers could be in the form of coroutines. For example, as the sharing of code segments is possible with the SLIM design, a set of code segments providing kernel services as coroutines could reside at a SLIM node. Presently, kernel services other than the bootstrap, such as transport level interprocess communication, are provided as a library of routines that are assembled with every SLIM process. As a result, the code of these messageing routines is, therefore, reproduced many times across the system.

The messageing routines provided in the SLIM Operating System are similar to the synchronous interprocess communication facilities provided by VEREX [Lockhart 79]. There are three primary messageing routines:

\[
\begin{align*}
    &\text{Send}(\text{pid}, \text{buf}, \text{n}) \\
    &\text{Receive}(\text{pid}, \text{buf}, \text{n}) \\
    &\text{Reply}(\text{pid}, \text{buf})
\end{align*}
\]

The send routine expects its first parameter (pid) to identify the destination process. The next two parameters (buf and n) designate the buffer, containing information to transmit, and its size. The
size parameter does not mark the length of the message to transmit, but instead delimits the amount of space available to the returned reply. The length of the message to transmit is actually contained in the first location of the buffer. The receive parameters are similar to the send parameters except that the pid will contain identification of the source process. The reply routine returns a response or acknowledgement to the invoker of a send routine. Note that reply does not expect a return message and, therefore, does not require the n parameter. All three of these routines map easily onto the SLIM interprocess communication primitives provided by the IPC device.

![Diagram of IPC Send Buffer and IPC Reply Buffer](image)

Figure 19. Transport Protocol between clients and servers.

The input / output activity of all system clients and application software is mapped by a library of routines onto the interprocess communication facility described above. The protocol is similar to the object-based protocol presented by Cheriton in a technical report entitled "Distributed I/O using an Object-based Protocol" [Cheriton 81]. Server requests transmitted to resource or management servers, via the send routine, are structured as shown in figure 19. The reply buffers are also shown. The library routines, such as are necessary for output of numbers or strings (WRCH(ch), WRITEN(number), WRITES(string)) are similar, in function, to those of the BCPL portable library [Richards 77b].

The architecture of all servers are similar. As well, the transport protocol between these servers is the same. These conditions provide for flexible control and of all servers in the system.
For example, servers are interchangeable at the user level such that the shell set command does not specify the class of server in use. The general architecture of each server in the is as follows:

```c
IO_server()
{
    state = Initialize();

    repeat
    {
        Receive(pid, buffer, n);
        select (OPERATION_CODE[buffer])
        {
            case CREATE_INSTANCE_REQUEST:
                reply = Create_instance(state, buffer, pid);
            case CLOSE_INSTANCE_REQUEST:
                reply = Close_instance(state, buffer, pid);
            case READ_INSTANCE_REQUEST:
                reply = Read_instance(state, buffer, pid);
            case WRITE_INSTANCE_REQUEST:
                reply = Write_instance(state, buffer, pid);
            case QUERY_INSTANCE_REQUEST:
                reply = Quey_instance(state, buffer, pid);
            default: reply = ILLEGAL_REQUEST;
        }
        REPLY_CODE[buffer] = reply;
        Reply(pid, buffer);
    }
}
```

The algorithm presented above describes the general form of the resource and management servers in the SLIM Operating System. In addition to the object-based protocol, this algorithm was also borrowed from [Cheriton 81].
5.0 RESULTS AND EVALUATION

The implementation of multi-process SLIM marks the realization of an intermediate machine suited for use as the software base of a distributed computing system. As this is the goal pursued in the thesis, a cursory evaluation of the research must render a favorable judgement. This chapter attempts to evaluate further the merit of the intermediate machine as a software base, thereby, providing some form of measure by which an intermediate machine based distributed computing system can be related to other computing systems.

In that the objectives of the research were met, this thesis may be considered a useful contribution to the study of distributed computing systems. The first objective was to determine the feasibility of applying the intermediate machine to the problem of providing a software foundation or base for a distributed computing system. The second objective was to define a form for the intermediate machine in this application. The third objective was to exercise the intermediate machine approach to distributed computing system design by means of experimentation and trial implementation. A response to each of these objectives is provided in following paragraphs.

In this thesis, the feasibility of the Intermediate Machine approach is established both on a theoretical level and an experimental level. The theoretical basis of its feasibility, which was detailed in chapter three, is established by the parallel between the intermediate machine and another form of software base, the "security kernel". However, stronger evidence for feasibility of the approach is demonstrated experimentally by application of the Intermediate Machine approach in the implementation of the SLIM Distributed System. The successful development of the SLIM Operating System suggests that it is possible to implement a comprehensive and complete distributed operating system entirely within the machine independent environment provided by an abstract intermediate machine software base.
When workable specifications were finally determined, the construction of the intermediate machine layer, the SLIM machine, proved not to be a difficult matter. Given an above average understanding of a new target machine, the estimated time frame for the construction of a SLIM software base for this new machine, which would consist of an instruction interpreter and a minimal number of device interfaces, totals one month. Such a minimal SLIM node could possibly attach to and function within an existing SLIM network. However, a minimal number of devices alone would not be sufficient for the total requirements of a distributed operating system. Each additional device interface would extend the time frame by one to four weeks, depending on the complexity of the device and the sophistication of the hardware support. Of course, if existing BIOS support is already available at the new target machine and can be immediately integrated into the intermediate machine layer, the time frame can be shortened considerably.

Fortunately, upon completion of the intermediate machine layer, the target is prepared for a direct transfer of any and all operating system software. Thus, the total time for extending or porting an existing distributed operating system to a new and architecturally different site, using the Intermediate Machine approach, could be much less than the time taken for traditional architectures. There are, however, some prerequisites to the implementation of the SLIM machine at a new site. First of all, one should expect that it would prove extremely awkward to implement an abstract machine as sophisticated as SLIM on a microprocessor less advanced than the one which served as the target of the thesis experiment, the Intel 8086. A good number of registers combined with a functional instruction set, memory protection operations and multi-process support, as well as hardware interrupt vectors, are the minimum prerequisites for the target processor.

A notable obstacle and prerequisite pertaining to the networking of SLIM nodes is that a physical link between the target processors of each SLIM node must be established. In addition, the intermediate machine layer, essentially, provides the Network Layer, as described by the ISO OSI reference architecture [Tanenbaum 81], in terms of protocol hierarchy. This communication service must be coded into the intermediate machine microcode, which greatly prolongs software extending and porting time frames, or it must be provided as a BIOS or firmware operation at the target processor.

At one point in the development of the SLIM distributed system, experimentation led to an attempt to implement a rudimentary network facility across an RS 232 physical link. As neither BIOS nor firmware, interrupt driven, network support was available for an RS 232 link on the target machine, the interprocess communication device interface had to be entirely microcoded into
the intermediate machine layer. This component of the intermediate machine developed into the most complex and unreliable of the software modules, though this was perhaps partially due to the author's lack of experience with such software, and accounted for two months of the time spent on software experimentation.

The second objective, that of defining a form for the intermediate machine, was also met by both theoretical and experimental means. In the realization that an intermediate machine is a form of software base, it was recognized that theoretical questions regarding the function and form of such a base had been addressed numerous times and by highly qualified people [Popek and Kline 79]. As mentioned in chapter three, the theoretical approach to determining the form of the intermediate machine was simply to extend the work of other researchers in security kernel design, to the design of an intermediate machine software base. Experimental research, represented by iterative attempts to implement an intermediate machine layer, provided an opportunity to refine the design of this intermediate machine software base. The point that must be emphasized, however, is that the specification of multi-process SLIM is not intended for purposes of providing the definitive solution to the problem of intermediate machine design. Instead, multi-process SLIM is offered as one of possibly many candidates for application as a software base. As well, it is recognized that a totally different intermediate machine design may be required under circumstances of another distributed system implementation.

In satisfying the third objective, an empirical evaluation of the Intermediate Machine approach became possible. Although it is stressed that the development of the SLIM Distributed System was intended only as an experimental exercise of the Intermediate Machine approach, an actual implementation provides an opportunity to investigate both the benefits and difficulties of the approach. Further, it presents an opportunity for meaningful application of the criteria introduced in chapter two, which were listed as homogeneity, portability, extensibility, function and performance, to an intermediate machine architecture. By examining the architecture of an intermediate machine based distributed system under the constraints of these criteria, one can make a better determination on its advantageous and disadvantageous.

First of all, it can be said that a collection of identical SLIM implementations represents a homogeneous set of nodes. Since the target hardware in this research is a network of IBM Personal Computers, and each machine executes the same SLIM microcode, this set of SLIM nodes is certainly homogeneous. Note that each of the SLIM nodes in the network does not necessarily provide an identical set of services; however, for those services that are replicated by various
nodes, the intermediate machine guarantees that the replication is uniform. For example, not every SLIM node may provide a floppy diskette storage device, but where this service is provided, the interfaces are homogeneous. Thus, in the SLIM Distributed System, a degree of homogeneity is attained whereby the same floppy server process code can execute on any SLIM node providing the floppy diskette storage device, in spite of any hardware differences among the physical floppy diskette storage objects.

In the case of a storage object, there are certain physical capabilities or parameters which the intermediate machine does not make homogeneous. The capacity of the storage, usually measured in blocks or bytes, is one such parameter. Such parameters must manifest, at least to the operating system layers, and sometimes even to the application and user layers. Fortunately, the intermediate machine provides the means to implement parameterization in a manner which can be made homogeneous across a particular class of device.

![Distributed Computer System Diagram](image)

Figure 20. Ideal configuration of an intermediate machine system.

If every intermediate machine node in a network does not provide the same set of services, as in the SLIM Distributed System, it may be argued that the intermediate machine layer is not fully homogeneous. In such an argument, it would be expected that each SLIM node provides the same complete set of services regardless of the physical configuration of its hardware. An intermediate machine layer structured in this way forces complex network management into the intermediate
machine implementation and, as well, would support network based services. Admittedly, SLIM does not represent this level of abstraction or sophistication in an intermediate machine, and a network facility is provided only as a service to its client. The SLIM design does not incorporate networking at the intermediate machine layer and, instead, expects network based services to be developed at the operating system level. It may or may not be desirable to incorporate networking into other intermediate machine designs.

In terms of the reference architecture described in chapter two, figure 20 above depicts the structure of an idealized intermediate machine based distributed system. Figure 21 below, in contrast to this idealized system, depicts the actual structure of the SLIM Distributed System. Figure 20 suggests that despite, the heterogeneity of both hardware nodes in the system, all nodes above the Intermediate Machine layer are homogeneous. The nodes of the Kernel layer differ from the nodes of the Intermediate Machine layer, only in terms of the services provided by the Kernel layer, and, in the same way, the Services layer nodes differ from the Kernel layer nodes. However, all nodes of the Intermediate Machine layer are homogeneous with respect to each other, and, unless either the Kernel layer or Services layer deviate from this aim, the same is true of nodes at each of these layers as well.

In an ideal intermediate machine based distributed system, all nodes of the intermediate machine would provide exactly the same services, or, in other words, the intermediate machine layer would be fully homogeneous. The intermediate machine layer of the SLIM Distributed System is not fully homogeneous when each of the hardware processors provides a different set of services. However, if all hardware processors do provide the same set of services, the SLIM Distributed System intermediate machine layer can produce the same full homogeneity as the ideal intermediate machine layer. Of course, the actual implementation of hardware services can be different at each processor node since only the logical abstraction at the intermediate machine layer of these services is considered in the determination of the full homogeneity. For example, a network of machines, each providing a console service (keyboard and screen), storage service (floppy diskette or hard disk), and printer service can produce a fully homogeneous intermediate machine layer. If one or more nodes fails to provide a storage or printer service, by definition the intermediate machine layer is not fully homogeneous. The same would be true if one processor node provides a service, such as the printer service, where the logical abstraction at the intermediate machine layer does not map onto the physical hardware. In this instance, a different printer device interface would be required at the intermediate machine for this node, which likely will result in heterogeneity of printer services.
Although the SLIM intermediate machine layer does not provide network based services and, as a result, is not necessarily fully homogeneous like the ideal intermediate machine based distributed system, the SLIM Distributed System does provide a fully homogeneous Services layer. Therefore, at the level of the Applications layer, or from the view of the user, there are few differences between an ideal intermediate machine based distributed system architecture and the architecture of the SLIM Distributed System.

In figure 21, the three hardware processors provide different services and, consequently, are marked heterogeneous. The services provided by hardware nodes A and B are functionally similar; however, hardware node C, although perhaps similar to nodes A and B in some respects, is functionally different from them. Functional similarities or differences are marked by a continuous or broken layer notation. In the SLIM Distributed System, hardware nodes B and C represent two IBM Personal Computers. The functional dissimilarity between the two nodes arises from the fact that node C provides a printer service, while node B does not. In all other respects, notably the storage service (floppy diskette drive), console service (keyboard and screen), and processor service (Intel 8088), the two nodes are alike; however a single difference such as the printer service makes them heterogeneous. Hardware nodes A and B are functionally similar in that each provides the same set of services as the other, but these nodes still are not homogeneous because one or more of the services at node A has an interface different from the identical service at node B. For
example, when the processor at node A, which is expected to be a Motorola MC68000, is different from the processor at node B (Intel 8088) then this same service presents different interfaces at the two nodes.

Across a functionally homogeneous hardware layer, SLIM can provide a strictly homogeneous set of process nodes. However, when a set of hardware nodes is neither strictly nor functionally homogeneous, the process nodes of the SLIM intermediate machine layer will not be homogeneous. For this reason, the SLIM processes associated with hardware node C are heterogeneous with respect to the processes of hardware nodes A and B. Since hardware node C provides a printer service, while hardware nodes A and B do not, only SLIM processes associated with node C can support a printer device interface.

In the SLIM Distributed System, the Kernel layer provides a bootstrap facility but, at present, does not provide any other services. Thus, although a kernel may supersede the intermediate machine serves with network based services, the SLIM Distributed System kernel does not. Since effectively, except during bootstrap the SLIM kernel is absent, its services are provided uniformly across all nodes of the Intermediate Machine layer. As well, the logical nodes of the SLIM kernel are homogeneous only to the extent that the intermediate machine nodes are homogeneous.

The Services layer of the SLIM distributed System does implement network based services, the result of which is that all its nodes are homogeneous. Therefore, an application process can migrate freely among process nodes of the Services layer independent of the heterogeneity of lower layers. Figure 18 in chapter four details the structure of the SLIM Distributed System and describes the Services layer as a combination of three sub-layers. Since the processes of the bottom sub-layer - a collection of Resource Servers - utilize the device interfaces of the intermediate machine layer, these processes are still limited in their migration to compatible Kernel nodes. However, the processes of the other two sub-layers, the Management Servers and the System Clients, make use of network based services only and are able to migrate freely across all nodes of the Kernel layer.

In practice, the extensibility of the SLIM Distributed System architecture has not been tested, although, in theory, it holds much promise. The difficulties with extensibility of the SLIM System is localized to the intermediate machine layer, since the software of the Kernel, Services and Application layers are totally portable across SLIM machine implementations. In this respect, the architecture of the SLIM Distributed System has a great advantage over most other distributed
systems, as the intermediate machine layer represents only a small portion of the total system and application software. Although the SLIM design already offers a strong record of portability, every new hardware site poses its own unique problems and difficulties. These unknowns make it difficult to predict the success of future attempts at system extension, but, in general, the SLIM Distributed System architecture should improve extensibility while lessening the time and effort involved as compared to re-coding the system at the new node.

Similar remarks apply to the portability of the SLIM Distributed System. The size of an implementation of the SLIM intermediate machine varies with the choice of device support, but the average is approximately 5K bytes. Porting the SLIM Distributed System requires reproduction of this 5K bytes of code at the new site and a rewrite of these programs usually will be necessary. This reproduction may take longer than one might expect for code of this size, since efficiency and simplicity are important. In the interest of efficiency, the thesis implementation of the SLIM machine was coded in assembler language; however, at the expense of some efficiency, one could improve portability by developing a high-level language implementation. Unfortunately, by definition, the intermediate machine layer is a machine dependent program, which means it is unlikely that a high-level language implementation will port to a new site without major modification. Differences in hardware architecture often present the programmer with portability obstacles. The isolation of machine dependent software to the intermediate machine layer does not ease the confrontation with these obstacles, although it does lessen their impact on operating system software.

In terms of function, the potential of the Intermediate Machine approach appears strong in relation to the other distributed system designs discussed in this thesis. Since an intermediate machine layer provides a homogeneous environment, distributed functions can be implemented with the simplicity pursued in the design of the EDEN project. Unlike EDEN, however, there is no need to limit the hardware base to a set of homogeneous machines. In contrast to LOCUS, the presence of an intermediate machine layer facilitates the same function while eliminating requirements of addressing heterogeneity at the Services layer. Unfortunately, the functional capacity of the experimental SLIM Operating System does not meet the potential of the Intermediate Machine approach. Although remote process invocation and resource sharing are possible, neither dynamic process migration nor host migration were implemented. Certainly, if such operations were to be implemented, their implementation in a homogeneous environment would be more workable than in a heterogeneous one.
The major weakness of the SLIM design is found in its performance. A common form of performance evaluation is provided by the "Sieve of Eratosthenes" benchmark. The table below shows the results of this benchmark.

<table>
<thead>
<tr>
<th></th>
<th>C/8086</th>
<th>SLIM</th>
<th>BASIC</th>
</tr>
</thead>
<tbody>
<tr>
<td>SIEVE/PC</td>
<td>~1 sec</td>
<td>36 sec</td>
<td>219 sec</td>
</tr>
<tr>
<td>SIEVE/AT</td>
<td>&lt;1 sec</td>
<td>13 sec</td>
<td>87 sec</td>
</tr>
</tbody>
</table>

For the test, three forms of the sieve program were implemented. The prime end of the scale is represented by an implementation programmed in the C language and compiled to Intel 8086 machine code. A version executable by the MS DOS BASIC interpreter marks the slow end of the scale. The SLIM version is programmed in SLIM assembler language and assembled to SLIM machine code. The SLIM benchmark test was conducted in a manner such that the sieve program was the one and only executing process. In the chart above, the performance of the SLIM interpreter improves when the target machine, instead of an IBM PC, is replaced with an IBM AT.

As one might expect, the total time taken for the sieve task to complete increases when the process must compete with others for the resources of the target machine. The relationship between the execution time and the number of processes is not linear because certain processes are blocked during the benchmark evaluation.

<table>
<thead>
<tr>
<th></th>
<th>1 Process</th>
<th>3 Processes</th>
<th>7 Processes</th>
</tr>
</thead>
<tbody>
<tr>
<td>SIEVE/PC</td>
<td>36 sec</td>
<td>72 sec</td>
<td>92 sec</td>
</tr>
</tbody>
</table>

In the chart above, the "number of processes" heading is the total of processes, including the sieve program, on a single SLIM node. The operator, console and sieve programs are the three processes in competition for the benchmark evaluation represented by the second column. Since the console process is always actively monitoring for keyboard activity while the operator, for the most part, is blocked, the total time of execution becomes twice the value of the single process benchmark. In the third column, the operator, loader, filer, console, as well as two instances of the shell and the sieve compete.

A further evaluation of SLIM Operating System performance was carried out on the process loading activity. The loading of processes is performed first by the bootstrap software, and, after
the operating system is fully bootstrapped, by a shell process. The time taken by this activity relates closely to the performance of intermediate machine device facility. Unfortunately, in this respect the SLIM design is currently limited to character input and output only, and block transfers of information between the SLIM CPU and SLIM devices are not possible. Revisions to SLIM providing block transfer capability to the floppy diskette device (DSK), memory device (MEM) and interprocess communication device (IPC), could greatly increase the performance of the SLIM Operating System.

The bootstrap program loads a block of code from the floppy diskette and transfers it, character by character, to the memory device. The shell process performs much the same activity; however, it makes use of the services provided by the filer and loader processes. In addition, interprocess communication is necessary between the filer and shell, and then between the shell and loader, again character by character. Mainly, as a result of this overhead, the chart below shows the time taken to load processes increases from about 2.5 seconds per 1K, during bootstrap, to 15 seconds per 1K.

<table>
<thead>
<tr>
<th>Bootstrap</th>
<th>Local Node</th>
<th>Remote Node</th>
</tr>
</thead>
<tbody>
<tr>
<td>Shell.i/PC</td>
<td>7.5 sec</td>
<td>45 sec</td>
</tr>
<tr>
<td>(2110 bytes)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The chart also provides the time taken to load a process at a remote SLIM node. Note that, within the resolution of 1 second, the time taken for a remote load is the same as a local load. The reason for the similarity between the local and remote operation is that the overhead of the ARCNET local area network is insignificant when timed with a 1 second resolution.

In terms of its performance, the SLIM system intermediate machine rates poorly; however, in its defense, the author relates its potential to the success of other intermediate machine implementations [Hayter 83]. In fact, the work of this thesis is premised on an assumption that intermediate machines can be made to operate within the parameters of acceptable software performance. Hayter's work concentrates on optimization of a SLIM interpreter and his thesis provides several options for improving the performance of our SLIM interpreter.

In an effort to better substantiate the claims of this thesis, some optimization techniques were applied to the sieve benchmark. Firstly, several instructions were removed from critical loops in the "Sieve of Eratosthenes" program. The effect of such optimization is noted in the second column
(OPTIMIZED) of the table below. Secondly, a different instruction decoding mechanism was implemented in the SLIM interpreter. Although this modification produces a relatively large increase in the size of the interpreter, the performance increase is equally large. The results of the modification are shown in the third column (SLIM V1.1) of the table.

<table>
<thead>
<tr>
<th></th>
<th>SLIM V1.0</th>
<th>OPTIMIZED</th>
<th>SLIM V1.1</th>
</tr>
</thead>
<tbody>
<tr>
<td>SIEVE/PC</td>
<td>36 sec</td>
<td>30 sec</td>
<td>16 sec</td>
</tr>
<tr>
<td>SIEVE/AT</td>
<td>13 sec</td>
<td>11 sec</td>
<td>6 sec</td>
</tr>
</tbody>
</table>

When compared to the earlier results of this benchmark - above, shown in the first column - our SLIM machine holds greater promise. As well, the SLIM design is based on the pursuit of code compactness and not efficiency of program execution. With these facts in mind, one should expect that the performance of an intermediate machine software base can be made acceptable.
6.0 Conclusions

The intermediate machine, employed as a foundation in the construction of a distributed operating system, provides the architects of distributed computing systems a method for dealing with the complexities of heterogeneous hardware. The formalization and exploration of the Intermediate Machine layer, which constitute a major portion of the efforts in this thesis, draw affirmation of its relevance in computer system design. It is this relevance that the author would like to expound in these concluding remarks.

This thesis has proposed a new element, the Intermediate Machine layer, in the structure of a distributed computing system. The intermediate machine serves as a tool with which one may isolate software primitives influenced by machine dependence. Its application is the principal role in a strategy for development of a hardware independent distributed operating environment. Whereas existing distributed computing implementations - without the assistance of an intermediate machine layer - often surmount the problems posed by the physical separation and heterogeneity of networked components, few of these solutions are likely to deal with these problems more effectively or more completely than an intermediate machine based implementation.

Intermediate machine specification must serve strict requirements imposed by the function of the Intermediate Machine layer. Enumerated and summarized, the function of the Intermediate Machine layer is as follows:

[1] The intermediate machine layer serves to buffer software development from hardware evolution. The intermediate machine defines a static interface to which both hardware and software must always conform.

[2] The intermediate machine "extends the base machine architecture to provide an environment suitable" for operating system development.
The intermediate machine layer furnishes its client, the operating system, with a machine independent architecture. All hardware dependencies are localized to, and addressed by, the intermediate machine.

The intermediate machine layer establishes a homogeneous network model across heterogeneous hardware nodes by means of instruction interpretation, device abstraction and hardware extension. This homogeneous network model promotes software portability between nodes of the Distributed Computing System.

The identification and refinement of this function together with the development of an Intermediate Machine layer, which implements the function, is offered as the major contribution of this thesis.

The many issues and concerns effecting the design of an operating system kernel apply equally well to the design of an Intermediate Machine layer. In fact, it may be argued that the Intermediate Machine layer is simply an alternative form of operating system kernel. However, it is suggested the converse view reflects the paradigm more accurately; that is, an operating system kernel should be considered a subordinate form of intermediate machine. Also, I believe further efforts in intermediate machine design would benefit from considerations associated with this view. For example, the implementation of an intermediate machine software base might follow the kernel team design scheme used in the Team Shoshin kernel [Wang 86, Acton et al. 85].

Popek lists function, performance, hardware, and security policy as the main factors of kernel specification. Traditionally, and more so since the advent of the security kernel, hardware consideration and security policy have dominated at the expense of function. Furthermore, an engineering effort employing program verification methods to establish the reliability of an operating system kernel produces one artificially constrained in size and complexity. Unfortunately, one should expect that software developers, applying similar limitations to the kernel of a distributed operating system, will become discouraged by conflicts between function and other requirements. The author advocates a kernel based operating system architecture and the value of a minimum size, secure operating system base is not in dispute; however this thesis suggests that this role is better challenged by the Intermediate Machine layer. The kernel of a distributed operating system, then liberated from such primary responsibilities, can impact functional requirements which are propounded to be greater for a distributed system more thoroughly.

In composing this thesis, the analysis of computer systems with respect to their perceived
homogeneity and heterogeneity proved useful. An attempt to define homogeneity of computer
networks resulted in classification of distributed computer systems and a distinction between
distributed computer systems and federated computer systems. These classifications and the
method by which they were delimited may seed further interest in similar study of computer system
homogeneity. Essentially, in this thesis, the homogeneity of a computer system is described as a
reflection of component homogeneity. In borrowing Watson's "Distributed Systems Architecture
Model", it was credible to suggest that a computer system is a composite of hardware and software
layers and that the homogeneity of these layers determines the degree of homogeneity asserted by
the computer system itself. Furthermore, an attempt was made to describe the homogeneity of a
layer which is defined true if the set of nodes effected by the layer is homogeneous. Conveniently,
the set of one node is always homogeneous.

The development of a major software system, frequently, is an iterative process and, in this
way, the development of the SLIM Distributed System provides many opportunities for future
research. As poor performance of the intermediate machine layer results in impairment of the
distributed system operation, the first opportunity lies with improvements in this regard. However,
instead of reworking, extending and continuing the efforts of the thesis implementation, it would be
better to make use of the SLIM Distributed System as a road map in the top down development of a
new, more comprehensive and complete distributed system. In addition, a number of design
corrections and refinements, as per the advantage of hind sight, could be better implemented
through a second attempt at the design and development process.

The design of the SLIM Operating System was in response to the need for experimentation
with the implementation of the SLIM intermediate machine. Consequently, its development was
more a bottom up process than a top down one, the result of which is a functionally primitive and
incomplete distributed operating system implementation. A second attempt at employing the
intermediate machine approach in the development of a distributed computing system should begin
with a thorough analysis of user requirements and expectations for the system, followed by
specification of an integrated, consummate, responsive and functionally powerful distributed
operating system. Actually, as very little consideration was given to aspects of the user interface of
distributed operating system, such steps would lead to significant improvements in this respect.

Although the entire SLIM Operating System executes in a SLIM environment, none of the
software, in this thesis, was programmed in a high-level language. A SLIM assembler was
developed and all server and client software was programmed in the SLIM assembler language. Since arguments for the use of a high-level language over the use of an assembler language need not be repeated here, even for one as sophisticated as the SLIM assembler language, a strong recommendation would be to implement a high-level language compiler for SLIM, in the development system used for the implementation of the operating system. For this thesis, the IBM PC / DOS development system was used and, although BCPL/SLIM compilers do exist, at the time, none had been implemented for it. A further suggestion would be to implement a SLIM compiler for a more popular high-level language, such as C or Pascal, either for operation in place of, or in conjunction with, BCPL software.

The following suggested refinements, within the SLIM operating system, might help to improve both its function and performance. Ideally all loader servers, which are the processes responsible for management of a processor resources, should provide homogeneous SLIM processor services. However, the management of other processors, perhaps for special numerical computations, could easily be accommodated by the introduction of another class of loader server. Access by application software to this new class of loader would be no different than their access to SLIM loader or other servers. However, the information or data transferred to these new loaders would not be SLIM programs. Instead, only programs specifically designed for uses with these special processors would be valid. A further operating system refinement involves a rework of both the SLIM interprocess communication facility and the transport layer protocol of the SLIM operating system. In this thesis, a simple and incomplete interprocess communication facility was implemented, and, although it adequately served its purpose, there was much room for improvement in terms of both function and performance [Acton 85]. For example, the static allocation of the process identifiers - which are used by interprocess communication - could be replaced by a better host identification scheme [Chanson 86].

Regarding the performance of the intermediate machine layer, a number of improvements are recommended. The speed of the interpreter can be increased by certain techniques, such as those discussed in the thesis by Ron Hayter [Hayter 83, Bell 73], including implementation of threaded code. A time quantum, per process, different from the 50 ms implemented in the SLIM interpreter of this thesis, might improve the responsiveness of the operating system. A process prioritization scheme might also improve responsiveness although perhaps for high priority functions, the use of dedicated processors is a better alternative. The implementation of block input/output for SLIM device interfaces would be a significant design correction over slower character input/output which
was the sole means of device interfacing in the thesis version of SLIM. A more extensive use of hardware interrupts and less use of polling loops, with respect to process blocking due to device control, would also improve SLIM performance.

A special performance improvement is possible under the constraint of an anchored process. Certain resource dependent processes, such as file servers and printer servers, are considered to be anchored to a machine node providing support for the resource. Migration of such processes is expected to be minimal if not nonexistent. The SLIM device interface to the machine processor, termed the loader device in previous chapters, might be specially configured to better accommodate such nonmigratory processes. A translator, converting SLIM code to the object code of the target processor, could be built into the SLIM intermediate machine layer thereby eliminating instruction interpretation for these designated processes. Thus, while a SLIM process is being loaded, automatic translation of SLIM instructions to target machine instructions could take place. Since anchored processes are not expected to migrate from the original machine, the costs of this run-time translation are small compared to the execution speed benefits. Further, note that the instructions of SLIM, being an intermediate machine, are very similar to instructions of the target machine and so, the translation process itself is expected to be straightforward. However, other issues, such as multi-processing and device management, may complicate the translation process to some extent.

The SLIM machine design includes a set of device management interfaces, which in some ways is analogous to a vector of software traps provided by the Intel 8088 and other processors. A SLIM device is similar to an asynchronous hardware process in that, although the process initiates a device operation, as the result of a combination of microcode, direct memory access (dma) and device controller, the device and process function independently. Essentially, this set of devices is user definable, although across a network of machine, some conventions must be established when process portability is a requisite. In the current design, there is a direct relationship between the logical device services provided by SLIM and the physical devices existing at the hardware base; however, a refinement of SLIM might incorporate the concept of the process alias as described below by Ravindran and Chanson, to create a uniform structure to handle state inconsistencies common in distributed systems [Ravindran and Chanson 86a].

"A process alias is an ancillary process that executes on behalf of a main process to carry out a limited, well defined function. These aliases may be created by the process itself or by the kernel. In the later case, they are transparent to the process and are known as invisible aliases. The alias executes
asynchronously to its associated process. The process on whose behalf the aliases execute is known as a corporate process." [Ravindran and Chanson 86b].

The process alias provides a mechanism by which intermediate machine can offer services different from and complementary to the logical device analogy. Application of the process alias could be in such circumstances as hard (or fatal) and soft error control where the process alias construct would be the high level manifestation of certain intermediate machine activities.

The author suggests that the above recommendations, in conjunction with the efforts of this thesis, could produce a performance oriented and highly functional distributed computing system.
References

[Agrawala and Tomlinson 76]

[Acton 85]

[Acton et al. 85]

[Bell 73]

[Berg 82]

[Bourne et al. 75]

[Boyle 82]

[Cheriton 81]

[Cheriton 84]

[Coleman 74]

[Chanson and Ravindran 86]

[Deitel 84]
[Feldman et al. 82]

[Fox 78]

[Brinch Hansen 69]

[Hayter 83]

[Lazowska et al. 81]

[Lockhart 79]

[Madnick and Donovan 74]

[Microsoft 82]

[Mock et al. 58]

[Overgaard 80]

[Peck 83]
Peck, J. "The Essence of Portable Programming" (draft), The University of British Columbia, 1983.

[Popek and Kline 79]
[Ravindran and Chanson 86a]

[Ravindran and Chanson 86b]

[Richards 71]

[Richards 77]

[Richards 77b]

[Svobodova 79]

[Tanenbaum 81]

[Walker et al. 83]

[Wang 86]

[Watson 81]