A PLATFORM FOR PROTOTYPING PIMOS SYSTEM SERVICES

A Thesis

Submitted to the Graduate School
of the University of Notre Dame
in Partial Fulfillment of the Requirements
for the Degree of

Master of Science

by

Paul W. Schermerhorn, B.A., M.A.

Vincent W. Freeh, Director

Department of Computer Science and Engineering

Notre Dame, Indiana

April 2002
A PLATFORM FOR PROTOTYPING PIMOS SYSTEM SERVICES

Abstract

by

Paul W. Schermerhorn

This paper presents the PIM Runtime Emulator, a platform for designing and testing distributed system services for the PIM Operating System (PIMOS). Developers implement services for the emulator to be ported to PIM hardware when it becomes available. The emulator executes on a cluster of Linux workstations, providing abstractions of PIM nodes and communication mechanisms in massive PIM arrays. The PIM Runtime Emulator is an effective platform for developing the unique system services essential to effectively using PIM machines, and also serves as a validation of the proposed PIMOS specification described herein.
This work is dedicated to

my wife, Alice, for her seemingly endless supply of patience.
CONTENTS

FIGURES ................................................................. v

TABLES ................................................................. vi

ACKNOWLEDGEMENTS .................................................... vii

CHAPTER 1: INTRODUCTION ............................................. 1
  1.1 Background ...................................................... 2
  1.2 The PIM Runtime Emulator ....................................... 5
  1.3 Overview ....................................................... 6

CHAPTER 2: RELATED WORK ............................................ 7
  2.1 Intelligent Memory Systems ...................................... 7
  2.2 Distributed and Parallel Systems ............................... 8
  2.3 Distributed and Parallel Programming Models ................. 11
  2.4 Other Related Work ........................................... 15

CHAPTER 3: THE PIM OPERATING SYSTEM ............................ 18
  3.1 Characteristics of PIM Machines ................................ 19
  3.2 Goals of the PIM Operating System ............................ 21
  3.3 PIM Execution Model ........................................... 23
  3.4 The PIMOS ...................................................... 26
    3.4.1 PIMOS Overview ........................................... 27
    3.4.2 The PIMOS Kernel ......................................... 29
    3.4.3 System Services ........................................... 30
  3.5 Summary ....................................................... 30

CHAPTER 4: THE PIM RUNTIME EMULATOR ........................... 31
  4.1 Overview ..................................................... 32
  4.2 Communication Processes ..................................... 33
  4.3 Node Processes ............................................... 35
    4.3.1 Objects .................................................. 36
4.3.2 Parcels ................................................. 40
4.3.3 Threads .............................................. 41
4.4 Summary .................................................. 42

CHAPTER 5: PIM SYSTEM SERVICES .................................. 44
  5.1 Name Resolution Service ..................................... 44
      5.1.1 Object Naming ....................................... 45
      5.1.2 Name Resolution ....................................... 46
      5.1.3 Resolution Implementation ............................ 48
      5.1.4 Name Caching ......................................... 49
  5.2 Object Swap ................................................ 51
  5.3 Load Monitor ............................................... 53
  5.4 Summary .................................................. 57

CHAPTER 6: CONCLUSIONS ........................................... 59
  6.1 PIM Runtime Emulator ...................................... 59
  6.2 PIM System Services ....................................... 60
  6.3 Limitations and Evaluation ................................. 61
  6.4 Future Work ............................................... 63
FIGURES

3.1 PIM Execution Model ........................................ 23
3.2 PIMOS System Structure ................................... 26
3.3 Parcel Handler ............................................. 27
3.4 PIMOS Program Execution ................................. 28

4.1 The PIM Runtime Emulator ................................. 32
4.2 Node Process ............................................... 35
4.3 p.out Object File ......................................... 37
4.4 The p.out header ......................................... 38
4.5 Parcel format ............................................. 40

5.1 Name Resolution .......................................... 47
5.2 Load Monitor Message Frequency ....................... 55
TABLES

5.1 Name Cache Performance: Total Resolve Requests .............. 49
5.2 Name Cache Performance: Maximum Resolve Requests ........... 50
ACKNOWLEDGEMENTS

I would like to thank Dr. Vincent Freeh for his guidance and assistance during the process of bringing this thesis to light. His suggestions often provided simple solutions to difficult problems.

I would also like to thank Rich Kendall and Vsevolod Panteleenko for their contributions during the formative period of this work. Many of the details of the PIMOS design were hammered out during meetings with them and Dr. Freeh.

This research was supported by NSF Career grant CCR-9876073, the HTMT Architecture Project (JPL), and the Arthur J. Schmitt Fellowship.
CHAPTER 1

INTRODUCTION

Processing in Memory (PIM) technology combines computational units with storage, creating a new computing environment. Combining logic and memory greatly increases bandwidth from the memory to the processor, and greatly reduces latency. PIM systems, therefore, provide opportunities for programmers not offered by traditional computing platforms. However, a PIM node is resource-poor relative to a desktop computer, with a small amount of memory and reduced computing power. Solving problems in this environment will require new approaches to programming, as well as substantial support from the operating system.

PIM chips have the potential to be deployed in several configurations. They may be used as coprocessors along side a main processor, or, similarly, in a client-server configuration where the PIM nodes respond to explicit requests from the main node. This research, however, focuses on the use of PIM chips in very large arrays of thousands or even millions of nodes. Effectively using thousands of PIM modules poses entirely new problems for the system developer. For example, existing approaches depend heavily on centralized global information, which is likely to prove problematic for very large PIM arrays. Efficient computing in the PIM environment requires new methods; existing models do not take advantage of the strengths of PIM, and do not address the demands imposed by PIM. Of particular interest are the services offered by the operating system. An operating system for the PIM environment
must allow users easy, efficient access to PIM resources. Many system services can be implementations of well-known mechanisms that all operating systems share. Low-level services, such as scheduling, fit into this class.

In addition to the low-level operating system services, an array of PIM nodes needs higher-level distributed services that allow PIM nodes to work together. Existing approaches (e.g., SPMD, asynchronous message passing) fail to take advantage of the power of PIM, especially because they tend to be implemented for systems with much higher communication costs and greater resources than a PIM machine. New solutions must be found in order to realize the full potential of the PIM architecture. Distributed services for PIM systems provide the system developer with an opportunity to create new and interesting approaches to existing problems; much of the interesting research on the PIM Operating System will focus on distributed services.

This thesis demonstrates that the PIM Runtime Emulator provides an effective platform on which to experiment with distributed system services, such as distributed name resolution. The emulator allows users to design efficient solutions that take advantage of the strengths of the PIM architecture. Distributed services can be debugged and evaluated on the emulator before being ported to a PIM system for final testing. This thesis describes the PIM Runtime Emulator and some distributed services that have been implemented on it.

1.1 Background

The emergence of PIM technology raises the question of how to most effectively use PIM chips. Intelligent memory systems exhibit characteristics not found in current computing environments. For this reason, the appropriate model of com-
putation for PIM is unclear. The right solution uses existing mechanisms where applicable, and employs new ones where necessary.

The PIM project [4, 13, 12] seeks to address the increasingly problematic memory bottleneck. Increases in memory speed are simply not keeping up with increases in logic speed, making it difficult to provide processors with enough data to achieve high CPU utilization. Integrating logic modules with memory macros on a single chip allows for dramatic increases in bandwidth, as well as dramatic decreases in latency.

Large numbers of PIM modules can be connected together by a network to form a PIM machine. These PIM modules perform computation on data local to themselves, and communicate with other PIM modules to accomplish the overall computational goal. One major challenge for the designer of system software is how to allow these modules to work together efficiently. Communication costs must be kept to a minimum, and mechanisms that allow users to take advantage of remote resources need to be developed.

There are three characteristics of PIM that make PIM system software distinct from other system software: PIM nodes are relatively resource-poor, they are connected in large arrays, and they are memory devices. PIM nodes have small amounts of memory (i.e., a few megabytes) and limited processing power, relative to traditional computing platforms. This means that the jobs done by individual nodes need to be small. Users must employ a more fine-grained level of problem partitioning than on current machines, and the system software must make this simple and inexpensive. Furthermore, the system software footprint must be very small so that user programs will be able to use as much of the PIM node’s limited resources as possible.
The second distinguishing characteristic of PIM systems is their high degree of parallelism. PIM machines are composed of many PIM nodes communicating via fast native instructions, so the amount of potential concurrency is quite high. However, scaling to very large PIM systems requires careful attention the the scalability of global system services over many nodes. Furthermore, the high degree of parallelism and fine-grained partitioning of problems indicates that there will be a great deal of communication between PIM nodes. System support for communication must, therefore, be efficient and unobtrusive.

The third distinguishing characteristic is that PIM chips are memory components, and need to be able to perform like memory (i.e., they need to be able to respond quickly to requests for data). Furthermore, as memory parts, PIM nodes are not directly connected to the wide variety of peripherals to which generic computers are typically connected, which requires new solutions to some old problems (e.g., swapping). Each of these three characteristics of PIM offers advantages and imposes constraints.

System software for PIM machines must address the constraints imposed by these PIM characteristics, and also must take advantage of PIM's strengths. The operating system proposed here uses objects as the basis for computation. Users create objects, which are loaded into the system. The object sets up handlers for incoming messages, or parcels. When a parcel arrives at a PIM node, it is addressed to a particular object, and specifies which handler should be invoked. Any parameters required are part of the parcel’s payload. This model of execution allows data to migrate to computation (i.e., a handler can send a request to another object for the data needed to complete the computation at hand). However, it also allows computation to migrate to data. When a program reaches a point at which it needs more data to continue, instead of bringing the data to the current node, the
program can ship the code to the node hosting the needed data, and computation
 can continue on that node. The ability to migrate computation is one of the most
 interesting aspects of the PIM environment.

1.2 The PIM Runtime Emulator

The PIM Runtime Emulator is a platform on which to experiment with PIM
operating system services. The emulator allows users to implement system services
in a more familiar development environment using C. Once the algorithms have been
worked out on the emulator, the services are ready to be ported to PIM hardware for
testing and optimization. This is especially useful now, when there is no assembler,
compiler, or even hardware for prototyping services on PIM.

The emulator implements an abstraction of the PIM Operating System proposed
as a part of this thesis. The operating system has not been implemented at this time;
the description given below represents the assumptions embodied in the emulator.
All major components of the operating system have been implemented as part of
the emulator. The emulator, therefore, also serves as a proof of concept for the
operating system proposed here. The fact that the services described below were
implemented in the emulator provides some evidence that the operating system
constructs described here can provide an effective platform for PIM computing.

The PIM Runtime Emulator allows users to create objects (code and data) and
submit those objects for execution. PIM nodes are emulated, as is the communica-
tion network connecting the nodes. The emulator executes on a cluster of Linux
machines, and is configurable with regard to the number of nodes emulated. Many
of the low-level system services (e.g., memory allocation) are implicitly provided by
the host operating system (i.e., Linux). The PIM Runtime Emulator is not intended
for experimenting with low-level operating system services, so it does not simulate a
PIM at the instruction level. Instead, the emulator concentrates on providing multi-
node emulation with communication. It is, therefore, best suited for prototyping
distributed services (e.g., name resolution).

1.3 Overview

The remainder of this thesis is organized as follows. Chapter 2 describes re-
lated work, including work on intelligent memory, distributed operating systems,
distributed programming models, and distributed services. Chapter 3 examines the
PIM execution environment, including the execution model, the role of the PIM
operating system kernel, and a brief discussion of kernel operations and system ser-
vices. Chapter 4 describes the implementation of the PIM Runtime Emulator. The
various components of the emulator are described, along with the object file format
and the handling of system calls and library calls. Chapter 5 discusses system ser-
vices in more detail, and gives examples of distributed system services implemented
on the PIM Runtime Emulator, and Chapter 6 presents conclusions.
CHAPTER 2

RELATED WORK

Computing with intelligent memory is similar in some ways to distributed computing and in other ways to parallel computing. Below is a discussion of several distributed and parallel systems that are related to the PIM Runtime Emulator. Section 2.1 discusses some projects related to intelligent memory. Section 2.2 enumerates several distributed systems, including operating systems and toolkits. Section 2.3 presents some notable programming models for distributed and parallel computing. Finally, Section 2.4 looks at a handful of other related projects.

2.1 Intelligent Memory Systems

Many projects (e.g., PIM [4, 13, 12], IRAM [22], C-RAM [5], and PPRAM [18]) have explored the possibility of processing in memory. PIM is a response to the growing gap between processor and memory performance. Delays for memory accesses are a limiting factor in system performance. Combining logic and memory helps to overcome the performance gap by allowing increased bandwidth and reduced latency. The PIM project is focused in part on using a very large number of PIM chips concurrently to achieve petaflops level performance.

Active Pages [19] is a computation model that shifts some computations to memory in response to the performance gap between processors and memory. Users must partition applications between the processor and the memory. To perform
computation in memory, users bind functions to pages or groups of pages. These bound functions are started by memory-mapped writes to user-defined areas in each page, and their outputs are read from other user-defined areas. Thus, the programming model is a coprocessor.

ActiveOS [20] is an operating system designed to allow users to execute Active Pages workloads in a multi-process environment. ActiveOS provides three categories of mechanisms specific to Active Pages: process services, interpage communication, and virtual memory. Process services allow users to allocate active pages, bind functions to active pages, and synchronize computation within active pages. The master processor mediates communication with other active pages via an interrupt to the operating system. Finally, ActiveOS provides a virtual memory mechanism that acknowledges the active nature of the memory (e.g., by periodically bringing pages into memory so that they can execute).

2.2 Distributed and Parallel Systems

The advent of inexpensive yet efficient networking hardware has made it possible to design systems specifically for distributed environments. Tanenbaum [25] enumerates several advantages to distributed systems. First, it may be possible in many cases to create a cluster of very inexpensive workstations (e.g., Beowulf [24]) that outperforms more expensive supercomputers. Second, some systems are inherently distributed, such as retail chain inventory control systems. Distributed systems can also enjoy greater reliability; a single point of failure in a centralized system makes such a failure catastrophic, whereas a distributed system may be able to recover in certain circumstances. Finally, it is easier to grow a distributed system incrementally as need arises than it is to upgrade a single centralized system. To take advantage of these benefits of distributed systems, well-designed systems software
will be needed. This section examines several projects that attempt to address this need. None of these systems is in widespread use at this time.

Globe [27] seeks to build an infrastructure upon which wide-area applications can be built. It provides support for object-specific policies using a model of *distributed shared objects*. Each object can be physically distributed (i.e., its state may be distributed over several machines), and is completely self-contained (i.e., encapsulates its own policies). Globe's objects allow users to design programs that take advantage of computing resources on several machines.

Legion [10] can share and manage resources that are distributed over multiple platforms, each with its own administrative, support, and security policies. It seeks to provide a solution to this problem that is better than the patched-together solutions commonly used to solve it. Legion uses object-based techniques to manage complexity. Programmers are provided a global system image, a universal shared name space. Legion is designed to provide three key features: scalability, user-definable security mechanisms, and mechanisms for fault tolerance.

Amoeba [26] is a distributed operating system whose goal is to provide a transparently distributed system. Users are not aware that the system is distributed. Amoeba is object-based, and uses capabilities for protection. Associated with each object is a server process that responds to RPCs by operating on the object. Amoeba is essentially a microkernel, handling only communication and some process management. Other system resources (e.g., memory, files, directory services) are managed by servers.

The Globus [6] project is a toolkit for creating metacomputing infrastructures. Metacomputing is defined as networked virtual supercomputing using geographically distributed resources. The Globus toolkit is a platform with which metacomputing resources can be constructed. Globus provides infrastructure for resource location
and allocation, communication, real-time resource information distribution, authentication, process creation, and data access. The collection of these services defines a *metacomputing virtual machine*.

Sprite [21] is a distributed operating system designed to capitalize on the availability of networked multiprocessor machines with large memory. Sprite is a multithreaded kernel that provides an RPC mechanism upon which many of the distributed services are built. Sprite’s file system is transparently distributed, so users cannot tell whether the files they are accessing are stored locally or remotely. The file system takes advantage of large memory by caching on both client and server machines. The virtual memory system uses ordinary files for backing storage, and negotiates with the file system for available memory, attempting to find an optimal balance for the current load. Sprite migrates processes to balance the load. Sprite’s migration mechanism is transparent: processes do not know they have been migrated.

The HYDRA kernel [33] is designed to take advantage of high-performance multiprocessor hardware. HYDRA defines three basic object types: *procedure*, *LNS*, and *process*. A HYDRA procedure is similar in many ways to the normal concept of a procedure. They can receive parameters and return values. However, HYDRA procedures also include protection facilities, in the form of *capabilities*. Capabilities are object references combined with a set of access rights that define what objects the procedure has access to and what the procedure can do to those objects. When a procedure is invoked, an local name space (LNS) is created. The LNS is a list of the capabilities inherently associated with the procedure (caller-independent capabilities) and those contributed by the caller (caller-dependent capabilities). The LNS of a procedure invocation defines the set of objects to which the procedure has access during the current invocation. Finally, a HYDRA process is a stack of
LNSs that represent the current state of a task. HYDRA procedures and procedure invocations are sequential in nature, while many processes may be executing concurrently. These basic object types are the basis of HYDRA’s support for multiprocessor hardware, and, in turn, for an operating system built around the HYDRA kernel.

Mach [3] is a microkernel developed to provide a base for other operating systems, to provide transparent access to network resources, and to exploit multiprocessor and multicomputer hardware configurations. Mach is a microkernel, and as such separates the control of basic hardware resources from the control of the operating environment (e.g., process and memory management, communication). Mach treats operating systems as services that can execute on top of the Mach kernel. Support for multiprocessor platforms is provided by the C Threads library, a user-level interface to Mach’s kernel threads. The kernel transparently maps Mach abstractions (e.g., process creation) onto distributed hardware to provide support for multicomputer environments. Mach abstractions are location transparent, and can, therefore, be relocated by the Mach kernel without intervention on the part of the user or the client operating system.

2.3 Distributed and Parallel Programming Models

There are three basic models for distributed and parallel computing: message passing, remote procedure call (RPC), and shared variables. Shared variables are typically used for parallel computing in a very tightly coupled environment. Concurrently executing processes communicate and synchronize by writing and reading shared data such as semaphores, locks, or monitors. Message passing protocols (e.g., MPI [14] and PVM [8]) make sharing explicit by requiring processes to send messages to other concurrent processes, rather than simply write to an address that is
visible to all processes. RPC protocols allow processes to execute code remotely in the form of procedure calls. CORBA and DCOM (both described below) are examples of RPC protocols.

OMG’s Common Object Request Broker Architecture (CORBA) [28, 29] is the key component of their Object Management Architecture (OMA). OMA’s goal is to provide standardized means by which developers can create “interoperable distributed object systems in heterogeneous environments.” CORBA is the part of OMA that allows these distributed objects to communicate and interact. CORBA consists of:

- the ORB core, which handles communication requests between objects transparently, hiding object location, implementation, execution state, and communication mechanisms;

- the IDL, which is a declarative language with which developers specify interfaces for their objects, and language mappings, which map IDL specifications onto programming languages, including C, C++, Smalltalk, and Ada 95;

- an interface repository, which allows object interfaces to be added at runtime;

- stubs and skeletons, which are (respectively) the client- and server-side mechanisms for issuing and delivering object requests, and are translated directly from IDL specifications;

- dynamic invocation and dispatch, which are like generic stubs and skeletons, allowing objects access to other objects whose interfaces are not known at compile-time, and object adapters, which are responsible for making sure that requests and responses are translated into the format that the recipient is expecting.
CORBA 3.0 includes three significant features not found in previous versions: the Portable Object Adapter (POA), CORBA Messaging, and Objects by Value. The POA provides a standard object adapter that allows CORBA applications a greater degree of portability. The CORBA Messaging Specification now includes support for asynchronous messaging, as well as quality-of-service guarantees. Finally, the Objects by Value facility allows applications to pass objects by value. Previously, object references were passed, requiring applications to perform remote invocations over the network.

Microsoft’s Component Object Model (COM) [32] allows components from different sources to be combined to create applications and systems. COM defines a programming language-independent binary standard for interoperability between components. Components are binary executables that can provide services to other programs, whether they are provided by the same vendor or not. COM provides mechanisms for interprocess communication, shared memory management, error reporting, and dynamically loadable components. COM applications interact through *interfaces*, collections of functions that define expected behaviors and responsibilities. The Distributed Component Object Model (DCOM) [15] is an extension of COM. Whereas COM component applications communicate via the COM interprocess communication facility, DCOM components communicate via a network protocol. DCOM hides the location of components, so the invocation of a component’s methods is identical whether the client is on the same machine as the component or not. Connections between clients and components are managed by DCOM. Reference counts are maintained to ensure that components are not freed while clients are connected, and DCOM uses a pinging protocol to detect when connections have been broken (e.g., due to hardware failure). DCOM also provides transparent distributed garbage collection.
Active Messages [31] seeks to reduce communication overhead and increase overlapping of communication and computation, thereby enhancing the effectiveness of message passing machines. An active message contains the address of a message handler to be executed on the target machine. The programming model is, thus, SPMD. The handler takes the message from the communication layer and integrates it into the computation, executing for a short time, and to completion. Active Messages eliminates receive buffering. When a message is received, data contained is copied immediately into already allocated memory, and simple requests are responded to without delay. Normal send/receive models require extensive buffering and context switches to remove data from the network; this is where Active Messages’ advantage lies.

The Actors [1] programming model seeks to provide flexible support for both parallel and distributed computing. Actors are autonomous objects. They encapsulate a thread of control as well as data and methods. Actors communicate via messages. Each message initiates execution of a specified method with arguments specified by the message payload. Actors employ globally unique names, called mail addresses, and they may only send messages to other actors whose mail addresses they know. Actors provide asynchronous point-to-point communication, and leave more complex communication schemes (e.g., RPC) to the user.

Linda [9] is a distributed programming model that utilizes generative communication to allow networked computers to work together. Generative communication is an example of associative memory; users create messages called tuples and place them in a tuple-space to be retrieved by some cooperating process. This form of communication is said to be generative because the tuples remain in tuple-space until they are removed regardless of the execution state of the process that places them there. Linda uses out, in, and read statements to place tuples into tuple-
space and remove or inspect them from tuple-space. The eval primitive creates concurrent processes to perform a specified task. Communication in Linda is completely orthogonal; a receiving process has no information about who generated a tuple, and a sending process has no information about who will retrieve it from tuple-space. Linda provides a distributed dynamic global namespace to support this communication orthogonality. Tuples are buffered by their generating process until they are removed from tuple-space or the process exits, at which point the Linda virtual machine takes over the task.

2.4 Other Related Work

The Domain Name System (DNS) [16, 17] is a distributed database that provides a mapping between host names and IP addresses. Domain names are hierarchical. There are several top-level domains, and all other names are branches off of these top-level names. Organizations maintain second-level domains that contain naming information for their hosts. There may also be third- and lower-level domains. User applications query a local name server, which returns a translation if the name is a part of the local domain. Otherwise the name server does a remote lookup. This is achieved by querying the top-level domain server for the address, which returns the address for the appropriate second-level domain server. This process is continued, moving further down the hierarchy, until the mapping is found. The local name server then returns the IP address to the application, which can then send the message. This scheme avoids the danger of having a single point of failure in a centralized scheme, but there is the potential for bottlenecks, with a few very often accessed points in the hierarchy.

The Network File System (NFS) [23] allows clients to transparently access files on remote servers. The NFS protocol provides a file system model to which clients
and servers map local file system actions. The NFS protocol is stateless. The server does not maintain any information about requests once they have been serviced. There is no concept on the server side of an open file associated with a particular client; instead, clients must send a full description of the file for each read or write operation. NFS clients, however, do maintain state. When users modify files, the client side holds the data in a local cache until it is flushed to the server. The stateless design of NFS simplifies crash recovery on the server side. NFS volumes can be mounted at any point on a client machine. Thus, users may have different views of shared volumes depending on the machine they are using.

The Andrew File System (AFS) [11] is a distributed file system that provides a fully transparent view of a global filesystem. AFS volumes are mounted in a directory structure that is consistent across all hosts, providing users with location-independent transparency. AFS servers maintain information about client state. This allows AFS to provide an approximation of standard Unix file semantics. AFS uses a mechanism called callbacks to inform clients when data they have cached is changed by other client machines. Security is also enhanced by AFS via more flexible file access permissions.

The Virtual Interface Architecture (VIA) [30] reduces communication inefficiencies by providing processes with direct, protected access to network hardware, thereby reducing expensive calls or traps to the operating system. There are four basic components of VIA: Virtual Interfaces (VIs), Completion Queues, VI Providers, and VI Consumers. A VI Provider is comprised of network hardware and kernel code that manages that hardware. VI Consumers are application programs that access the network through a Virtual Interface. A Virtual Interface, in turn, is the mechanism that allows a VI Consumer direct access to a VI Provider. Each VI is composed of two Work Queues, the send queue and receive queue, into which
VI Consumers post requests to send or receive data. VI Providers asynchronously process these requests and mark them as completed. VI Consumers can establish Completion Queues to which notification of completed requests can be directed. Notifications can be placed on the Completion Queue without an interrupt, allowing VI Consumers to synchronize on a completion without a system call or trap. VIA requires hardware support from the network interface and operating system support.
CHAPTER 3

THE PIM OPERATING SYSTEM

The focus of this research is an operating system and services for machines consisting of very large arrays of PIM chips. These systems differ significantly from traditional computing platforms, and, therefore, require a different operating system in order to take advantage of their characteristic properties. In many ways, each PIM node is like a stand-alone machine. With its own processor and memory, each PIM node executes an independent instruction stream. A group of PIM nodes is, therefore, similar in a lot of ways to a group of machines working together as a cluster. There are many reasons, however, that a traditional operating system, even with clustering middleware attached, will not make effective use of PIM hardware. This chapter outlines many of the characteristics of PIM that are distinctive, and suggests ways to take advantage of those characteristics.

After a brief description of the relevant characteristics of PIM machines, the goals of the PIM Operating System (PIMOS) are outlined. After that is a description of the PIM execution model, followed by a discussion of the organization of the PIMOS. These sections compose the outline of a specification for the PIMOS; the operating system has not been implemented as a part of this thesis, although parts of it are emulated by the PIM Runtime Emulator, described in Chapter 4.
3.1 Characteristics of PIM Machines

The approach to PIM computing described here views PIM machines as more closely related to workstation clusters than to traditional SMP machines because PIM nodes are used more like autonomous machines than like processors in a shared-memory context. SMP machines typically have one operating system that manages all nodes in the machine, whereas each PIM node executes its own operating system. Thus, PIM nodes do not have access to information about the state of other nodes, unlike SMP nodes, which can examine the array of parameters that the operating system keeps for each node (e.g., the run queue). PIM nodes also manage their own memory, and communicate with other nodes by explicit message passing rather than by reads and writes to shared memory. The many similarities between PIM machines and workstation clusters make clusters an ideal point of contrast to highlight the distinctive features of PIM machines.

Probably the most obvious difference between PIM machines and traditional clusters is that PIM nodes are memory modules. This allows users the flexibility to use PIM chips in a variety of configurations (e.g., coprocessor, client-server). However, because they are memory parts, PIM nodes have to be able to exist in traditional machines and act as memory does (i.e., respond quickly to off-chip memory requests). Although a PIM may receive remote memory requests, indiscriminate off-chip memory accesses must not be allowed. The memory access policy must ensure that the memory on which programs are working is not unexpectedly changed. The PIM Operating System employs a very fast request servicing mechanism that can dispatch memory requests with very low overhead while still enforcing safe memory accesses.

PIM nodes also have very efficient communication mechanisms. Latency is lower in PIM systems than in clusters of PCs, due to the tightly integrated nature of PIM
machines; PIM nodes are all part of a single machine, rather than standalone machines connected by a post hoc network. The wide memory bus connecting PIM nodes also provides greater memory bandwidth than many traditional cluster solutions. Efficient native communication instructions allow programs to take full advantage of the fast memory bus. The efficiency of communication makes PIM systems better targets for fine-grained distributed computation than their traditional cluster counterparts.

The third difference is that PIM nodes are relatively resource-poor. The memory per PIM chip is significantly less than what is found in conventional machines. The operating system needs to keep its memory footprint small so that the largest possible portion of PIM resources is available to applications. In addition, PIM nodes do not have access to the wide array of hardware devices typically available to workstations. This is advantageous because it means that the operating system does not have to support a lot of different devices, however, it also makes many services more difficult to provide for PIM nodes than for workstations. For example, virtual memory normally swaps to and from a disk. PIM nodes are not connected to a disk, so some other mechanism must be designed. A simple mechanism that swaps to another PIM node is described in Chapter 5.

Finally, PIM machines are grouped together in very large numbers. PIM machines, therefore, provide greater opportunity for concurrency and distributed computing than their traditional counterparts. The distributed nature of PIM computing makes careful attention to the efficiency of distributed services essential. Global system information is impractical in such a highly parallel system. Distributed information services, such as name resolution, are important components of an efficient PIM operating system. Scalability of system services will be an important goal of PIM system programmers.
3.2 Goals of the PIM Operating System

The major characteristics of PIM machines impose constraints on system software. The following are goals for the PIM Operating System:

- Fast service for “dumb” memory requests,
- Efficient mechanisms for “smart” memory computing,
- Support for implicit operations,
- Scalability,
- Autonomy of individual nodes, and
- Location anonymity for objects.

PIM nodes must have the ability to act as “dumb” memory nodes. That is, they need to be able to service off-chip read requests in a time frame that is short enough to appear as though the memory is no different than any normal memory part. A very fast request handler will be needed to meet this goal.

Conversely, there must be mechanisms in place that allow users to make use of the “smart” features of the PIM node. Users must be able to create programs that efficiently make use of the computing power available in an array of PIM nodes to perform operations on data in the memory. The PIM execution model addresses this need, along with many distributed system services.

The PIM Operating System provides an opportunity for system software developers to create new types of services, including implicit ones. Implicit services execute in the background, providing important information and functionality without requiring direct interaction by the user. Examples include load balancing and thread migration.
PIM machines are very large arrays of PIM nodes. In this context, scalability of system services is of extreme importance (i.e., services must continue to perform well even when the number of nodes in the system is very high). It is infeasible, for example, for thousands of PIM nodes to converge on a single information server at the same time. Scalable, distributed mechanisms for information sharing must be worked out. These may include mechanisms for accessing information distributed across the system, or mechanisms that distribute close approximations of global information to nodes implicitly.

Although PIM nodes are parts of a larger system, each PIM node is itself a self-contained computer. Each node must, therefore, be allowed to operate autonomously. Nodes must be able to manage their own memory and computational resources while remaining tightly integrated into the overall system. Achieving a working balance between these demands remains an important challenge for PIM system services.

Finally, PIM objects must have location anonymity. Resources must not be tied in any way to an individual PIM node. References to PIM objects must be via their object names, not via any hardware address. If the operating system allows objects to be accessed via hardware addresses, PIM nodes will no longer have the freedom to manipulate resources to make efficient use of limited memory. This would violate the autonomy goal.

Any system that attempts to allow efficient computation in the PIM environment must meet each of these goals. Failing to achieve any of them would result in a system that does not realize the full potential of PIM systems. The execution model and operating system specification for PIM systems have been designed with these goals in mind.
3.3 PIM Execution Model

PIM computation is object-based. Users define objects that operate on data and communicate with each other to accomplish tasks. A set of objects that cooperates comprises a program. Cooperation takes the form of object member function invocations, either to move data to a requesting node or to have an object on another node perform some procedure on data local to it.

PIM objects consist of data and code that acts upon that data. The object member functions are invoked by *parcels*, messages that are addressed to objects to invoke specific member functions. Each member function executes in an independent thread; there may be many threads concurrently active in an object’s address space. Once the parcel has been handled (i.e., the member function has completed execution), the thread spawned for that parcel is deleted.

A PIM program is a collection of objects cooperating to achieve some goal. These objects may be located on several different PIM nodes, and there may be many concurrently executing threads at any point in time. Program execution begins when a program’s initial objects are instantiated. The system instantiates
an object by allocating space and (optionally) executing a user-defined constructor. There is no thread of execution associated with an object after the constructor has been executed; all computation (aside from the constructor) is delayed until a parcel handler is invoked. Figure 3.1(a) depicts the arrival of a parcel requesting the creation of a new object. The constructor for an object can set up handlers for different message types (Figure 3.1(b)), and there are predefined handlers set up by the system for primitive operations such as read and write. Thus, the simplest object is a block of data with only the system-provided read and write handlers. An object is passive (i.e., has no thread of execution associated with it) until it receives a message that invokes one of its handlers (Figure 3.1(c)).

Communication in the PIM environment consists of parcels that are sent between objects. Every parcel is addressed to some object, and invokes some handler on that object. An object may process more than one parcel concurrently, and therefore may have more than one handler thread active at the same time. Handlers are responsible for any necessary synchronization. To invoke a handler, a parcel is sent to the target object. The parcel has a handler selector and a payload that can contain any arguments the handler needs to execute. Some parcels, including “dumb” read requests, require immediate service, and are handled directly by the operating system, while others are placed on a run queue to be executed according to the scheduling algorithm implemented by the operating system.

There are two levels at which the concept of an address space may be applied in the PIMOS. At a low level, each thread executes within its own address space consisting of its object and the stack allocated by the system. This version of an address space is essentially the traditional concept, and multiple threads executing on one object share everything other than their stacks. Within a thread’s address space, the conventional load-store memory model applies; parcel handlers have direct
access to data encapsulated within an object, making computation at this level very similar to computing in conventional systems.

At a high level, the address space of a program includes all of its constituent objects. Objects executing as part of a program access one another via the parcel interface. Protection at this level is minimal; if a program knows the name of an object, it can invoke the object’s parcel handlers even if it is not part of the object’s address space. This scheme is sufficient in the PIM environment where malicious users are not expected. However, it is possible for users to implement their own protection by, for example, requiring a key to be passed along with the parcel.

Cooperation between nodes is key to achieving non-trivial goals in PIM systems. There are two basic mechanisms necessary for nodes to work together: data migration and computation migration. Data migration tends to view remote PIM nodes more like memory than processors. When the currently executing handler comes to a point at which it cannot continue with the data located locally (i.e., part of the current object), it sends requests to one or more other objects to retrieve the data it needs. These external objects, whether they are located on remote PIM nodes or the local node, send responses to the requesting object containing the data it needs to continue execution. Data migration gathers the data to the computation.

The other mechanism of cooperation, computation migration, views remote PIM nodes more as processors than as memory parts. Using this model, when the current handler is finished processing its own data, it passes the computation to the next object, rather than requesting its data. Computation migration typically involves packaging up the state of computation on the current node as the payload of a handler invocation and sending it to the node hosting the data required to continue computation. In many cases, computation migration is more efficient than data
migration, because the amount of data sent (i.e., the interim results) is less than the total amount of data used to solve the problem.

3.4 The PIMOS

The PIM environment clearly places different demands on the system software than normal cluster environments. The PIMOS must be compact, because it consumes part of the memory of the PIM node, and the PIM node must behave as a memory part. It must have some fast mechanism for dealing with generic reads and writes, so that PIM chips can look to host processors like normal memory chips. It must provide fast communication with low overhead, and, finally, it must provide a variety of distributed system services.
3.4.1 PIMOS Overview

The basic structure of a system running the PIMOS is presented in Figure 3.2. The PIMOS kernel provides an interface to the PIM hardware via a small set of system calls. The operating system also provides several high-level system services (i.e., services that, while not required to use the system, make the system easier to program and use), such as name resolution and load balancing. Finally, a set of library calls are provided as part of the system, either to be compiled into objects or to be linked dynamically at run-time.

A program executing on a PIM machine is a collection of objects and their associated handlers. Figure 3.2 shows three programs in execution, two with two objects, and one with three. The objects that compose a process need not all exist on a single node, although individual objects cannot span nodes. Also, more than one instance of a given program may be active at a time. PIMOS provides a global name service, with distinct names for any new objects created. Thus, requests intended for objects in a newer invocation of a program always go to objects created by that invocation, not to those created by other invocations.
Each object can have zero or more threads associated with it. These threads are executions of parcel handlers. Figure 3.3 shows the relation between a parcel handler and the system. Parcel handlers execute user-level instructions, kernel calls, system-provided services, and system-provided library calls. Kernel calls typically involve some sort of hardware access (e.g., communication). System-provided services are the high-level services that the PIMOS provides, and are typically distributed services. In many cases, system services are invoked by the system on behalf of user objects (e.g., name resolution is implicitly invoked when a user invokes `send`). The system also provides library calls, which are low-level calls that often involve manipulation of data (e.g., name concatenation).
Figure 3.4 depicts an instance of program execution in PIMOS. There are several programs (i.e., collections of cooperating objects) in secondary storage. Each program may be instantiated more than once at the same time. Within an instantiated program, there are several instantiated objects, and for each object type, there may be several instantiations of that type within the same program (e.g., a list object would be instantiated several times when building a linked list). Each instantiated object may host multiple parcel handler invocations concurrently; Figure 3.4 shows an case in which two instances of the same parcel handler execute concurrently.

There are no plans at this time to allow objects to span more than one node. Allowing multi-node objects would violate the requirement of autonomy discussed above; a parcel handler on such an object would need direct access to data stored on remote nodes. This is because accessing a remote portion of the current object would not be via parcel handlers, since a portion of an object would share a name with the whole object, making it impossible to address a sub-object via a parcel. In the programming model proposed here, child objects (or some other configuration consisting of multiple objects) are the way to deal with data needing to span nodes.

3.4.2 The PIMOS Kernel

The kernel of PIMOS is itself an object, and is accessed in the same way that a normal object is accessed. Thus, a system call is actually a parcel handler invocation for the system object. The kernel provides system calls for object management (e.g., create, handle, destroy), parcel dispatch (i.e., invoke), and other low-level operations. Because of the limited amount of hardware support required, the operating system is very compact.
3.4.3 System Services

The PIMOS also provides several services to the PIM user. Services are typically more complex than system calls, and require more time to complete execution. As such, they are often placed on the run queue to execute alongside user code (i.e., these services must share CPU time with normal user code, and are not afforded any greater share of resources than other threads executing on the node). Although they are not part of the kernel, many system services (e.g., name resolution) are indispensable for effective computing on PIM systems. System services are discussed in more detail in Chapter 5.

3.5 Summary

The PIM Operating System is shaped by several unique characteristics of PIM systems. PIM nodes are memory parts, they are resource-poor in both storage and processing power, and they are connected in very large arrays. Users program PIM machines by creating objects, setting up handlers to processes different types of requests, and invoking those handlers by sending parcels. The PIMOS kernel is an object like any other (except that it is privileged), and system calls are handler invocations on the PIMOS kernel object. PIMOS also provides several system services, including name resolution and object migration. This unique computation environment allows users to develop efficient distributed applications for PIM systems.
CHAPTER 4

THE PIM RUNTIME EMULATOR

The PIM Runtime Emulator facilitates the construction of distributed system services for PIMOS by providing a platform upon which developers can prototype services before implementing them on an actual PIM. The goal of the emulator is to model those characteristics of the PIM environment most relevant to implementing distributed system services. As such, it provides abstractions for multiple nodes that can each execute several threads concurrently and that communicate with each other via a message passing protocol. It does not, however, attempt to provide a low-level simulation of the PIM architecture; that task is left to other projects. Many details of computing on PIM chips have also been abstracted away (e.g., the details of message format and delivery). In most cases, this is not only to reduce the complexity of the emulator, but also because the details have not been determined at this time. The fact that the emulator ignores some of the details of implementation means that services that are prototyped on the emulator will need to be implemented and tested on a PIM platform when one becomes available.

This chapter describes the implementation of the PIM Runtime Emulator and describes how it is used to prototype system services for PIM systems. Section 4.2 discusses the emulator’s abstraction of a PIM machine’s physical network. Section 4.3 details the emulation of individual PIM nodes, including the object file
format, parcel handling mechanisms, and the thread subsystem implemented in the emulator.

4.1 Overview

Figure 4.1 gives a high-level view of the structure of the emulator. It runs on a cluster of Linux workstations, each of which is referred to as a physical machine. The machine being emulated consists mainly of PIM nodes and a communication layer (probably a memory bus), along with some backing store available to at least one PIM node. Linux processes are used to emulate the virtual PIM machine, including PIM nodes and communication. Each node in the PIM machine is represented by a single process. These node processes perform most of the tasks normally handled by an operating system, including scheduling, I/O, and resource management. Node processes are discussed in Section 4.3.

The communication layer of the PIM virtual machine is also represented by Linux processes. All communication is routed through a communication process executing on each physical machine. When a user program sends a message, the node process
sends the message to the communication process to be routed to the node on which
the destination resides. The communication process manages all information related
to the location (i.e., physical machine) at which a node resides, thus freeing the node
processes to concentrate on providing the system services needed to execute PIM
programs. Section 4.2 explains the design of communication processes.

The emulator executes code contained in emulator object files. The object file
format created for the emulator ($p$.out) provides a small feature set designed to
allow easy manipulation of objects (e.g., object migration). Users create $p$.out files
by writing parcel handlers in C and compiling them with $pcc$, a utility that assembles
the C functions and header information into executables. Section 4.3.1 describes the
$p$.out file format in detail.

Objects compute by executing member functions to handle incoming parcels.
Each parcel handler executes in its own thread within the node process. The user-
level threads package developed for the emulator provides preemptive scheduling,
and is detailed in Section 4.3.3.

4.2 Communication Processes

Communication processes abstract the details of PIM communication. There is
one communication process executing on each Linux workstation. Any communica-
tion between node processes goes through a communication process. The commu-
nication process manages the details of network routing, allowing node processes to
ignore the details of how to make sure a parcel arrives at its destination.

Communication processes execute as event-driven loops. At the top of the loop,
the process blocks on a select on two sockets, an INET socket for messages from
communication processes on other physical machines, and a UNIX socket for mes-
sages from node processes on the local physical machine. The communication pro-
cess is awakened when a message arrives on either socket. After it determines the parcel's destination, it sends the parcel and returns to the top of the loop. Communication processes are able to execute synchronously in a simple loop because they receive and immediately forward messages, and are not subject to scheduling delays present in other processes (e.g., node processes).

Node processes address parcels to logical PIM nodes, and send them to their local communication process. Communication processes keep tables of UNIX domain addresses for communication with local node processes, as well as tables of INET domain addresses for communication between node processes on distinct physical machines. Two sockets (one UNIX datagram and one INET datagram) are monitored for communication by each communication process.

When a message is received from a local node process (i.e., on the UNIX datagram socket), the communication process first determines whether the destination node process is executing locally, or on another physical machine. This is accomplished by a simple modulo function applied to the total number of nodes and the number of physical machines in the current emulation run (a table lookup could be employed instead to allow more flexibility in the emulator's configuration). If the destination is local (i.e., a node process on the current physical machine), the parcel is forwarded to the address for that logical node in the local node table. If the destination of the parcel is on a remote physical machine, the communication process forwards the parcel to the communication process on that machine. Node processes are distributed across physical machines in uniform blocks, so it is easy to determine which block contains the logical address. The block in which a node resides is used as an index into the INET lookup table, making it simple for the communication process to find the address of the machine on which the destination resides.
When a message is received from a remote communication process (i.e., on the INET datagram socket), it is checked to ensure that it has reached the correct physical machine, and is forwarded to the correct machine if necessary. If the destination is on the current physical machine, the communication process looks up the address of the logical node in the local node table and forwards the message to the node process representing that logical node.

4.3 Node Processes

The emulator uses node processes to represent PIM nodes (Figure 4.2). The number of logical nodes emulated on each physical machine is configurable at startup, although that number is currently constrained by a Linux kernel limit on the number
of user processes available. Node processes allow users to load executable p.out files. Once loaded, objects can send and receive parcels and perform computation in the form of parcel handlers. Each parcel handler executes in its own thread. Scheduling is round-robin, with an adjustable time quantum and a single priority level.

Unlike communication processes, node processes employ interrupt-based message handling. This allows nodes to easily suspend user threads while processing newly arrived messages. Messages arrive on only one socket, always from the communication process executing on the local physical machine. While there is only one communication process per physical machine, there may be hundreds of node processes, depending on the desired configuration of the virtual PIM machine.

Much of what a node process does is to take care of a lot of the details that operating systems normally need to manage. Node processes provide scheduling, memory management, and system calls for user programs, allowing them to concentrate on implementing algorithms for system services. A node process is an abstraction of the operating system and the hardware for an individual PIM. With the low-level details of the PIM Operating System abstracted, node processes deal mostly with three types of things: objects, parcels, and threads.

4.3.1 Objects

The PIM Runtime Emulator is object-based. All computation is accomplished via object member function invocations. There are two views of objects, either as static object files or as active objects loaded into memory and potentially executing code. Static object (p.out) files consist of code, data, and header information. Users create object member functions in C. Member functions take two arguments, a pointer to the current object (similar to a C++ “this” pointer), and a parcel. Each function is compiled individually by gcc. The target file type is flat x86 binary (i.e.,
there is no ELF or a.out header information in the target file). The code for the function starts at the first byte of the target file.

Once all of the member functions for an object have been compiled by gcc, the p.out compiler (pcc) is invoked to create a p.out object file (Figure 4.3). The compiler just collates the output files, simply placing a header at the beginning of the file and concatenating member function files and data files after the header. It then computes addresses for the header and fills them in. Figure 4.4 shows those portions of the header that are of interest to the compiler. The header contains information about the code segment (i.e., where it starts and how it is divided up), the location and size of the data segment, and pointers to system-defined functions for sending parcels and invoking library calls. At compile-time, the addresses in the
typedef struct object {
    int size;  // Total size of the object file
    void *data;  // Address of the data segment
    int dataSize;  // The size of the data segment
    int (*pSend)(message *);  // Pointer to the send syscall
    int (*pLib)(enum libcall, ...);  // Library call interface
    int numMethods;  // Number of member functions
    char *methods;  // Address of code segment
    int *methodSize;  // Table of member function sizes
} pout;

Figure 4.4. The p.out header

header are all relative to the top of the object. When the object is loaded, these addresses are adjusted to reflect the absolute position of the object in memory.

Member functions can make three types of calls: system calls, library calls, and intra-object member function calls (inter-object calls are, of course, parcels). System calls in the emulator are similar to system calls in other operating systems. They provide users with protected access to hardware. Examples include send and create. System calls are implemented as member functions of the kernel object. User programs invoke system calls by sending a parcel to the kernel object. When an object is loaded, a pointer to the send system call is initialized. Send is, therefore, an exception: it is invoked directly rather than as a parcel handler, and is, therefore, more fundamental than other system calls. Member functions can then access the send system call through the “this” pointer.

Library calls are similar to system calls, but are typically higher-level routines that would normally be built on top of system calls. They include write to console (in this case it is the Linux console that is written to), and functions that manipulate object names. A distinct calling mechanism was desired for library calls for two reasons: first, to emphasize that they are calls of a very different sort than system
calls, and second, so that they could be invoked directly instead of requiring users to create, initialize, and send a parcel for simple tasks such as printing debugging messages to the console. Library calls are accessed via a pointer in the object header similar to system calls. All library calls share the same function pointer, with the first argument being a selector to determine which library call to execute. Library code is also a part of the emulator.

The third type of call is an intra-object member function call. Member functions can directly invoke the member functions of the current object via the “this” pointer. This ability allows users to bypass the potentially expensive process of executing a send system call to one’s own object to invoke a peer member function.

Objects are loaded into the virtual machine via the create system call. When create is invoked, the object is read from the disk. A small amount of relocation is done (i.e., computing addresses for member function pointers and the data segment, setting pointers for system calls and the library call wrapper, etc.), and information about the object is stored in system tables, including the object’s parent information and information about child objects, if any. At this point the object’s constructor can be scheduled for execution and the object is ready to begin computation.

The design of the p.out file format is shaped by the goals of making objects small and easily relocatable. Compiling functions to raw binary code with no header information minimizes their size. The emulator has no need for many of the features of a conventional file format, and can easily create a header containing only the information it needs. Keeping library code in the emulator allows users access to those routines without having to compile them into the object file. Accessing data and member functions via the “this” pointer allows objects to be moved easily without a great deal of address relocation within the object. This system is a
typedef struct parcel {
    int node;          // Destination node
    char destName[NAMELEN]; // Destination object
    int originalID;    // Source node
    char originalName[NAMELEN]; // Source object
    int method;        // The member function to be invoked
    int size;          // The size of the parcel
} parcel;

Figure 4.5. Parcel format

prototype; in the final version of PIMOS, a compiler will easily be able to compute
targets, thereby removing this requirement.

4.3.2 Parcels

Parcels are sent by object member functions to communicate with other objects
or to invoke member functions on those objects. The format of the parcel header is
shown in Figure 4.5. The header contains information about the nodes and objects
to and from which the parcel is going, along with the size of the parcel and which
member function to invoke. Parcels are represented by UDP datagrams that are
allowed to be arbitrarily large, unlike parcels in the final PIM product.

There are two different types of member functions: those supplied by the user
and the default member functions supplied by the system. The built-in member
functions are needed because PIM nodes have to be able to respond as memory
units. They are fast reads and writes that are serviced immediately by the kernel
without passing control to the object. Having the kernel handle reads and writes
allows these basic operations to be accomplished very quickly and with low overhead,
making it possible for reads and writes to be serviced very quickly. Fast reads and
writes enable the PIM to present itself as a normal memory unit when necessary.
Built-in member functions are part of the kernel and are not present in p.out files.
Parcel headers and handling are both areas in which the decision has been made
to abstract the details away for the purposes of the emulator. In a real PIM envi-
ronment, parcels will probably be unable to include all of the information detailed
in Figure 4.5 because of space considerations. It is likely that some communication
stack will need to be implemented that will break parcels into hardware packets to
send and reassemble them to receive. This level of parcel handling is beyond the
scope of the emulator as it is currently defined, but will need to be addressed in the
future if the emulator is expected to provide realistic emulation of parcel services.

4.3.3 Threads

The PIM Runtime Emulator executes object member functions in threads man-
aged by the node processes. When a node socket begins execution, it sets up two
interrupt handlers, one for communication and one for thread scheduling. When
a parcel arrives on the socket, the currently executing code is suspended and the
communication handler is invoked. If the member function being invoked is a built-
in method, the kernel executes it immediately before returning from the handler.
User-defined member functions are executed outside the communication handler.
The handler removes the parcel from the socket, creates a new thread, passes the
parcel to the thread, and places it at the end of the run queue.

The thread context switch code used by the emulator is from the Filaments par-
allel programming package [7] and the SR concurrent programming language [2];
the remainder (e.g., scheduling, thread creation) is original work. The low-level
thread creation and switching code is x86 assembly code called by the thread con-
trol system. Scheduling is preemptive; each thread executes for its allotted time
quantum or until completion, whichever is shorter. Preemption is accomplished by
an interrupt handler invoked periodically by a Linux timer. When the timer goes
off, the handler is invoked. If there is a thread on the run queue, the current thread is placed at the end of the run queue and control is switched to the thread at the head of the queue. If there is no thread available to which to switch, execution of the current thread continues.

The thread switching package used by the emulator is simple and efficient, and is able to handle thousands of threads simultaneously. Although the scheduling algorithm is very simplistic, it would be a simple matter to implement a more complex algorithm (e.g., priority scheduling).

4.4 Summary

The PIM Runtime Emulator provides a platform for prototyping distributed system services. Users create object files composed of member functions and data and submit these to the machine for execution. The virtual machine loads the object and begins accepting requests for that object. Requests take the form of object member function invocations, each of which executes in a separate thread. Member functions can perform local computation, request data from other (possibly remote) objects, and pass computation to other objects via member function invocations. This allows users to easily implement distributed services. The emulator abstracts away many of the low-level details of programming system services, allowing users to concentrate on developing algorithms in a more familiar Unix-like programming environment.

As part of the emulator, processes to emulate PIM nodes and communication networks were implemented. Communication processes manage routing between logical PIM nodes. Node processes include a preemptive thread scheduler, a runtime loader for p.out files, and several system and library calls, along with IPC mechanisms for interfacing with communication processes. A simple object file
format was designed, and a compiler (pcc) was implemented to link object code generated by the C compiler gcc.
CHAPTER 5

PIM SYSTEM SERVICES

The PIM Runtime Emulator is a platform for prototyping PIM system services. The distributed nature of computing in the PIM environment provides both challenges and opportunities for the system programmer. Normal system services must be designed to make it easy and efficient to access data or perform computation on remote nodes. Consequently, nodes must share information. This global information (e.g., the location of a particular object) must be maintained in a scalable manner, and accessing that information must be fast and efficient. Thus, existing approaches to even generic system services may not be appropriate. The name resolution and swap services described in this chapter are examples of this sort.

The PIM environment also provides opportunities for system software authors to experiment with new types of services. For example, services can be designed to “patrol” nodes in a region to check on the status of resources on those nodes in order to provide information about neighboring nodes or to tune system parameters to allow nodes to work more closely together. The load monitor service described in Section 5.3 is an example of a patrol service.

5.1 Name Resolution Service

Without a mechanism for locating objects, the ability of users to create programs that perform meaningful computation would be severely limited. For this reason,
one of the earliest services prototyped on the emulator was name resolution. This section describes the naming scheme devised for the emulator, and the methods used to match names with locations.

5.1.1 Object Naming

Objects in the emulator are identified by name, not by address. The naming service implemented by the emulator provides unique, location-independent object names, along with a name resolution service that locates objects based on their names. The naming scheme is hierarchical; each new object’s name is based on its parent’s name. A name consists of a human-readable character string of integers separated by a delimiter (currently `/`). Thus, an object could be named `/24/129/30`. This object’s parent is `/24/129`. Any children of the object will be named `/24/129/30/0`, `/24/129/30/1`, etc. Names are allocated locally by PIM nodes; there is no global naming service. An object can be created either as a child of an existing object or as a top-level object (i.e., an application instance). Uniqueness is ensured for child objects by keeping track of how many children the parent object has created. The name of the child object currently being created is simply the parent’s name, the delimiter, and the number of this child. This ensures that no object will ever have two child objects with the same name. To ensure unique names for top-level objects, a similar scheme is used, with the local system object playing the part of the parent object. Thus, a top-level object created on PIM node 4 might be named `/4/87`. This scheme allows PIM nodes to create unique names for objects without having to consult a central resource (e.g., a global name broker). Note, however, that an object’s name does not provide any information about its current location; top-level objects may have moved from the node on which they
were created, and child objects’ names are based on their parent’s names regardless of the node on which they are created.

In this naming scheme, every object knows the current location of its parent and all of its children, which eliminates the need for a centralized name service. When an object is created, the emulator ensures that the system data structures associated with both the parent and the child are updated correctly. With object migration, it will be necessary for migrating objects to inform their parents and children of their new locations in order to ensure that the chain leading from ancestors to descendants remains intact.

5.1.2 Name Resolution

Name resolution takes advantage of the location information associated with each object in the system. At a high level, resolving a name can be viewed as an object asking either its parent or child about the location of a named object. That object may need to ask its parent or child about the name, but eventually some (perhaps distant) relative will be able to map a location onto that name. This mapping is returned to the original object, and the message can be sent.

In reality, it is impractical to allow objects to manage their own parent and child information. If an object should incorrectly update its parent’s location, for example, it and all of its descendants would be unable to resolve names outside of the object subtree rooted at that object. For this reason, the system keeps track of the location of each object’s immediate relatives (i.e., parent and children) as part of the object descriptor. Resolving a name involves a series of queries between the system objects on the nodes hosting the requesting object and the query object. If the query object’s descriptor does not contain the location of the query name (i.e., the name being resolved), the system object will forward the query to the node
hosting either its parent or one of its children. If the query object is a descendant of the current object, the request is forwarded the node of to the child that is an ancestor of the query object. Otherwise the request is forwarded to the parent’s node. At some point, the request will reach a node that recognizes the query name and will be able to resolve the location.

Resolution responses are sent directly back to the source node rather than traversing the entire resolution path. When the source node receives a resolution response, it places the information in the local name cache and unblocks any messages waiting for the information.

Figure 5.1 shows an example in which an object (/0/1/0) wants to send a message to a sibling object (/0/1/1). Two data structures are depicted in the figure, the
object descriptor table and the name map, which is really the union of the parent
and child information in each object descriptor along with the contents of the name
cache. The resolution mechanism can tell that the destination object is neither a
descendant of the source object (otherwise the source’s name would be a prefix of
the destination’s name), nor an ancestor of the source object (otherwise the desti-
nation’s name would be a prefix of the source’s name). The default action is to send
a resolution request to the system object on the parent’s node (i.e., /0). When /0
receives the resolution request, it determines that the destination is a child of /0/1.
It can, therefore, look up the name in /0/1’s object descriptor and send a resolve
response containing the child objects address. When the response is received, the
resolution is placed in the name cache and the message is sent.

5.1.3 Resolution Implementation

The emulator’s name resolution mechanism employs two system object member
functions (resolution request, which services queries, and resolution response, which
services response messages). Resolution also requires the send routine to initiate the
lookup procedure when a message is being sent. Keeping a cache of name-address
pairs on each node enhances efficiency; resolution is relatively expensive, requiring
a number of sends linear in the height of the tree in the worst case. This resolution
scheme does, however, avoid congestion at one, or even a few, central name servers.
Although there is potential for congestion at the root of the tree, few resolutions
will ever need to climb that high in the name tree due to the locality of reference
exhibited by most code. Most accesses will likely be to objects in the current subtree
of the program, making long resolution paths unlikely. Eliminating the bottleneck
in the centralized version was the main determining factor in the decision to use the
distributed version.
### Table 5.1. Name Cache Performance: Total Resolve Requests

<table>
<thead>
<tr>
<th>Method</th>
<th>Cache Size (in entries)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
</tr>
<tr>
<td>Distributed Send Caching</td>
<td>12154</td>
</tr>
<tr>
<td>Distributed Resolve Caching</td>
<td>12154</td>
</tr>
<tr>
<td>Centralized Send Caching</td>
<td>10000</td>
</tr>
</tbody>
</table>

5.1.4 Name Caching

When a node resolves the location of an object, it sends the message and places the resolution in a name cache. The name cache is just a table of name-address pairs that the system can use to short-circuit the resolution process when possible. Three versions of name resolution were implemented in the emulator, two distributed services, and one centralized version. The distributed versions differ only in the extent to which they use the name cache. One version checks the name cache only on the node that is sending the message, while the other also checks the name cache on each node in the resolution tree. The performance of these schemes is measured by a test program that creates 73 objects (a tree of one node with eight children, each of which has eight children). When one of these objects receives a parcel, it increments a counter in the parcel and selects the next object to receive the parcel. The destination is chosen randomly, but is weighted toward objects close to the current object. Thus, 75% of the time the parcel is sent to an object local to its parent’s subtree, and the rest of the time it goes to a more remote object. Giving closer objects preference in destination selection simulates the locality of reference typical of real-world loads.

The test parcel is sent 10,000 times, and the emulator keeps track of the total number of resolution messages that are generated by the parcel. Table 5.1 shows the results (in number of resolution messages) for various cache sizes (zero, ten, twenty-
<table>
<thead>
<tr>
<th></th>
<th>Cache Size (in entries)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
</tr>
<tr>
<td>Distributed Send Caching</td>
<td>2433</td>
</tr>
<tr>
<td>Distributed Resolve Caching</td>
<td>2433</td>
</tr>
<tr>
<td>Centralized Send Caching</td>
<td>10000</td>
</tr>
</tbody>
</table>

Table 5.2. Name Cache Performance: Maximum Resolve Requests

five, fifty, or unlimited name-location entries).\(^1\) These results indicate that the increased performance of checking the name cache at every step of the resolution, while not dramatic, is substantial (12% with a cache size of 25). The number of resolution messages required demonstrates exponential decay with higher cache sizes, typical of cache performance in other contexts.

Centralized name resolution performs well compared with distributed versions when total resolution messages is the metric of efficiency, as shown in Table 5.1. However, a centralized name server is vulnerable to overload when many resolutions are attempted at once. The distributed resolution scheme implemented for the emulator requires more resolution messages, but no single PIM node must handle all of them. Table 5.2 compares the performance of the three schemes with regard to the maximum number of resolution requests any node has to service. The distributed versions perform very well in cases where the name cache is stressed, cutting the maximum number of requests serviced by any node at least in half. This indicates that one goal of implementing a distributed name resolution service (eliminating potential bottlenecks at a central server) has been achieved.

These results represent only some of the possible name resolution schemes and cache configurations. Further testing will allow developers to compare these ap-

\(^1\)There is a discrepancy in the table: for resolve caching the number of resolutions goes up when the cache size is increased to infinity. This is an increase of only 1%, and is due to the use of different random seeds in each run.
proaches with others to determine the best solution for PIM machines. The PIM Runtime Emulator gives developers the opportunity to develop and test new services in the absence of PIM hardware.

5.2 Object Swap

PIM nodes are in general not attached directly to any kind of secondary storage. If a PIM node runs out of memory space, it will need to swap to another PIM node. The PIM Runtime Emulator implements an object-level swap mechanism that uses the object, rather than a page or segment, as the unit of swap. When swap is required, the node packages up the target object and sends it to a neighboring (hopefully less memory-strained) node. This is not a migration mechanism; while an object is swapped out, it cannot receive parcels or perform computation. On the remote node, the swapped object is nothing but a chunk of data. To be used again, it must be swapped back in to the original node.

The swap mechanism described here is only a means with which to move an object one node to another and get it back again when it is needed. It does not define a swap policy. Such a policy would need to define procedures for making decisions about swapping. Policy issues that would need to be implemented in a fully functional system include: which node to swap to, what to do if that node has insufficient memory available, which object to swap (the largest? oldest? least active?), and when to swap an object back in (to avoid starvation). The swap mechanism implemented using the emulator does not make any of these decisions. It demonstrates a mechanism for freeing memory by swapping to another node without worrying about any policy issues.

Briefly, an object is selected for swap and sent to a neighboring PIM node. A message arrives for that object on the original node, at which a message is sent
to the swap object on the second node. The object is sent back to the original node, where it is copied back into memory. Any parcels waiting for the object are queued, and execution continues normally. The remainder of this section gives a more detailed description of the swap mechanism.

Objects are swapped out by a system object member function. This **swap** function creates a new object (the **swap object**) on the remote node. The new object consists of the data (which is the object being swapped, the **swap target**) and member functions for swapping the object back in to the original node. Once the swap target has been packaged up, the **swap** function sends the object create message to the remote node and sends the data to the newly created swap object.

When a message arrives at a node, the parcel processing routine checks to see if the destination object has been swapped. If not, a new thread is forked and placed on the run queue. If the destination has been swapped, a message is sent to the swap object informing it that the swap target needs to be swapped back in, and the parcel that initiated the swap in is placed on a queue.

The swap object is a normal object file. When the remote node receives the object load parcel, it loads the swap object file and copies the swap target into the swap object’s memory. The swap object then waits on the remote node until it receives a message from its home node. When it receives a **return** message, it immediately sends its data (i.e., the swap target) to the system object on its home node. The return parcel invokes the **swap in** member function of the system object.

The system object’s **swap in** member function places its payload in memory; the swap target is ready to receive and service parcels at this point. The final step in the **swap in** procedure is to place any parcels waiting for the swap target on the run queue. From this point on, the swap target behaves as a normal object.
The object swap mechanism implemented using the emulator provides a simple way for PIM nodes to quickly free up space when resources become strained. Swap mechanisms typically swap pages instead of higher-level units such as objects. In the case of massive PIM arrays, it is unclear that any full-blown virtual memory system will exist. For that reason, page swapping may not be possible in PIM systems. Implementing object swapping demonstrates a method of dealing with memory pressure in the absence of demand paging. For a higher degree of granularity, it would be possible to implement partial object swapping for large objects that would perform similarly in many ways to traditional page swapping. Furthermore, object swapping is not as sophisticated (or perhaps as useful) as a full-blown migration mechanism, where the objects are moved to the new node and continue to be active (i.e., accepting and servicing parcels). However, it is effective at managing memory resources in a very uncomplicated manner, demonstrating the usefulness of the PIM Runtime Emulator for prototyping PIM system services.

5.3 Load Monitor

The swap mechanism described above frees up memory by temporarily moving objects to another node. This strategy will perform poorly when nodes have no way to make smart decisions about which nodes have memory available to host swap objects. The PIM Runtime Emulator's load monitor service ensures that nodes have up-to-date information about nodes with low memory use.

The load monitor is an example of a patrol service. Patrol services move from node to node, collecting information or performing system tasks. Fast, efficient communication mechanisms make it possible to create patrol services that have little effect on the system. A load monitor parcel moves from node to node compiling and updating a list of nodes with low memory usage. Nodes can use this information to
decide where to send swap objects. Patrol services like this are likely to be a very important way for PIM systems to maintain global information, because they do not attempt to create a consistent, up-to-date description of the system’s state at every node. Such a description would be nearly impossible to maintain in a system with as many distinct nodes as PIM systems. In many cases, a close approximation is all that is needed, and patrol services are ideal at providing such approximations.

The load monitor is self-regulating: the system works to make sure that load information is updated relatively frequently, but not so frequently as to overload the system with bookkeeping tasks. PIM systems will require scalable, reliable information services in order to function efficiently. Patrol services are scalable, but may not be reliable. If a patrol parcel is at a node when the node dies, the patrol stops. The system needs a mechanism for making sure the work of the patrol is completed. Self-regulating services are a way to ensure that the number of patrol parcels in the system is within an acceptable range.

The load monitor operates as follows: each node maintains a table of information about nodes with low memory use. Load monitor parcels are passed around the system updating both the nodes’ and their own load information. If a load monitor parcel arrives at a node that has seen one too recently, it is dropped on the assumption that there are too many in the system. If too long goes by without seeing a load monitor parcel, a new one is generated on the assumption that there are not enough in the system. In this way, the service regulates the number of parcels currently active without needing to know exact numbers.

In greater detail, each node keeps track of how long it has been since it last saw a load monitor parcel. A counter is incremented every time the thread scheduler is executed. When this counter reaches a user-definable threshold (the Generation Threshold), the node assumes that there are not enough load monitor parcels in the
Figure 5.2. Load Monitor Message Frequency

system, and creates a new one. The new parcel is initialized with the information 
the node currently holds, and is sent out.

When a node receives a load monitor parcel, the load monitor system object 
member function is invoked. If the node finds that it has received a load monitor 
parcel too recently (i.e., that the counter is below the Deletion Threshold), it con-
cludes that there are too many load monitor parcels active in the system and drops 
the current one. Otherwise it checks the node list in the parcel to see if its own 
memory usage is low enough to be included on the list. If it is, the list is adjusted 
accordingly. Then the local list of potential swap targets is updated, and the parcel 
is sent on to a randomly selected node.
The emulator was tested with various values for the Deletion and Generation Thresholds to determine the effects of changing these values. Figure 5.2 plots the results of various configurations executing for 1,000 system ticks (a system tick is the time quantum allocated to a thread before the scheduler preempts it). Tests were run for three different Generation Thresholds, and for each Generation Threshold, a test was run for each Deletion Threshold less than it. Each configuration was executed eight times with random seeds generated by the Linux `time` system call, and the values in the figure represent mean values. The values along the Y axis show the average number of system ticks between load monitor parcels. The X axis is the Deletion Threshold, and each line represents one value for the Generation Threshold.

The results are unsurprising in that they show that configurations with low values for both thresholds will have fewer ticks between parcels than configurations with high values for both thresholds. One interesting effect that can be seen, however, is that configurations in which the sum of both thresholds is the same tend to have very similar results. So, for example, the 10–15, the 5–20, and the 0–25 configurations all report average system ticks between load monitor parcels around 19. Subsequent tests indicate that the average number of load monitor parcels generated by a node in each of these configurations is 25, 15, and 7, respectively, over 1,000 system ticks. System designers could use this when deciding whether they prefer to forward existing load monitor parcels more often (use the 0–25 configuration) or create new ones more often (use the 10–15 configuration). At first glance, it would seem more efficient to use the parcels already created, but there may be other factors to take into consideration.

Figure 5.2 also shows that in many cases the average time between messages is higher than the Generation Threshold. This is because the unit being measured is
average time between load monitor parcel arrivals. The Generation Threshold only
determines how long a node waits before creating a new load monitor parcel, not
how long it is before a node receives a new parcel. For example, if the Generation
Threshold is twenty, a node will create a new load monitor parcel after twenty
ticks. However, it may be a while before it receives another parcel. The Generation
threshold is, therefore, not a reliable estimate of the average time between messages.
A more accurate way to predict the average time between messages is suggested by
Figure 5.2. If the desired time between messages is \( x \), set the Generation Threshold
to \( x \) and the Deletion Threshold to \( x/2 \). This method roughly predicts the average
time between messages.

The load monitor mechanism is an example of a PIM system service that takes
advantage of the strengths of the PIM environment to solve a problem that does
not often come up in a normal computing environment. Inexpensive communication
makes it economical to have many messages moving around the system keeping
track of system information. The load monitor demonstrates the use of patrols to
dynamically keep track of global system information in a way that can be configured
by the user to have minimal system impact, or extremely current information, or
some compromise of the two.

5.4 Summary

The PIM Runtime Emulator provides a platform upon which to experiment with
PIM system services, particularly distributed system services. This chapter outlined
three services that were implemented using the emulator to take advantage of the
distributed nature of the PIM environment. The name resolution mechanism allows
users to take advantage of a global name space to perform computation on multiple
nodes without needing to explicitly manage data locations. The swap mechanism
takes advantage of available resources on other nodes to shift memory burden from one PIM node to another. The load monitor is a self-regulating distributed service that gathers information about the memory loads of PIM nodes to facilitate in the selection of swap or migration targets. Each of these services solves a problem in a non-centralized, distributed fashion. The emulator proved a useful tool for experimenting with various implementation details for this set of services.
CHAPTER 6

CONCLUSIONS

The PIM Runtime Emulator provides an effective platform on which to experiment with distributed system services. It includes abstractions of PIM communication and of individual nodes and the basic PIM operating system. To demonstrate the usefulness of the emulator, three system services that take advantage of the distributed nature of the PIM environment were constructed. The successful implementation of these services, along with the lessons learned from testing different configurations of the services, demonstrates the usefulness of the emulator for building distributed system services.

6.1 PIM Runtime Emulator

The emulator executes on a cluster of Linux workstations. Users can specify the number of PIM nodes to be emulated, and can create and load objects to perform computation. Support for this emulation comes from three key components (the emulator architecture, the thread subsystem, and support for the emulator’s object file format), which account for the bulk of the effort invested in the emulator.

The emulator is composed of two main parts: node processes and communication processes. Node processes emulate the resources of individual PIM nodes, including memory and operating system support. They are responsible for object creation, resource allocation, network I/O, and thread scheduling. Communication processes
manage message routing between nodes, allowing node processes to execute sends
without worrying about how a message gets to its destination, similar to placing a
parcel on a memory bus.

Within each node process, object parcel handlers execute within threads. The
thread manager implemented for the emulator builds on existing context switching
code to provide preemptive thread scheduling. The current scheduling algorithm
is a fixed-time quantum, round-robin scheme, but a more complex algorithm could
easily be substituted.

A simple object file format was designed for the emulator to allow easy manip-
ulation of objects. The pcc compiler joins precompiled C functions into a p.out
file that can then be loaded into the emulator. The node process’s run-time loader
reads p.out files and adjusts their headers. Once an object file has been loaded, it
is ready to receive and handle parcels.

6.2 PIM System Services

While the emulator can be used to design generic PIM programs, the motivation
for building it was to provide a platform on which to design distributed system
services for PIM. System services are procedures provided by the operating system to
allow or facilitate computation in the system. Distributed services are distinguished
from other system services by their use of multiple PIM nodes, or their support
for programs that use multiple PIM nodes. Some distributed services correspond
to services found in the “middleware” of traditional clusters, whose purpose is to
allow multiple nodes to cooperate (e.g., process migration). Others, however, are
services that are necessary for computing even within a single node, but that must
be designed in a distributed fashion to allow them to scale to multiple nodes (e.g.,
name resolution).
To test the emulator’s suitability as a platform for designing distributed services, three were implemented. The name resolution service uses distributed name information to locate objects without overloading a centralized server. The swap mechanism allows nodes to send objects to other PIM nodes until they are needed. Finally, the load monitor service patrols the system gathering and updating memory load information throughout the system. The success of these services indicates that the emulator is an effective platform for prototyping system services, and also validates many of the PIMOS design decisions embodied in the emulator.

6.3 Limitations and Evaluation

One of the key points in the design of the emulator was the decision to separate the abstractions of communication and nodes. Implementing the two modularly reinforces the distinction between the the hardware components of the PIM (i.e., the communication bus or network) and the operating system. Communication processes emulate communication hardware, while node processes emulate the operating system. This design is elegant and useful for highlighting the boundaries between components, but in practice it proved problematic. Routing every message sent on a workstation through a single point created a bottleneck that can cause errors under high message volume. The IP socket receive queue for the communication process can fill up, making it impossible for node processes to send new messages. Attempts to increase the queue size using the `setsockopt()` system call failed; it appears that there is either a maximum queue size or the Linux system call does not function correctly. Eliminating the communication process and forcing node processes to manage network routing themselves is probably the best solution to this problem, and would probably be necessary if the emulator were to be used for a very large, communication-intensive task. For tasks like the ones de-
scribed above, and for computation-intensive tasks, the emulator functions well in its current configuration.

The name resolution service implemented for the emulator works well in ideal conditions emulated. In a PIM system containing thousands of nodes, however, the potential for node failures makes robustness an issue for system services. A physical node failure has the potential to sever subtrees from the remainder of the program, making it impossible for those objects to resolve names. A robust version of name resolution will require a backup resolution method to ensure that, even in the absence of its parent, an object has some way of finding other objects. Allowing orphaned objects to be adopted is a potential solution; objects could be required to keep location information for a handful of potential adoptive parents that they could contact in the event that they are unable to contact their parents. This scheme, or something like it, will need to be added to the name resolution service described above to make it robust enough to handle node failures.

The swap mechanism prototype uses the object as the unit of swap. An alternative swap mechanism could swap pages to other nodes instead of whole objects. This way, when a parcel arrives for an object, the parcel handler can execute and an expensive swap in operation can be avoided until the missing page is actually referenced. Additionally, swapping pages will allow nodes a higher degree of flexibility than swapping objects, which will, in many cases, be larger than the size of a page. The choice between swapping objects and pages presents tradeoffs (e.g., simplicity vs. flexibility) similar in many ways to the tradeoffs between file and block caching in AFS [11] and NFS [23].

Overall, the limitations just described do not prevent the emulator from serving its intended purpose. System services do not tend to bombard the network with messages on the scale required to cause communication to fail, and the name reso-
lution service is robust enough to handle the emulation environment. The emulator provides all the facilities necessary to build the services described in this document, and the environment is relatively straightforward and easy to understand. In short, the emulator is an effective platform for prototyping distributed system services for PIM systems.

6.4 Future Work

The PIM Runtime Emulator was created to allow users to experiment with system services, so much of the future work will be to implement more substantial services for the PIM environment. Prior to doing beginning that work, however, it will be important to move the emulator from the two-level design of node and communication processes to a design that sees node processes communicating directly. This will eliminate the bottleneck problems described above, and will allow users to concentrate on interesting services without worrying about overloading the system.

The emulator could also be extended to provide better monitoring, logging, and debugging facilities. In addition, a more realistic implementation would include features such as code management (e.g., code sharing between duplicate objects concurrently loaded) and porting the system object to the p.out format rather than compiling it as part of the node process. The pcc compiler could be extended to compile objects directly instead of calling an outside compiler, and many features of the PIMOS could be tested to find optimal configurations.

Object migration is a very interesting service that would require the system to package the object and all currently executing threads and move them to another node. Implementing the p.out object format as well as the thread subsystem were important steps toward achieving object migration. Having a well-defined and well-
understood overall picture of an active object will make encapsulating it relatively straightforward.

The swap mechanism presented in this thesis does not implement any swap policy. It would be interesting to put a policy in place and observe the system under heavy loads (computational or storage). Furthermore, implementing a page swapping mechanism would allow a comparison of system performance using that and the object swap mechanism described above.
BIBLIOGRAPHY


