The Design and Implementation of a High-Performance Active Network Node

by

Erik L. Nygren

Submitted to the Department of
Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Engineering in Computer Science and Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

February 1998

Author .....
Department of Electrical Engineering and Computer Science
February 4, 1998

Certified by .....
M. Frans Kaashoek
Associate Professor
Thesis Supervisor

Accepted by .....
Arthur C. Smith
Chairman, Department Committee on Graduate Theses

The Design and Implementation of a High-Performance Active Network Node

by

Erik L. Nygren

Submitted to the Department of Electrical Engineering and Computer Science
on February 4, 1998, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Computer Science and Engineering

Abstract

A capsule-oriented *active network* transports *capsules* containing code to be executed on the network nodes that they pass through. This approach makes networks more extensible by allowing new networking protocols to be deployed and used without any changes to the underlying network infrastructure. This thesis project describes the design, implementation, and evaluation of a high-performance practical active network node that can serve as a testbed for research into active network performance and resource management issues. Nodes provide resources to executing capsules containing Intel ix86 object code. Although the current implementation does not yet provide safety or interoperability, the results of experiments performed on the system implemented for this thesis indicate that an active network architecture may be able to provide significantly flexibility while only incurring a small performance overhead relative to traditional networks.

Thesis Supervisor: M. Frans Kaashoek
Title: Associate Professor

Acknowledgments

I would like to thank the members of the MIT Student Information Processing Board and the members of the PDOS and SDS groups of the MIT Lab for Computer Science for providing invaluable suggestions and comments since the start of this project. I'm particularly indebted to my thesis advisor, M. Frans Kaashoek, David Wetherall, John Guttag, Li-wei Lehman, Greg Ganger, Dawson Engler, Butler Lampson, John Jannotti, Costa Sapuntzakis, Katy King, my parents John and Karen Nygren, and many others for their helpful suggestions, words of encouragement, and occasional vehement disagreement. This thesis would also have never been completed on time if it wasn't for the contributions of the free software community — the developers of such wonderful tools as Linux, GNU Emacs, gcc, Perl, gdb, and tgif. Metro Link, Incorporated's contribution of a module loader to XFree86 4.0 also proved critical to the success of this project.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Active networks make the network infrastructure much more dynamic by allowing code to be executing within the network. However, a number of challenges lie in the way of creating a practical active network architecture. This thesis work focuses on the design, implementation, and evaluation of PAN, a high-performance active networking system that provides a foundation for building a practical active network while demonstrating that the high degree of flexibility provided by active networks can be obtained with very little performance overhead.

1.1 Background

The current network infrastructure is essentially static. Although active code may be sent from servers to clients (such as web applets) and from clients to servers (such as OO database queries), internal network nodes (such as routers) passively switch packets. This infrastructure is standardized using monolithic protocols such as IP[27]. Adding functionality to these core network protocols is performed by adding complexity to the protocols through a lengthy process of prototyping, standardizing, developing, and deploying. The result is that although the core protocols become bloated, they are still incapable of incorporating all of the functionality within the network (such as convergecasts or data caching) that applications may desire. Until now, the only solution other than adapting protocols has been to place specialized

servers within the network to perform tasks such as multicast tunneling, web caching, and network monitoring.

By allowing computation to happen *within* the network as data passes through nodes, *active networks*[30] provide a different solution to these problems. Rather than standardizing on a protocol that describes how nodes should forward packets, an active network standardizes on an execution environment that is provided to the *capsules* that pass through network nodes. A capsule contains both data and a reference to code to execute at each node the capsule passes through. In a traditional network, routers look at packet headers and decide where to forward the packets. In an active network, routers execute the code referred to by capsules, and this code tells the router where to forward the capsule.

This active networking approach would allow network protocols to evolve much more rapidly. In an active network, protocols can be written and immediately deployed without any need for an extensive standardization process. Because new protocols can be written (or existing protocols can be modified) to provide exactly the functionality that is needed by applications, the large bloat associated with monolithic protocols can be avoided. Active networks take the end-to-end argument[24] to the extreme by allowing applications and protocols to do exactly what they need to do exactly where they need to do it. Utilizing active networks may make it much easier to implement and deploy new protocols for tasks such as multicasting, convergecasting, data caching, network monitoring, and dynamic data distribution.

1.2 PAN — a step towards addressing the challenges facing active networks

Unfortunately, active networks may not come without costs. In order to become practical, they must be able to overcome the hurdles of performance, safety and security, and interoperability. Active networks must be able to achieve performance comparable to existing networks. They must be able to provide safety, security, and

robustness equaling or exceeding that of existing networks — a difficult challenge for a system designed to allow code to execute and migrate within a huge distributed system encompassing many administrative domains. Finally, due to the highly heterogeneous nature of an internetwork, an active network must have the same high degree of interoperability that a traditional network has. Unless all three of these challenges can be addressed, an active networking infrastructure will remain impractical.

As my thesis project, I have designed and implemented PAN, an active network node that takes the first step towards the realization of a "Practical Active Network." One of the primary goals of PAN is to answer the performance question: in the base case of simply routing packets to their destination, can an active network node achieve performance comparable to a traditional network? After implementing and testing PAN, it appears that this hypothesis may be valid. Tests running on a 200MHz Intel PentiumPro running Linux have demonstrated that PAN is able to forward at least 100Mbps worth of data, and, when compared to the same Linux machine acting as a traditional router, incurrs a fixed overhead on the order of only 20 microseconds per capsule.

Although PAN is designed with the safety, security, and interoperability issues in mind, the current implementation does not fully address those issues directly. Instead, the current implementation focuses primarily on validating the performance hypothesis. However, it should be possible to add safety, security, and interoperability to PAN without any major redesigns of the system and without significantly affecting peformance.

Surprisingly, it doesn't take too much effort to achieve reasonable performance in an active network node. PAN achieves high performance in four ways: through processing capsules in the kernel, by minimizing data copies, through executing instructions native to the processor on the node, and through a design philosophy of providing capsules with maximum flexibility.

By processing capsules in the kernel, rather than in a user-space process, the node avoids having to copy data between kernel-space and user-space. Additionally, interference from the scheduler is greatly reduced.

PAN provides a uniform memory management system that allows handles to regions of memory to be passed around the system. In the base case of simply forwarding capsules, this eliminates the need for the node to copy or even touch the data body of capsules.

By caching native executable code that corresponds to the code that capsules refer to, nodes only incur the cost of loading and/or compilation on the first time that a new type of capsule is used. All subsequent capsules of the same type can be immediately evaluated without the need for interpretation or additional compilation.

Consistent with the overall philosophy of active networks, PAN provides capsules with a great deal of flexibility over what they can do. By allowing capsules to only do what they need to do and to do things in the way most appropriate to what they are doing, capsules can be written to work with abstractions rather than fighting against them.

1.3 Contributions of this work

This thesis makes a number of contributions. First, people designing and implementing active network systems should hopefully be able to learn from the lessons learned during the design and implementation of PAN. Second, PAN demonstrates that active networks can provide a high degree of flexibility at only a small fixed cost in the base case. As a result, this thesis demonstrates that performance may not be one of the significant challenges facing the acceptance of the active networking approach. Third, this thesis provides a functional active network implementation that can be used for developing active network applications and for experimenting with active network resource management, safety and security, and interoperability issues.

A public release of PAN will be made available sometime during 1998. This will allow other people to write applications for the system and to experiment with it. This release will be made available at:

`http://www.mit.edu/people/nygren/pan/`

1.4 Structure of this document

Chapter 2 contains a summary of other work that is related to this thesis. Chapter 3 discusses some of goals and philosophies associated with the design of PAN and provides some background into active networking issues. Chapter 4 presents the details of the design and implementation of the PAN system. Chapter 5 describes the experimental setup used to test the performance hypothesis and provides the results of these experiments. Chapter 6 discusses future directions which PAN my take. Finally, Chapter 7 presents some conclusions drawn from the results of this thesis.

Chapter 2

Related Work

The trend towards extensible systems is hardly new. Operating system architectures such as Exokernels[11] and SPIN[4] allow user applications to extend the operating system's functionality. Languages environments such as Java[14] allow web browsers and other end applications to be dynamically extended to run *applets.*

Possibly the first programmable network was a programmable packet radio network called Softnet[39] that was developed in Sweden in the early 1980s.

The current wave of active network development is fairly recent, having gotten off in 1996 with [30] and [28]. Much of the work in the area has been focused on either developing long term technologies or on developing prototype or proof-of-concept systems. A fairly complete survey of ongoing active network research was recently published[29].

Two somewhat different approaches have been taken to active networking: the *capsule-based* (or *integrated*) approach, which is what PAN uses, and the *programmable switch* (or *discrete*) approach. The integrated approach revolves around programming in the *messenger paradigm*[8] — capsules containing both code and data move around the network and are executed on the nodes they pass through. In the discrete approach, functionality can be added to nodes out-of-band from the packets being processed by the node.

The design of integrated approach systems, such as PAN, makes them much better suited to packet-oriented networks (such as IP) than to connection-oriented networks

(e.g. ATM). This is because code is executed for each packet that has associated code. The discrete approach is much better suited to connection-oriented networks than the integrated approach.

A number of projects at MIT have been making good progress towards demonstrating the usefulness of active networks. The ACTIVE IP Option [36] project embedded small Tcl programs in the option fields of IPv4 packets. The ANTS system [35] allows for the rapid prototyping of active network ideas and applications through a capsule-based active network system written in Java. A number of sample active network applications have been developed using ANTS [35] [21]. This thesis project draws heavily on many of the ideas and application ideas developed through the ANTS work.

A group at U. Penn and Bell Communications has been working on developing a programmable switch and on developing Caml and SML-based programming language technologies that may be used in future active networks [25]. They have implemented and tested an active bridge[2] written in Caml that runs in user-space under Linux. However, their performance measurements are substantially slower that the results measured with the PAN system.

The University of Arizona's Liquid Software[15] project is developing technologies for the high-performance compilation of mobile code. The project is also looking at mobile search applications.

The x-kernel [16], also developed at University of Arizona, is a network-oriented operating system that provides a consistent interface for constructing and composing network protocol stacks that can be configured into the kernel at compile-time.

The BBN Smart Packets project[17] is looking into lightweight, but still expressive, capsule-based active networks. After having evaluated existing languages, they are developing a highly compact bytecode language for their active networking system. Many aspects of their system are similar to what was developed in this thesis project. The most significant difference is their focus on network management and diagnosis applications, which have stronger security requirements and weaker performance requirements than PAN.

The Netscript[38] project at Columbia University is developing a language, programming model, and run-time environment for a discrete approach active network where agents set up packet dataflow structures and allocate resources. The Netscript system provides a "universal" abstraction of a programmable network device and a "dynamic" "dataflow" language.

A project at GeorgiaTech [5] is developing a programmable switch approach active network. They are investigating using active networks as a tool for dealing with network congestion problems.

In addition, work at CMU is investigating "Application Aware Networks" and issues of resource allocation in networks.

Chapter 3

Goals of PAN

PAN is designed to provide a testbed for active network research while demonstrating that active networks can achieve acceptably high performance. By doing so, PAN provides a good first step towards the implementation of a practical active network.

In order to be *practical*, an active network must address three primary but conflicting goals: safety, interoperability, and performance. Although techniques are available to address each one of these issues individually, there are still a large number of unknowns regarding how to address all three issues simultaneously in the context of active networks. This chapter not only explains the goals that the current implementation of PAN is designed to achieve, but also discusses practical active network requirements that have not yet been solved but which significantly influence the design of PAN.

3.1 Active networks: a moving target

Because the uses for active networks are still being determined, the degree to which performance, safety, and interoperability must be addressed is still unclear. The applications that run on top of the system will have a substanial effect on all three of these requirements. For example, the requirements on performance are substantially different for a network management system than for a system that handles all of the data traffic within a network.

The best solution to this problem is to develop an active network system for experimenting with both applications and implementation issues. This is exactly what PAN does.

3.2 Obtaining high-performance

Routers are expected to provide high performance, both in latency and throughput. As a result, a practical active networking system must have a low performance overhead. However, the performance requirements of an active network are also determined by the applications that run on top of the active network. Even if active nodes never replace backbone routers, a future network architecture may contain active nodes distributed throughout the network, allowing protocols to use them as data caches, multicast routers, and network monitoring stations. The level of performance that active networks are capable of achieving will have a significant effect on the applications for which they are employed.

PAN is designed to address the performance issue by demonstrating that active networks can, in fact, achieve high performance. Ideally, an active network should achieve performance comparable to a traditional network. For example, an active node processing capsules containing the code:

```
if (capsule_at_destination) then deliver_to_app();
else route_to_next_node();
```

should obtain performance (in bandwidth and latency) comparable to a traditional hard-coded network node that simply routes packets to their destinations or delivers them locally. Although the active node incurs a small fixed cost per capsule that the traditional node does not incur, this cost should be small relative to the overall cost of processing the capsule. With the active networking solution, additional functionality (such as keeping track of the nodes the packet passes through or caching the packet at each traversed node for faster error recovery) should add only a small incremental cost over the small fixed cost of the base case. Although PAN does not yet fully provide

safety or interoperability, experiments performed on PAN (described in Chapter 5) appear to validate this performance hypothesis.

In addition to providing high performance in the base case, PAN is also designed to provide good performance for protocols with additional functionality. As a result, there should not be a significant performance penalty if a capsule modifies its contents or caches data in a node. PAN is able to achieve this through its flexible memory management architecture.

By providing a high-performance active network implementation, PAN should also prove useful to researchers developing applications and protocols for active networks. By being able to test out active networking protocols in a system comparable in performance to a more traditional network, it is possible to directly compare new active protocols to their traditional counterparts.

PAN runs on top of existing workstation operating systems such as UNIX and is designed to achieve performance as similar as possible to a traditional networking implementation on such a machine. In reality, many intermediate network nodes use specialized hardware that provides superior performance to intermediate nodes running on workstation operating systems. As a result, PAN may have significantly lower performance than traditional specialized hardware solutions (this has not yet been experimentally verified). However, it will provide much more functionality. Once active networks are demonstrated to be worthwhile, specialized network hardware may eventually contain hardware to execute active networking capsules. In order to both provide high performance, and to integratable into specialized hardware at some point, the PAN system is designed to be as light weight as possible without sacrificing functionality. This is one reason why a large mobile code system, such as Java, is not used in the current implementation.

3.3 Providing a testbed for experimentation

PAN is designed to be both flexible and powerful, allowing it to be used for prototyping active network technologies. Many of the active networking ideas used in this

project come from the ANTS project[35], a prototype active network that is being used at MIT for experimenting with active networking ideas and applications. In order to allow applications designed on ANTS to be easily implemented on PAN, PAN uses capsule programming paradigms and node interfaces similar to those provided by ANTS. In a number of cases, however, PAN provides interfaces that are more powerful or more flexible than those provided in ANTS. In these cases, an attempt is made to provide a mechanism whereby the interface provided in ANTS can be easily implemented using the interfaces provided in PAN. See Section 4.11 for a description of the significant differences between the interfaces offered by PAN and ANTS.

In addition to acting as a testbed for applications, PAN can also be used as a testbed for experimenting with various resource management and active networking issues. Experimenting with these issues is beyond the scope of this thesis, and experimenting with some mechanisms may require substantial changes to the existing design. Resource management issues that might be explored using the PAN system include memory and CPU management schemes (how much memory should capsules be provided and how should their run-time be bounded), network utilization issues (how much bandwidth should capsules be able to use), soft state cache management and keying schemes, and protected node resource access policies (e.g., who should have access to modify routing tables). All of these should be looked at both for individual capsules and across flows of capsules.

Other design and implementation issues that might be explored using PAN include how to do memory management (reference counting software segments or a full-blown garbage collector), how to efficiently provide memory protection (sandboxing or software segments or some other scheme), how to bound code run-time (timers or calculating execution cost prior to run-time), how to allow packets to suspend their execution, whether to interpret or compile bytecode or simply use native executables, the amount of load-time optimization to perform on bytecode, whether to include code names or executable code in capsules, and how to provide safety and security within a node (code verification and/or code signing). Because it is easier to automatically insert checks and code into a simple bytecode than into native object code, it may be

easier to address some of these issues with a later version of PAN that uses a bytecode instruction set.

3.4 Design for future interoperability

Any real network consists of a large number of heterogeneous devices running different operating systems on different types of hardware. As a result, any practical active networking system must address interoperability and code mobility issues. There are two aspects to interoperability: providing consistent node interfaces and providing for code mobility.

The current implementation of PAN provides consistent interfaces and a consistent view of a network node to code executing on the node. As a result, nodes appear identical to capsules regardless of the operating system they are running on top of and even regardless of the node address family that they are using. The current interfaces even provide mechanisms for accessing capsule data independent of the byte ordering of the node's host hardware.

Any executable code designed for use within active network must somehow be *mobile*, meaning that it does not depend on any specific processor or operating system. Most existing mobile code systems (such as Java[14], Perl, ML and LISP variants, Safe-Tcl, and PCC) are either too general, too specific, or too heavy-weight to be the ideal solution for a practical active networking system[17].

PAN does not yet fully address the code mobility problem. The current implementation distributes code objects that contain Intel ix86 object code in either an ELF or a.out format. Although this allows code objects to be used on almost any node running on a machine with an Intel ix86 processor, a bytecode will need to be developed in order to provide full interoperability. This project aims to provide insight into what is a good level for an active networking bytecode so that maximum flexibility is provided without sacrificing performance.

The design of the current implementation of PAN takes a great deal of care to ensure that it will be easy to replace the current binary object code loader with a

just-in-time bytecode compiler or a bytecode interpreter. In order to not generate performance results that wouldn't carry over to a bytecode-based system, the current implementation performs many of the same checks (such as run-time bounds checking on memory accesses) that compiled bytecode would also need to perform. As a result, a bytecode-based system should be able to achieve performance comparable to the existing system.

3.5 Design for future safety and security

PAN does not yet provide any safety or security guarantees. However, safety and security are not afterthoughts to the PAN design — the current design keeps them in mind and provides mechanisms whereby they can readily added to future versions. Safety protects a node and a network from programming or design errors, isolates related faults and prevents related denials of service, and prevents the system from taking undesirable actions, such as inadvertently releasing private data. Security is a subset of safety and aims at preventing intentional or malicious attacks against the system. Ideally, a system that provides safety also provides security. Because high reliability is important to network nodes, providing safety and fault-tolerance is just as important as providing safety.

An active network must provide security and safety at both individual nodes and across the entire network. These are two fairly different problems that will require different approaches to solve.

3.5.1 Node safety and security

A practical active network must prohibit itself from being used as a channel for gaining unauthorized access to system resources (such as access to the disk or to unauthorized portions of system memory). In addition, a node must protect against denial of service by an attacker or by malfunctioning code. For example, code runtime and memory utilization must be limited.

A degree of security (but not safety) can be provided by cryptographically signing

code using a public key system. This can allow a node to only accept code signed by a trusted party. Unfortunately, this doesn't prevent against programming errors and thus doesn't provide safety. In addition, the concept of a "trusted party" doesn't scale to a global internet. If different administrative domains only allow code from different trusted parties, interoperability will suffer. However, cryptographic signatures may be useful within active networks. For example, they may be used to securely bind protocols together or may authorize capsules to access protected node resources such as routing tables. In addition, code signing may be used in testbed networks to prevent an attacker from breaking into a testbed by injecting malicious capsules into the network.

By using a type-safe bytecode rather than native object code, it should be much easier to provide safety. Load-time checks can be performed on the bytecode, insuring that it accesses node resources safely. The security and resource management model designed into the current implementation of PAN assumes that a type-safe bytecode will eventually be used. As a result, mechanisms such as name-space control and unforgeable handles to objects are used to maintain control over resources. Note that although these mechanisms provide guidelines to existing ix86 object code regarding what it is allowed to do, they do do not prohibit the code from violating the constraints. In the remainder of this thesis document, references are occasionally made to unforgeable handles and name space control. Realize that these are references to design decisions that will allow security to be added when bytecodes are in place and are not mechanisms that currently provide security.

3.5.2 Trust no one: network-wide safety and security

As PAN is designed to run over a large internetwork that spans multiple administrative domains, it is crucial that authors of code objects and users of the system remember that an active network is not only subject to the same security problems as a more traditional network, but that new security concerns may also become evident. Any active network designed for deployment in an internetwork should at least have a security model where all nodes and network links are untrusted. Users and

programmers should assume that an attacker would be able to insert a capsule with arbitrary contents at any point in the network. Attackers may also be able to view or modify capsules at any point in the network.

The current implementation of PAN fails to be secure under these conditions. However, any design decisions that have been made about security take this stringent security model into account. Much more work will need to be done to allow PAN to be safely usable in such an insecure network however.

Two clear challenges to providing global security and safety within an active network are preventing denial of service attacks and providing end-to-end encryption and accountability.

The most obvious potential denial of service attacks are those that could be waged by worm-like capsules that would rapidly spread to all nodes of a network and would maintain a network-wide broadcast storm, thereby consuming all network capacity. One of the challenges in preventing this sort of attack is being able to distinguish desirable behaviors (such as multicasting the video stream from a Mars probe to a million subscribers) from undesirable behaviors (such as a worm that replicates itself a million times). This can be especially hard to do when trying to determine whether to allow an activity from the viewpoint of a single node without any global knowledge. Some approaches, such as time-to-live (TTL) fields on capsules, may solve the problem in the short term but do not scale to an active global internetwork.

Another network-wide security problem is providing encryption, authorization, accountability, and security in a network where nodes are not necessarily trusted. End-to-end encryption of entire capsules will not be possible if capsule contents need to be accessed by nodes along the capsule's route. In addition, it may be hard to distinguish an action being taken by a capsule on behalf of a trusted application from an action taken by a compromised node that is "pretending" to process a capsule. In the extreme, the only solution to this problem may be to deploy a complicated distributed system security scheme similar to the one used in Taos[37]. Because this may place a heavy performance burden in places where it is not needed, it may be more appropriate to apply the end-to-end argument by providing encryption and

authentication tools (and possibly even a key distribution architecture) to capsules, but without dictating how the capsules use the tools.

Chapter 4

Design and Implementation

This chapter describes the design of the PAN system and provides information about the current PAN implementation. Section 4.1 provides an overview of the architecture of the PAN, Section 4.2 presents an overview of the design of PAN nodes, Section 4.3 briefly describes the two existing PAN node implementations, Section 4.4 explains how code objects and applications interface to nodes, Section 4.5 provides more information about code objects and protocols, Section 4.6 contains information about the format of capsules, Section 4.7 explains how memory is managed within a PAN node, and sections 4.8 and 4.9 give information about the details of application and network links, respectively. Finally, Section 4.10 describes some miscellaneous utility functions that PAN nodes provide to code objects, and Section 4.11 explains the motivation behind the some of the interface differences between PAN and ANTS.

4.1 Architecture overview

The PAN system uses an active network architecture similar to that used in the ANTS [35] system. A PAN *network* consists of a number of *nodes* interconnected with each other across unreliable *network links*, such that any node can potentially communicate with any other node (except in the case of network failures) by sending a *capsule* across one or more network links. As these capsules pass across the network, they may be processed by any or all of the nodes that they pass through. Many of

the leaf nodes of the network are connected to *applications* by *application links*. The general goal of the network is to allow applications to either communicate with each other or to allow them to obtain information about the state of the network itself.

The primary means of communication within the network is a *capsule*. Abstractly, a capsule contains both the data it is transporting as well as a reference to a *code object*. A code object is primarily a block of instructions that is evaluated at each node that the capsule passes through. The instructions specify what the node should do with the capsule. These instructions could ask the node to pass the capsule towards a destination node, modify the contents of the capsule, pass the capsule to an application, access state within the node, or just about anything else, within certain constraints. If a capsule references a code object which is not yet available in a node, *demand loading* is used to retrieve the code object and its dependencies from some other node. A code object is similar to a "class" in the object oriented programming paradigm, with capsules being instantiations of the class. At each node the capsule passes through, the *accept* method of the capsule's code object is evaluated. Some code objects may even *depend* on other code objects by call methods in them. A group of code objects which are designed to work together is collectively referred to as a *protocol*.

Nodes may also contain some state which may be accessed by the capsules which they are processing. In non-leaf nodes, this state is always *soft state*, meaning that capsules must not rely on the persistence of data stored within the network. Nodes may be periodically restarted, the network topology may be dynamically rearranged, and the fact that storage space within nodes is finite means that nodes must continuously flush old data as capsules add new data to the node. Because of this, it makes sense to think of the mechanism for storing state within nodes as a *soft state cache* rather than as a persistent store.

In an example use of PAN, an application might send a capsule across an application link and into a node as the first step of sending data from that application to another application elsewhere in the network. Once in the node, the *accept* method of the code object associated with the capsule is executed by the node. This method

might compare the current node address with the destination address stored within the capsule's data body. On determining that the capsule is not yet at its destination, the capsule requests that the node send it across a network link towards its destination. This process continues at the nodes along the way until the capsule reaches its destination. At that point, the evaluating capsule checks in the node's state whether a local application is associated with the capsule's code object. If there is such an application, the capsule asks the node to send the capsule across the application link to the application, thus completing the capsule's journey and delivering the data.

4.2 Overview of a PAN node

The primary purpose of a PAN node is to process the capsules that arrive and to manage the resources that are needed for this task. The existing implementation of PAN is designed to be portable, efficient, and eventually safe and secure. Nodes process capsules by evaluating them to completion in an *environment.* Through the environment, capsules' code objects are provided with access to node resources through a fairly simple interface layer called the PAN Node Interface (or PNI). See Figure 4-1 for a diagram of the node architecture.

By providing simple and consistent interfaces, code objects are able to run on PAN nodes on a wide range of platforms, thus providing portability and code mobility. The only current restriction is that the platforms must have the ability to run Intel x86 object code. As discussed later, future versions may not even have this restriction.

Memory within a PAN node is managed through a *software segment* interface. This memory management system provides a consistent interface to code objects, minimizes data copies, allows data to be shared between code objects, and provides buffer-management functionality similar to that found in many modern operating systems.

When a capsule arrives at a node, an environment is created in which to evaluate it. The environment contains information about the source of the capsule, the capsule's status (ready to run or waiting on a code object), and about resources that the

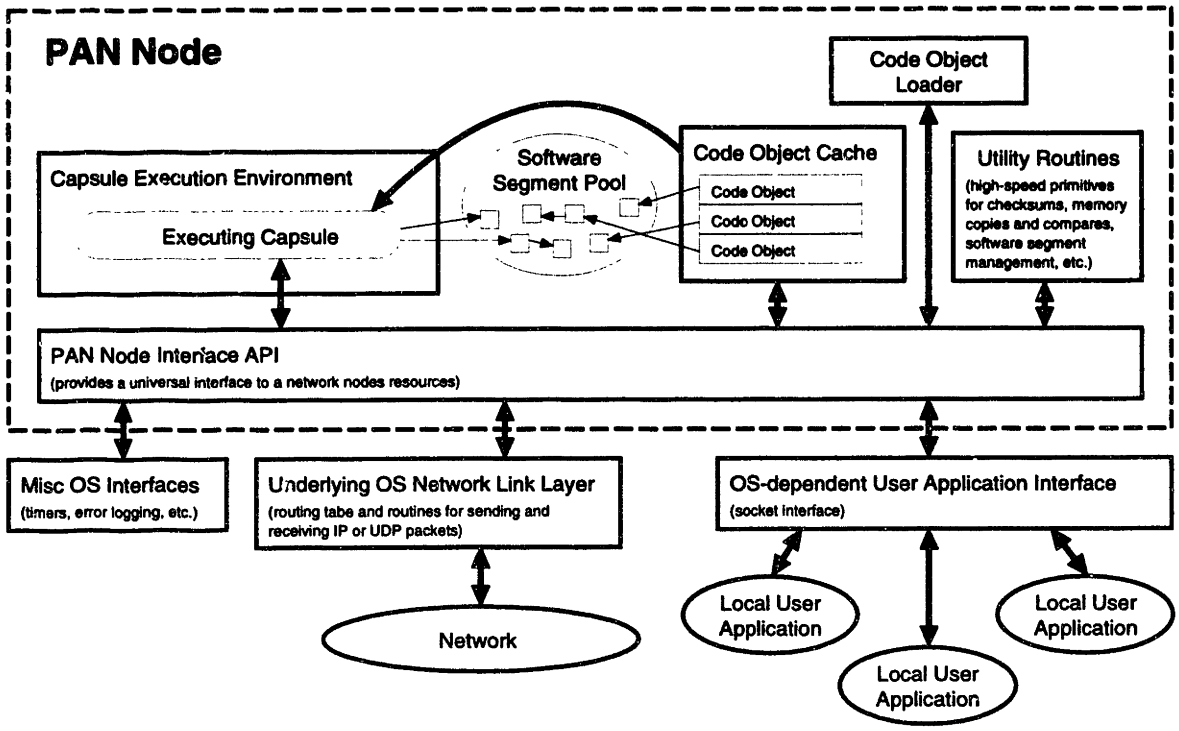

Figure 4-1: The architecture of a PAN node. An API provides the code objects of executing capsules with a consistent universal interface to the node's resources.

capsule is using. The latter is maintained so that the resources can be reclaimed when the capsule finishes its execution. In the current implementation, the node may only evaluate one capsule at a time.[1] Similar to the processing of traditional network packets, capsules which arrive when a capsule is already being evaluated have their execution environments queued up and are processed when the node is done with the current capsule.

4.3 Node implementation overview

Two PAN node implementations currently exist and share most of their code. One version runs as a user-space process on a UNIX-like operating system. Communication with applications is performed via UNIX domain sockets while messages are sent to other nodes using UDP/IP packets. The other version is implemented as a loadable kernel module for the Linux 2.0 operating system. The kernel implementation communicates with applications using a special socket type and communicates with other nodes using a protocol layered on top of IP. It should be relatively straightforward to add additional implementations (for example within a BSD-derivative such as FreeBSD or even within WindowsNT or on top of an Exokernel). The Linux kernel implementation used only existing kernel interfaces and required no changes to or recompilation of a stock Linux 2.0.32 kernel (however, a single line of code will need to be added in order to allow nodes acting as routers to intercept and process capsules that are not destined to them).

Code within the PAN node implementation is either platform-independent code that is used on all platforms, or is platform-dependent code that is only used in one implementation. As a result, a single source tree is used for all platforms, making the system much more maintainable.

[1] This is to prevent two capsules from simultaneously trying to access some shared resource in an unsafe manner — PAN does not yet contain a mechanism to force executing code to access resources in a synchronized manner or to force code to obey volatile resource access permissions that may change at any time. As long as capsule execution time is bounded and nodes only have a single processor, evaluating only one capsule at a time does not present many problems.

The platform-independent code has routines for managing and accessing memory and node resources through a unified *software segments* mechanism, for loading and processing code objects and making them ready to run, for maintaining a cache of code objects, for creating capsule environments, and for evaluating capsules within their environments. The platform-dependent code contains the routines for handling node addresses, for communicating with applications, for communicating across the network, and for dispatching capsules as they are received.

4.4 Code object and application interfaces to nodes

Code objects, and thus capsules, are presented with a uniform interface to a node called the *PAN Node Interface* or *PNI*. This interface presents a set of data types and functions that code objects can use for accessing memory and other node resources. The interface also contains routines for inserting code objects into nodes, for logging status information, and for sending capsules across the network or to applications. Other utility functions also provided to perform functions such as accessing the local node time or for performing checksum or cryptographic operations. The interface also provides a few container data types (such as a hash table) that may be useful to code objects for storing information in a node's soft state.

A library called the *PAN Application Library* or *PAL* is presented to applications which would like to be able to send and receive capsules. The library contains routines for connecting to nodes, inserting capsules into them, and for receiving data from them. The library also contains the same uniform memory creation and access routines (software segments and software segment streams) that are present within node code and within the PNI, so as to simplify writing applications. Note that many of the functions listed in the tables of code object interfaces presented in this chapter are also available from within applications and are also available and used extensively within the existing node implementation.

4.5 Code objects

Within the PAN system, sequences of executable instructions and associated data are grouped together into *code objects*. A code object is roughly equivalent to an object (.o) file or an object-oriented class, containing both static data and *methods* (also called "functions"). All code objects also have a unique *code object name* by which they are referred to.

Each PAN node also maintains a *code cache*, keyed by code object name, that contains all of the code objects that have been loaded into the node. Each code object also contains information about its status (e.g., is it linked and ready to process capsules). The code cache keeps track of dependencies between code objects and of how often code objects are used. As the code cache fills up, it may periodically evict infrequently or unrecently used code objects from the cache.

Each code object maintains its own *soft state* — data that capsules can leave at nodes for other capsules to later access. Soft state can be used for keeping track of protocol state variables, storing capsule data for retransmission, maintaining multicast routing tables, maintaining mappings from port numbers to application links, or just about anything else that a code object may need to maintain state between capsules for. As discussed later in subsection 4.7.5, the node monitors how much memory each code object is using and periodically instructs code objects on how much data they must free, but decisions on which data to release are left up to code objects.

All capsules begin with the name of the code object that should be used for processing them. On arrival, the code object is looked up in the code cache and either a request is sent to load the code object or the code object's *accept* method is applied to the capsule.

In addition to an *accept* method that is evaluated on capsule reception, code objects may also have *init* and *finalize* methods that are called when the code object is loaded and unloaded (see Table 4.1). This allows code objects to set up and clean up their soft state as appropriate.

4.5.1 Code object naming

Code objects are named with a cryptographic hash (such as SHA-1[12] or MD5[23]) of the code object's code. This allows code objects to be uniquely identified. This scheme eliminates the need for a centralized code naming system, guarantees that capsules are executed using the code objects they asked to be executed with (assuming that the capsule wasn't modified on the wire and assuming that the node has not been compromised), eliminates code versioning problems, and guarantees that code objects that reference other code objects are actually referencing the code objects (of the correct version) that they are trying to reference. In addition, capsules enter nodes with a pre-computed hash that can be used to rapidly look up their code objects in a hash table. The only times that hashes need to be computed is when new code objects are received by a node and when code objects are initially being compiled. However, this scheme does impose the restriction that the code object dependency graph can't have any cycles (i.e., two code objects can not both name each other). Note that all of these "guarantees" are only probabilistic and are dependent on the strength of the cryptographic hash. Although SHA-1 is currently used, code object names start with a description of the type of hash function used to generate them. This would allow other hash functions, such as MD5, to be used as well.

4.5.2 Code object dependencies and symbol resolution

Code objects may also *depend* on other code objects, allowing a code object to call functions or access variables in another code object on which it depends. This encourages modular programming and code reuse.

When code objects are loaded into a node, the node first waits for all dependencies to also be loaded. Once this happens, the code object is linked into the node and all unresolved symbols (i.e., references to functions and variables) are resolved. Each code object has a symbol table containing all of the symbols that it exports. An unresolved symbol is first looked for in all of a code object's dependencies. The first matching symbol found there is used. If the symbol isn't in any of the dependencies,

the code object loader looks up the symbol in the node's symbol table which contains entries for all of the functions exported by the node to capsules through the PAN Node Interface. If any of the symbols in a code object are unresolvable, the code object fails to load.

4.5.3 Code object distribution and loading

Code distribution is performed using a scheme discussed in [35]. When a capsule is received by a node that doesn't recognize its code object name, the node sends a capsule requesting the named code object back to the code object home specified in the capsule header. Along the way, nodes may have cached the code object, and if so they reply with the cached code object. If none of them have cached the code object, the node that originated the capsule replies. If the needed code object doesn't arrive within a reasonable amount of time, the capsule is dropped. Once a code object and its dependencies are loaded and linked, any capsules waiting on it are executed.

In order to provide maximum flexibility with the minimum number of different mechanisms, code object demand loading is done within the PAN system by a *dynamicload* code object that is available at all nodes. The PNI provides functions for both loading a code object from a software segment into a node and for retrieving a software segment containing a code object. These mechanisms are also used by the *insertco* code object to enable applications to load new code objects into nodes. Because there is nothing magic or special about the *dynamicload* and *insertco* code objects, it is possible to write code objects that do things like preparing the way for a capsule by traversing a network path and preloading a code object into all of the nodes along the way.

If a code object fails to load and link due to a problem such as unresolved symbols or an invalid format, the code object header remains in the code cache but has its status set to indicate that it failed to load. Any future attempts to load a code object with the same name into the node will also be rejected, as they would also fail to load. In addition, any code objects which claim to depend on a failed code object also fail to load, and any capsules using the code object are rejected. By caching failures, the

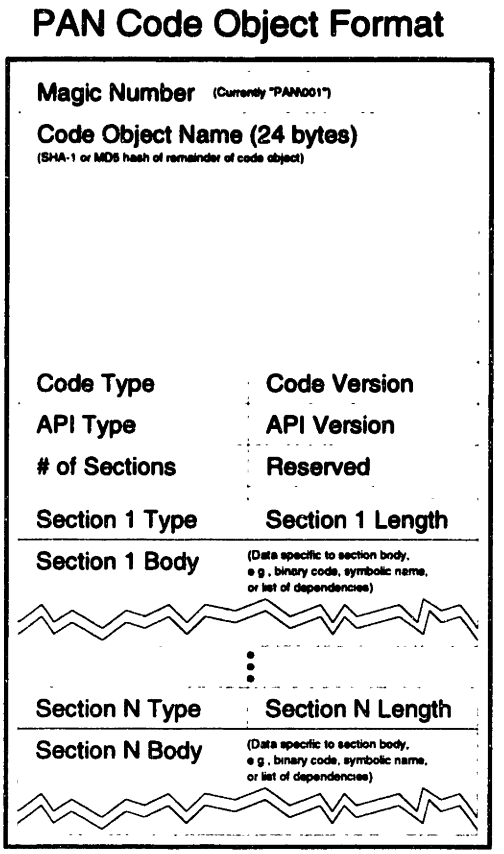

Figure 4-2: Format used to transmit code objects.

system prevents a denial of service attack from being launched by sending invalid code objects. This is important because the linking of code objects can potentially take a relatively long time relative to other operations that the node typically performs.

4.5.4 Current implementation of code objects

Code objects are transferred across the network as a header followed by a number of sections (see Figure 4-2). The header contains a magic number, the name of the code object (i.e., a cryptographic hash of everything following the name in the code object), type and version information for both the code object and the node API used by the code object, and the number of sections that follow the header.

Each section contains fields for the type and length of the section. This is followed

Code Object (PNI) Interface to Code Objects	
`pniCodeobj_load`	loads the code object starting at the current position of a pniStream into a node (used by *insertco* and *dynamicload*)
`pniCodeobj_lookup`	looks up a code object by name in a node's code cache and returns a software segment containing the code object (used by *dynamicload*)
`pniCodeobjName_read`	reads a code object name from a pniStream
`pniCodeobjName_write`	writes a code object name to a pniStream

Interfaces Code Objects May Export To Node	
`accept`	passed a capsule for the code object to process
`init`	called by the node when the code object is done loading into the node
`finalize`	called before a code object is expunged from a node's code cache

Application Interfaces To Code Objects	
`panAppCodeobj_load`	loads a code object from a file; searches the `PAN_CODEOBJ_PATH`
`panAppCodeobj_insertIntoNode`	uses *insertco* to insert a code object across an application link and into a node
`panAppCodeobj_getCodeobjName`	returns the code object name of a code object

Table 4.1: Code Object Interfaces

by the section body. Current sections include a section containing a list of the code object's dependencies, a section containing the symbolic name of the code object for debugging purposes, and a section containing the executable instructions.

In the current implementation of PAN, the executable instructions contained in code objects are Intel ix86 object files (.o files) in either an ELF or a.out format. The node links these together and resolves symbols using a binary object code loader derived from the loadable module loader written by Metro Link that will be included in XFree86 4.0. Using this system, code objects compiled on almost any Intel ix86 platform can be loaded into a node running on any type of Intel ix86 platform. In addition, because the PAN Node Interface remains consistent across platforms and between different types of nodes (such as user-space and kernel nodes), code objects can be used anywhere without recompilation.

Because no checks are performed on the object files, it's possible that a malformed code object could corrupt node state, cause the system to crash, or access "protected" state. In order to provide safety and security, a future version of the system will use cryptographically signed code objects and a type-safe bytecode language. By using a bytecode language, the system will also achieve code mobility.

4.5.5 Code objects as data abstractions and guardians

Because code objects can control which symbols they export (by declaring variables and functions as `extern` rather than `static`), it is possible for a code object to export a narrow interface to code objects which depend on it while keeping some state information private. Since soft state is maintained within code objects rather than in a global and unified soft state cache, it is possible for a code object to act as a way for multiple code objects to safely share data. Code objects which perform this task, called *guardians*, can maintain shared state, such as routing tables, while requiring that all code objects use a set of abstractions to access the state. These abstractions can perform access checks as appropriate. In the current system where code object instructions can do anything they want, this mechanism is not terribly effective. However, this mechanism should be effective in a future version of the

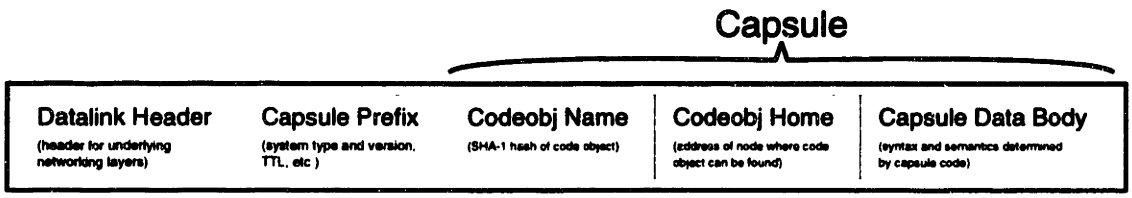

Figure 4-3: A PAN capsule contains the name and home address of a code object followed by some arbitrary data. During transmission, capsules are prefixed with a datalink header and a capsule prefix.

system which uses a type-safe bytecode language.

4.6 Capsules

Capsules are the primary means of communication within PAN. At an abstract level, they contain a reference to a code object in addition to some arbitrary data. On arriving at a node, an *environment* is created for the capsule, and the capsule is evaluated within the environment using instructions from the code object that the capsule refers to.

4.6.1 Capsule anatomy: headers and bodies

Unlike most traditional protocols, PAN takes a very minimal approach to the basic required contents of a capsule. In the current implementation, a capsule contains only the name of a code object (a cryptographic hash), and the address of a node on which the code object itself my be found. During transmission, capsules are prefixed with a link-level header (containing the next hop destination, the length of the capsule, and any other link-level information). In addition, the capsule header may also be prefixed with the version of the PAN system it corresponds to and information used for network resource management (such as a TTL field) [2]. This minimal header is followed by an arbitrary data body. See Figure 4-3 for an example.

[2]This is not implemented yet, but will be implemented before a distribution is released.

It is up to applications and code objects to decide what information goes into the data body of the capsule. A portion of the data body may be static information that doesn't vary during the capsule's traversal of the network. Other portions of the data body may be modified (and the capsule may even be extended in length) as the capsule passes through nodes. Capsule code is also be responsible for generating and verifying checksums of regions that require them, as appropriate. This approach to giving capsules control over how to utilize a uniform data space follows from the end-to-end argument [24] and from the extensible systems approach of providing a minimal set of core functionality in order to give maximum flexibility to application developers.

4.6.2 Capsule execution environments

When a capsule is received at a node, an environment is created for it. The environment provides a context in which the capsule executes. Interfaces to node resources can use this context for access checks or for aborting the capsule in the case of problems.

Using a software segment ring (see subsection 4.7.1), the environment keeps track of all of the resources that a capsule is using. This allows a capsule's resources to be freed when the capsule completes.

The environment also contains a jump point that allows the execution of code to be aborted by unwinding the stack. This is used to abort execution in the case where an operation fails or in the case that a code object tries to do something blatantly illegal.

In addition, the environment contains information about where a capsule is from. This provides a secure way of being able to tell if a capsule was inserted by an application linked to the node or if it was received over the network.

In the future, capsule environments may also contain information used for resource policy management. For example, the environment could contain information about a capsule's TTL value.

4.7 Memory management within PAN

PAN nodes, code objects, and applications all reference regions of memory containing data through a consistent *software segment* interface. The networking layers of most modern UNIX-like operating systems provide some sort of buffer-management system in order to efficiently utilize memory and in order to minimize the number of times data needs to be copied or touched (for example, mbufs within BSD[20] and sk_buffs within Linux[3]).

Software segments provide a buffer management system for PAN that is able to encapsulate the buffer management systems of different operating systems, allowing code objects to run efficiently without knowing anything about the underlying operating system. For example, within the Linux kernel implementation, a capsule can be received by PAN and then sent out across the network without any need to copy the contents of the entire capsule. Additionally, no copies are needed when a capsule places its contents into a node's soft state for future retransmission. This allows capsules to be speculatively placed into a soft state cache with almost no performance overhead.

The software segment scheme also provides a uniform resource tracking and reference counting system (e.g., as discussed later, handles to application links are a type of software segment). This allows a node to keep track of which resources a capsule or code object is using such that they can be unreferenced when a capsule terminates or a code object is unloaded.

A software segment contains a default header, optional additional header fields, and any associated resources. By default, a software segment header (see Figure 4-4) is five words long and contains a pointer to a region of data (the "contents" of the software segment), the length of the data, the type of the software segment, a reference count, a pointer to the next software segment in a chain, and a method for finalizing the software segment. The data pointer and the length must both have properly word-aligned values. The finalizer method is invoked when the software segment is freed and allows the software segment to free or dereference any resources

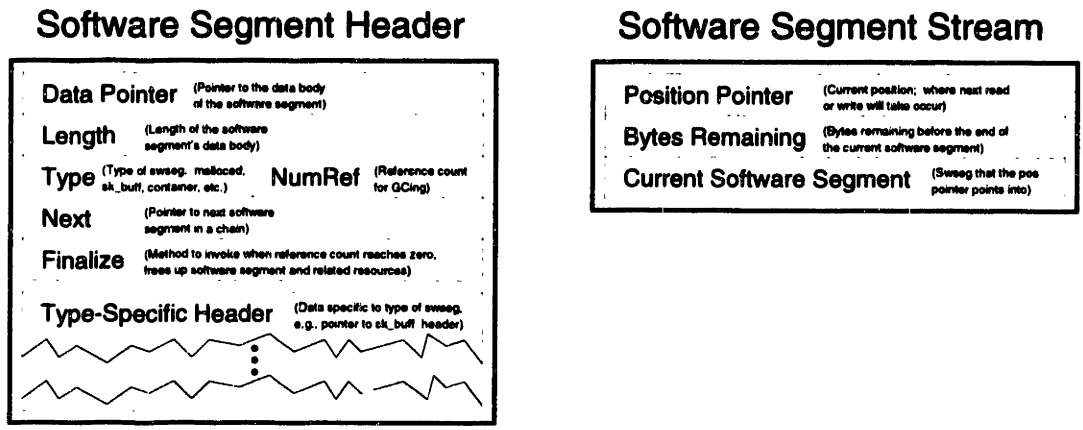

Figure 4-4: A software segment header and a software segment stream.

it may be using.

Software segments also allow capsules to be rapidly constructed by splicing together discontiguous regions of memory. By chaining together two or more software segments using the "next" field, code accessing the software segments can view the data regions in the software segments as being a single contiguous region of memory.

Different types of software segments may add additional fields to this header or may point the data pointer at different types of memory. Software segment types within PAN currently include software segments containing regions of malloced or kmalloced memory, software segments that overlay portions of some other software segment, software segments containing mmaped files (useful for writing applications), software segments that keep track of references to application links, software segments that act as containers for other software segments, and software segments that contain kernel sk_buffs.

4.7.1 Keeping track of software segments

Software segments are currently tracked using reference counting. When the refcount on a software segment reaches zero, its finalizer is called, freeing the software segment and unreferencing any software segments that it references. Although some checks are performed to prevent the creation of cycles, it is up to application writers to never

Code Object (PNI) Interfaces To Software Segments	
`pniSwseg_createByMalloc`	creates a new software segment containing a block of memory of a requested size
`pniSwseg_overlay`	creates a new software segment (or software segment chain) that overlays a portion of an existing software segment chain
`pniSwseg_chain`	chains two software segments together, making their contents appear as a single, contiguous region
`pniSwseg_getChainLength`	returns the number of software segments in a chain
`pniSwseg_getBytesInChain`	returns the length, in bytes, of a software segment chain
`pniSwseg_getSize`	returns the size of a single software segment
`pniSwseg_ref`	increases the reference count on a software segment
`pniSwseg_unref`	decreases the reference count on a software segment and finalizes the software segment if the reference count reaches zero
`pniSwseg_makePersistent`	makes a software segment persistent by referencing it and adding it to the software segment ring of the calling code object
`pniSwseg_releasePersistent`	unreferences a software segment and removes it from the software segment ring of the calling code object

Table 4.2: Software Segment Interfaces

Code Object (PNI) Interfaces To pniStreams	
`pniStream_init`	initializes a pniStream to the specified position in a software segment chain
`pniStream_seekFwd`	seeks a pniStream forward by a specified offset
`pniStream_dup`	duplicates the state of a pniStream
`pniStream_eofP`	returns whether there is still data available in a pniStream
`pniStream_getBytesRemaining`	returns the number of bytes remaining before the end of a pniStream
`pniStream_getState`	returns the software segment that a pniStream is currently accessing
`pniStream_copy`	copies data from one pniStream to another
`pniStream_read`	copies a specified amount of data from a pniStream into a buffer
`pniStream_read`	copies a specified amount of data from a pniStream into a buffer
`pniStream_write`	copies a specified amount of data from a buffer into a pniStream
`pniStream_read_net_uint32`	reads an unsigned 32 bit value from a pniStream in network byte order
`pniStream_write_net_uint32`	writes an unsigned 32 bit value to a pniStream in network byte order
⋮	similar functions exist for reading and writing both signed and unsigned 8, 16, and 32 bit values in network, little endian, and big endian byte orders

Table 4.3: Software Segment Stream Interfaces

create cyclical references. Given that reference-counting is used by such widely-used languages as Perl[32], it shouldn't place too many constraints on application-writers. In addition, reference-counting reduces the complexity and real-time problems sometimes associated with garbage collectors. However, it may be eventually necessary to have a garbage collector that occasionally runs and attempts to find memory leaks.

Capsule environments and code objects both need to be able to keep track of the software segments that they're referencing. This is performed through a common mechanism called a *software segment ring.* Each capsule environment has an associated software segment ring that keeps track of the software segment that capsule is referencing. Software segment rings are implemented as regions of memory that contain a combined list of software segments and a list of free slots in the ring. As software segments are added, the ring grows in size if there are no available slots. When a capsule finishes execution or a code object is removed from the code cache, they can easily unreference all of their associated software segments by destroying the software segment ring.

4.7.2 Why not just use sandboxing?

Some people have suggested that PAN could simply use sandboxing[31] (rewriting code to prevent memory accesses outside of a confined region) or create a single protected region of memory for each capsule to execute in. This has the problem that many PAN capsules need to share data (by either placing their contents into a node's soft state or by retrieving data from the soft state of a node). The number of copies this would require could place a substantial performance burden on the system. In addition, being able to perform sandboxing in the first place might either require substantial changes to an operating system's existing networking and buffering code, or might require all capsules to be copied into the protected memory region prior to evaluation due to memory alignment constraints.

4.7.3 Software segment streams

Because dealing with accesses to the discontiguous regions of memory encapsulated by software segments can get tedious, PAN provides *software segment streams*, also called *panStreams*, to simplify the common tasks of reading and writing sequential items to and from a chain of software segments. A panStream contains a pointer into a software segment's data region, a count of the number of bytes remaining in the data region, and a pointer to the current software segment (see Figure 4-4). Reading from or writing to a panStream results in the advancement of the stream position. Provided routines also perform bounds checking in order to know when to advance to the next software segment in a chain and in order to prevent illegally accessing memory. Because bounds checking is already done, adding a bytecode language with panStream-like primitives could only increase performance by allowing optimizations to be done. See Table 4.3 for a listing of the panStream interfaces used throughout PAN.

Functionality currently provided to code objects, nodes, and applications include routines for seeking forward in streams, copying data between streams, reading and writing arrays of bytes, performing checksum and cryptographic checksums on regions of memory, and reading and writing multi-byte data types in a platform-independent fashion (performing byte-swaps and conversions as needed). Routines are also provided for reading and writing commonly used data types, such as node addresses and code object names.

4.7.4 panSerGen: a serializer generator

Because writing code to serialize data structures into and out of panStreams can be tedious and error-prone, the PAN system provides *panSerGen*, a serializer code generator. Written in object-oriented Perl 5[32] and using a Perl version of Berkeley YACC, the serializer generator is a compiler that takes a file containing the description of a structure and outputs one or more files containing C structures and C code to read the structure contents from a panStream (performing any byte-swaps needed) or to

write the structure to a panStream (again performing any necessary byte-swapping). The description read by panSerGen can contain information about the ways in which various fields are accessed by various users of the generated code. For example, the same description file could be used to generate code for both an application and a code object. If the code object doesn't need to change (or even look at) some of the fields in the structure, panSerGen can generate code that skips over and does not bother to read in portions of the stream.

Although not yet implemented, the serializer generator could also perform some optimizations. For example, it could inline many of the panStream routines and eliminate bounds checks in the common case of reading from or writing to a single contiguous region or memory. It might also be possible to make panSerGen generate code to interleave message processing steps (such as performing checksums) along with the reads or writes[33].

4.7.5 Soft state, persistent software segments, and containers

It is often desirable or neccessary for protocols to keep a degree of state within nodes in the network. Although this state is soft state (meaning that it may disappear at any time as the network is reconfigured, as nodes fail, or as nodes expunge resources), it can still be very useful for maintaining protocol variables, capsule data for retransmission, multicast subscription tables, or just about anything else.

All soft state stored in a node is associated with a code object. Code objects can store small amounts of state (such as protocol variables) in static variables that persist for the code object's lifetime in the node.

In order to retain software segments beyond the lifetime of a capsule environment, a code object can ask the node to make a software segment *persistent.* This places the software segment in the code object's software segment ring. The code object can then safely place a pointer to the software segment in a static variable.

To easily allow data structures and collections of software segments to be stored in a node, PAN provides container software segments. A container software segment is a software segment that does not contain data, but rather contains references to other

Code Object (PNI) Interface to Hash Table Containers	
`pniHashTab_create`	creates a new container software segment as a hash table with a specified size and hashing function
`pniHashTab_lookup`	looks up a key in a hash table and returns the associated value
`pniHashTab_remove`	looks up a key in a hash table and removes the associated value
`pniHashTab_insert`	associates a key with a value in a hash table, replacing any previously existing mapping

Table 4.4: Hash Table Software Segment Container Interfaces

software segments. The data portion can be laid out as a stack, a heap, a hash table, or as anything else that may be appropriate. Each container software segment has pointers to methods for getting the size of all the software segments in the container, finalizing the contents of the container, and for releasing a certain amount of data from the container.

The only type of container currently provided is a hash table (see Table 4.4 for a list of interfaces). This is used by the portmapper (described in subsection 4.8.3) and by applications wishing to associate data with separate flows of capsules. In the future, other types of container software segments will be written to simplify the job of writing some types of code objects. However, there's nothing to keep a code object from implementing its own types of container software segments.

Although not yet fully implemented, nodes will eventually keep track of the amount of data stored within the persistent software segments of code objects. Code objects will be able to notify the node of their memory requirements: how much they minimally need to operate, how much they would ideally have access to, and the most they'd ever need. When the node becomes tight on memory, it will compare the amount of memory being used by code objects to the amount of memory they had requested. The node could then ask some code objects to free data from their

persistent state. Code objects would then be able to select which data to free from their persistent state in a manner that was appropriate to the type of data being stored. Code objects that did not free up enough persistent state would be punished by the node. This mechanism is similar to those used in the Exokernel[11]. In both systems, the desire is the same: to give applications or code objects as much control as possible over the management of their own resources.

4.7.6 Unsolved problem: safe sharing in a multi-threaded environment

One problem that has not yet been solved in PAN is how to safely share data in a multi-threaded environment. The "easy" solution would be to always copy data and to never share anything. Unfortunately, this could have a negative and substantial performance impact. There are two types of sharing that may happen within PAN: first, sharing between multiple capsules running simultaneously; second, sharing of data that has been sent out once already, and is being processed by the underlying network layer, but which is also still being used by the sender or by other capsules.

To get around the first problem, PAN is never within more than one capsule environment at a time and always processes capsules to completion (i.e., the node does not switch between multiple running capsules as this situation never arises). This may not be a good permanent solution, however, as it means that the system can't take advantage of multiprocessors.

It may turn out that this is one of the harder challenges to overcome in the creation of a safe and secure active network node. In order to ensure safe sharing, a safe language system would also need to enforce synchronized access to data, prevent race conditions, and prevent deadlock. It may not even be possible to do all of these. As a result, it may be necessary to instead develop schemes for limiting the problems resulting from code that doesn't correctly share data.

To get around the second problem, PAN relies on the authors of code objects to respect the invariant that once a capsule has been submitted to the node for

transmission across a node link, it should both be treated as read-only and should never be submitted to the node again for transmission across a node link. Although not yet been implemented, it should be possible to remove the second restriction by keeping track of when a software segment is being processed by the network layer and making a copy of it before transmission when this is in fact the case.

4.8 Application links

At the leaf nodes of a PAN network, applications use *application links* (abbreviated as *applinks*) to communicate with the nodes that they are connected to. Applications can use applinks to insert capsules into nodes for evaluation. Capsules can also use applinks to send data back to applications.

4.8.1 Application interface to application links

Applications link against a PAN Application Library (PAL) that provides routines for connecting to a node and for communicating with the node. In addition to providing routines that are common throughout PAN (such as software segment routines), the library provides the routines listed in Table 4.5.

This very minimal set of routines may be additionally supplemented in the future by utility libraries that provide demultiplexing services. In addition, utility routines for use by applications are provided with most code objects to make their use easier.

4.8.2 Code object interface to application links

The PAN Node Interface provides a minimal set of routines to code objects to allow them to send data to applications. The `pniAppLink_getSender` routine refers a handle to a *pniAppLink*, the `pniAppLink_deliverData` routine allows a capsule to deliver data to an application using a pniAppLink handle, and the `pniAppLink_isConnected` function indicates whether an application is still connected to the other side of a pniAppLink. See Table 4.5 for a summary of these interfaces.

Application Interface to Application Links	
`panNodeLink_create`	connects to node and returns a handle for communicating with it
`panNodeLink_send`	inserts a capsule into a node for evaluation
`panNodeLink_retr`	retrieves an entire capsule from a node, optionally blocking until one is received
`panNodeLink_read`	reads data from a node into a panStream
`panNodeLink_getLocalAddr`	gets the network address of the node
`panNodeLink_getSockFD`	returns a UNIX file descriptor for calling `select` on
`panAppCodeobj_load`	bootstraps code loading by using *insertco* to insert a code object into a node

Code Object (PNI) Interface to Application Links	
`pniAppLink_getSender`	returns a handle to the pniAppLink that inserted the capsule
`pniAppLink_isConnected`	returns whether a pniAppLink is still valid
`pniAppLink_deliverData`	passes data of a specified length from a panStream to an application across a pniAppLink

Table 4.5: Application Link Interfaces

A pniAppLink is a reference-only software segment that provides an unforgeable handle allowing a code object to communicate with a node. The only way for a code object to get ahold of a pniAppLink is to be running from a capsule that was inserted into the node through the applink. By storing the pniAppLink handle into the code object's state, the code object can use the handle later on to deliver capsules to the application. Because pniAppLinks can't be forged and because code objects can have control over who can access the handle, applications can have control over which code objects are able to send data to them.

4.8.3 Using a portmapper ...

Although the provided applink interface to code objects is minimal and flexible, it doesn't directly provide the same port multiplexing functionality that ANTS provides or that UNIX programmers are used to using (this is because there may be cases where port multiplexing is not needed and would just get in the way). To provide this functionality, a simple *portmapper* code object is provided with the PAN distribution along with a library of utility functions for applications (see Table 4.6 for a listing of the interfaces to the portmapper). There's nothing special about this particular portmapper — anyone could easily implement their own code object that did something similar but which provided semantics more appropriate to some particular application.

The provided *portmapper* code object acts as a guardian for a mapping of integer port numbers to pniAppLinks. An application can send a *portmapper* capsule into the node with a "bind" request. The capsule would then create a binding from a port number to the application that inserted it and would then return the port number to the application. An application wishing to act as a server could just request to listen on a particular globally-known port number.

Capsules arriving at a node with data to deliver to an application at the node would use code objects that depended on *portmapper* and would pass a port number to function within the *portmapper*, which would then return the pniAppLink associated with that port number.

Application Interface to the Portmapper Code Object	
palPortmap_bind	inserts a capsule into the node to create a mapping between an application link and a specific integer port numer or to the next free port number
palPortmap_release	inserts a capsule into the node to release the binding between an application link and a port number

Code Object Interface to the Portmapper Code Object	
pcoPortmap_bindPortnumToApplink	creates a mapping from a port number to the pniAppLink that inserted the currently running capsule
pcoPortmap_mapPortnumToApplink	takes a port number and returns the corresponding pniAppLink
pcoPortmap_deletePortnumMapping	deletes a mapping between a port number and a pniAppLink

Table 4.6: Portmapper Interfaces

4.8.4 ... or not using a portmapper

Note that protocols might not always use a portmapper. For example, a service like DNS might never have more than one server application per node. When the server starts up, it could send a capsule into the node to bind a pniAppLink within a code object used by the protocol to deliver data to the application. Rather than having to do any sort of port demultiplexing on capsule reception, the code object at the server node would immediately have a pniAppLink handle available to which to deliver data. An advantage to using pniAppLinks this way is that it not only controls who can send data to an application, but it also reduces name space conflicts by implicitly using the code object's name as a port number for demultiplexing packets.

4.8.5 Implementation of application links

The UNIX user-space node and the Linux 2.0 kernel node implementations differ in how they implement application links, although the interfaces are the same to both of them. Abstraction layers hide the implementations from both applications and code objects.

Applications communicate to user-space nodes across UNIX domain stream sockets[26]. Capsules are sent as a word indicating the length of the capsule followed by the capsule data. The `writev` call is used for sending chains of software segments, allowing the fragmented data to be sent with a single system call and to only be copied once when the kernel reassembles it into a buffer.

Applications communicate to Linux 2.0 kernel nodes using the new AF_PAN socket family. The application library connects to nodes by just using the `socket` function call. This returns a file handle that can be used just like any other file handle (e.g., it can be passed to `select`). The `getsockname` call is even used on it to find out the network address of the node. Capsules are inserted into the node using the `writev` system call and data is read back by using `read` and `ioctl(FIONREAD)`. The kernel side of applinks uses code derived from the implementation of UNIX domain sockets that's present in the Linux 2.0 kernel. When the node kernel module is inserted into

the Linux kernel, it automatically registers the new socket protocol family. Because the kernel implementation has control over what happens on both sides of the system call, it was actually almost easier to implement it in order to get the right semantics.[3]

4.9 Network links

Nodes within a PAN network communicate with each other by sending capsules across stateless (and possibly unreliable) network links (sometimes abbreviated as *netlinks*). All nodes within a PAN network are identified by unique *node addresses.* Different underlying implementations are used for the network links, depending on whether the network is made up of user space nodes (which communicate by sending UDP packets to each other) or of kernel nodes (which communicate by using a special PAN protocol layered on top of IP.

4.9.1 Node addresses

Node addresses have different formats depending on the underlying type of network being used. However, all addresses within PAN are 128 bits (16 bytes) long. This is done to make it easy to integrate IPv6 into the system, primarily by ensuring that code is written that treats network addresses as blocks of memory rather than as single words.

For user space nodes, the node address contains the 32-bit IP address of the host on which the node resides, followed by the 16-bit port number of the UDP socket at which the node is listening for capsules. Both numbers are in network byte order and the rest of the address is padded with zeros. By including a port number in the node address, multiple user space nodes are able to run simultaneously on the same host. This allows entire networks to be simulated on only one machine (although using more

[3]Writing this code made me realize that quite a bit of the code in the Linux kernel is fairly readable. If the man pages are unclear about the semantics of a system call, it may just be easier to read the kernel source in order to figure out what's going wrong. Since then I've applied this approach and have suggested it to a few others and have discovered that it can actually be easier and faster to look at the kernel source than to write test cases!

Code Object (PNI) Interface to Network Links	
`pniNetwork_sendCapsule`	sends a capsule of a specified size to a single destination
`pniNodeAddr_read`	reads a pniNodeAddr from a pniStream
`pniNodeAddr_write`	writes a pniNodeAddr to a pniStream
`pniNodeAddr_compare`	compares two pniNodeAddrs to see if they're equal
`pniNodeAddr_getLocalAddr`	returns the address of the current node
`pniNodeAddr_dup`	duplicates a node address

Table 4.7: Network Link and Node Address Interfaces

works as well). This allows code objects and applications to be easily prototyped. However, it does not lend itself well to being used for measuring performance.

For kernel space nodes, the node address contains just the 32-bit IP address (in network byte order) of the host on which the node resides. In machines with multiple interfaces with different addresses, only one of them is used as the node's address.

4.9.2 Code object interface to network links

When capsules are received by a node, either across a network link or an application link, the `accept` method of the capsule's code object is called. It is passed a panStream pointing to the start of the capsule along with an integer indicating the length of the capsule.

To send a capsule out towards another node, a code object calls the `pniNetwork_sendCapsule` function and passes it the destination node address, a panStream pointing to the start of the capsule, and an integer indicating the length of the capsule (see Table 4.7). The capsule is then sent out towards the destination node address. If there is a route to the destination address, the capsule travels there and is evaluated when it arrives. If any of the routers located along the route support

PAN, they also evaluate the capsule. If the node has a routing table listing active nodes, the node may decide to send the capsule to some active node that is along the route (but possibly one or two hops off of it) rather than sending the capsule directly to the destination. Although not yet fully implemented, this will allow active nodes to be placed near high-performance internetwork routers so as to be able to take advantage of active protocols (such as data caching or multicast) without requiring that high-performance internetwork routers know about or be able to specially handle PAN capsules.

There is currently no support for multicasting or broadcasting capsules to multiple nodes, although this hopefully will be added to a later version.

4.9.3 Implementation of network links

The UNIX user space implementation uses UDP sockets to send capsules between nodes. Each node listens at a particular port, sends capsules with **sendmsg**, and receives capsules with **recvmsg**. By using **iovecs** with **sendmsg**, a chain of software segments can be sent, avoiding any need to copy the data to a separate buffer before passing it into kernel space. Using **recvmsg** allows capsules to be received into a chain of software segments, allowing a minimum amount of memory to be used to hold the capsule without needing any a priori knowledge of the size of the capsule that is being received. Each node configures itself and builds up a routing table from a configuration file written in the Scheme[6] programming language and parsed using the SIOD embeddable Scheme interpreter. The routing table is inserted from the configuration file into the node with a single Scheme function call. Scheme is used so that reasonable sized test-beds of nodes can be configured using a single configuration file which contains a description of the network topology along with some Scheme code to generate routing tables for each node.

The Linux 2.0 kernel implementation sends capsules between nodes using a special PAN protocol that is layered on top of IP (i.e., at the same layer as TCP, UDP, and IPIP encapsulation for IP tunneling). This allows IP to be used to transport packets between nodes without either interfering with other IP-based protocols and without

having the additional overhead that would be imposed by having an existing protocol, such as UDP, encapsulate capsules. The Linux kernel also provides a clean interface to this level of the networking stack, allowing new IP protocols to be dynamically registered and unregistered.

Capsules are received by handling IP packets received using this new protocol type. PAN creates a sk_buff software segment that encapsulates the capsule data of the packet's sk_buff, provided by the kernel's networking layer, and passes this software segment into the capsule's code object. If this same software segment is passed back to the node to be transmitted somewhere else (as happens in the common case of capsules just looking at their contents and sending themselves on towards their destinations), the node uses the sk_buff already contained within the software segment for retransmission. The software segment is only copied into a new sk_buff if the software segment is a new one that was allocated by the capsule, or if the sk_buff is in the process of being transmitted by the kernel's networking layer. This minimizes the number of times that capsule data needs to be copied.

In the common case of a capsule that fits within an unfragmented IP packet and which only forwards itself on, the data in the capsule is never copied between being received by the network interface and being sent back out through another network interface. For some network drivers, untouched capsule data may never even be brought into the CPU or any caches.

Nodes send capsules to the network by appending an appropriate IP header to them before calling the `ip_forward` function exported by the kernel. Note that PAN nodes let the IP layer deal with fragmenting and unfragmenting packets that are larger than the maximum size that the underlying network interface can handle. The maximum capsule size is constrained by the maximum size of an IP packet, however. At some point it may make sense to allow code objects to query the node as to the largest size capsule that may be sent without fragmentation occurring, allowing them to perform fragmentation when appropriate in order to improve performance — PAN capsules need to be reassembled and fragmented at each node they travel though, while IP routers don't typically do this unless they need to in order to apply firewall

rules.

4.10 Miscellaneous utility functions

In addition to all of the other interfaces that nodes provide to code objects, nodes also export a number of miscellaneous utility functions. These interfaces include routines for logging error messages, data structures for a few software segment container types, and routines for obtaining and working with time stamps. The time stamping routines are particularly useful for writing benchmarks as they allow for much more accurate calculations of the times at which capsules depart and arrive at nodes. See Table 4.8 for a listing of some of the provided utility functions.

4.11 Differences between PAN and ANTS

In order to make it easier to port ANTS applications to PAN, the two systems use the same capsule-oriented programming paradigm and have similar interfaces. However, a number of ANTS primitives do not exist in PAN. Instead, lower-level primitives that are more flexible are provided. In all of these cases, the ANTS primitive can be implemented in terms of the comparable PAN primitive. This follows from the PAN design philosophy of providing the maximum flexibility to code objects in order to allow them to do exactly what they need to do.

There are four primary differences between the interfaces provided by PAN and ANTS: PAN has a more flexible system for assembling protocols from collections of code objects, has fewer required elements in the header of capsules, does not have a global soft state cache, and provides a simpler and more flexible interface for allowing capsules to deliver data to applications.

4.11.1 Protocols as collections of code objects

In ANTS, a protocol is a collection of classes that are bound together, and access privileges are granted to entire protocols rather than to individual classes. This

Code Object (PNI) Interface to Time Stamps	
`pniTimestamp_getCurrent`	gets the current time and stores it into a pniTimestamp
`pniTimestamp_getResolution`	returns the approximate resolution of the node's time stamping functionality
`pniTimestamp_read`	reads a pniTimestamp from a pniStream
`pniTimestamp_write`	writes a pniTimestamp to a pniStream
`pniTimestamp_compare`	compares two pniTimestamps and returns whether one is greater than the other or if they are equal
`pniTimestamp_subtract`	calculates the difference between two pniTimestamps and stores the result into a third pniTimestamp
`pniTimestamp_dup`	duplicates a pniTimestamp

Code Object (PNI) Interface to Logging Functions	
`pniLog_debug`	logs a string at the debug priority level
`pniLog_msg`	logs a string at the message priority level
`pniLog_warn`	logs a string at the error priority level
`pniLog_error`	logs a string at the warning priority level

Table 4.8: Miscellaneous Utility Functions

artificial abstraction limits what can be done with the system by making it difficult for multiple protocols to share data or otherwise interact.

Within PAN, protocols have no tightly defined boundaries. A protocol is simply a collection of code objects that chose to work together. Because access to soft state is on the granularity of code objects rather than protocols, two protocols can share data by simply sharing a code object in common to act as a guardian for the shared state.

Although not yet implemented, a code object will eventually be able to have control over which other code objects can depend on it. This will be implemented by providing a mechanism where by code objects can be signed using the secret key of a public key pair. By distributing a list of public keys along with a code object, the code object will be able to verify that the signature on some other code object matches one of the public keys before along the other code object to depend on it, and therefore access symbols from it. Using this mechanism, it will be possible to create cryptographically sealed protocols and groups of protocols that can be dynamically extended and that allow for secure data sharing between protocols.

4.11.2 Minimal requirements on capsule contents

ANTS capsule headers contain information such as the original source of the capsule and the destination of the capsule by default. Putting an original source field in the header may be useful for debugging, but it isn't terribly useful for implementing security policies.[4]

PAN uses a code object home address in capsules rather than using the capsule's source address as the home address. In the simple case where they are the same, the

[4]It is hard to guarantee that the source address has not been spoofed. If we had guarantees about the source address, we might be able to use it for access control. In order to provide such guarantees, the source node would have to sign the capsule with its public key. This would have the side effect that capsules either could not modify their contents or that each intermediate node would have to verify the signature on reception and then resign the modified packet on dispatch. There may also be other possible schemes to provide partial security, such as having the source node only sign a timestamp in the header. The danger with the latter approach is that it could lead to a false sense of security.

same functionality is provided. However, using a code object home address allows code objects to be retrieved from a location other than the originator of the capsule. For example, this may be desirable in cases where the capsule originator is separated from the rest of the network by a very low-bandwidth link.

In addition, PAN does not require that a destination address be contained in capsule headers. This is because some protocols, such as those for multicast, may not involve the concept of a single destination.

4.11.3 No unified soft state cache

Unlike ANTS[35], PAN has no single, unified soft state cache. This is because of the realization that different code objects have very different needs and requirements for how they store data within a node's soft state. Some code objects may want to store capsules in a cache for retransmission, others may just need to keep a few protocol state variables around, others may want to keep a list of capsule fragments, and others may want to keep lists of multicast subscribers. The requirements for how these objects are keyed in the cache, how long items stay in the cache, and how objects are selected for removal may vary greatly between different code objects (or even within a single code object). In order to keep PAN as flexible as possible, the system allows code objects to make software segments "persistent," meaning that they become associated with a code object so that they outlive the lifetime of a single capsule (i.e., they are placed into the code object's software segment ring). For very small amounts of state global to a code object, the code object can even store the data in a static (or global) variable. In fact, code objects usually store the pointers to persistent container software segments in static variables. Note that "persistent" state is still soft state as code objects are periodically forced to free up some of their memory usage.

To provide functionality similar to the soft state cache provided by ANTS, PAN provides a type of container software segment that acts as a hash table. However, code objects can have control over the hashing and cache replacement functions utilized by this container.

4.11.4 More flexible interface to application links

ANTS demultiplexes data destined for applications using integer port numbers in a fashion similar to that employed by TCP and UDP sockets. As described in Section 4.8, PAN provides the much simpler interface of requiring that handles to application links be used by code objects for passing data to applications. The only way for a code object to obtain a handle to an application link is through a capsule that was inserted into the node by the application. This allows applications to maintain tight control over which protocols can deliver data to them. It also allows protocols to use whatever capsule demultiplexing scheme is most appropriate to them.

To provide compatability with ANTS, a portmapper code object is provided along with a library of functions for use by applications.

Chapter 5

Experimental Procedures and Results

The performance of active network nodes, both in throughput and latency, is likely to have a significant impact on the practicality of active networks. Ideally, an active network node should be able to handle the base case of processing a capsule that simply asks to be forwarded to a destination with as little overhead as possible when compared to a traditional network node that simply routes packets on towards a destination.

This chapter analyzes the performance of two implementations of a PAN node: a user-space node that encapsulates capsules within UDP packets and a loadable Linux kernel module that encapsulates capsules within IP packets. The performance characteristics of both nodes are measured and are compared against the performance characteristics of a Linux workstation acting as a router. Measurements are taken of both the latency incurred by the active networking system and of the total throughput of the system.

These experiments clearly demonstrate that avoiding data copies, using compiled code objects, and implementing the node within the kernel are all critical to achieving high performance.

5.1 Experimental setup

All experiments were performed on a testbed network consisting of three Linux workstations connected together with dedicated 100 Mbps Fast Ethernet network links. The testbed consists of two end nodes, called *sender* and *receiver*, that may either be connected together either directly or through a third node called *middle*, depending on the experiment being performed.

Each machine contains a 200MHz Intel PentiumPro processor, an ISA/PCI motherboard with a 440FX chipset, and 64MB of RAM. The sender and receiver nodes each contain a single DEC DS21140 Tulip-based SMC EtherPower 10/100 network card running in 100 Mbs half-duplex mode. The middle node contains two of these Tulip network cards. In addition, each machine has a PCI video card, PCI NCR SCSI controller, and an ISA SMC EtherEZ 10 Mbps network card (used for accessing the machine from remote). The machines are connected together using either of two Intel Express 100BaseTX hubs.

Each workstation is running the RedHat 4.2 distribution of Linux using an unmodified Linux 2.0.32 kernel from the `kernel-2.0.32-1` package distributed by RedHat. Version 0.79 of Donald Becker's tulip driver is used with the Fast Ethernet cards.

Depending on its role in the experiment being performed, each node runs either the PAN Linux kernel implementation, the user-space PAN implementation, a simple UDP packet forwarder written in C, or is just using Linux's IP forwarding functionality. In all experiments, a client application runs on the sender node and sends capsules towards the receiver node which evaluates the capsules and optionally sends back a response depending on the experiment being performed.

Seven different testbed configurations are used during the tests of PAN (see Figure 5-1). In the *kernel/active* configuration, all three nodes are running the PAN kernel implementation, and capsules sent between the sender and receiver are processed by the middle node. In the *kernel/passive* configuration, the sender and receiver nodes are running the PAN kernel implementation but the middle node is just using the Linux kernel's IP forwarding to forward packets between the sender and receiver.

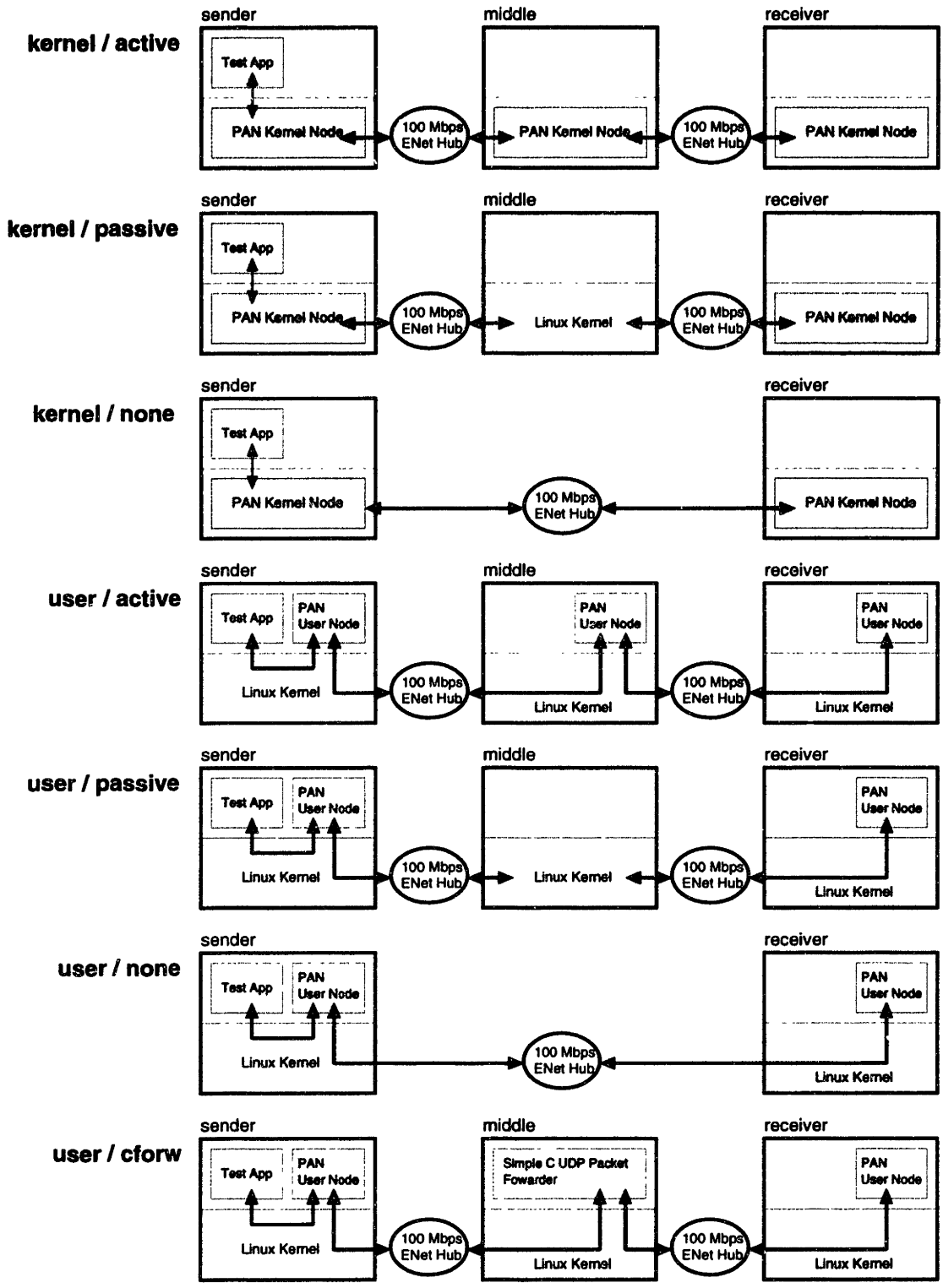

Figure 5-1: Seven different testbed network configurations are used for measuring the performance of the user-space and kernel-space PAN implementation.

The *kernel/none* configuration does not contain a middle node and directly connects the sender and receiver node, both of which are running the PAN kernel implementation. The *user/active*, *user/passive*, and *user/none* configurations are identical to the corresponding kernel node configurations, but use the user-space PAN implementation rather than the kernel implementation. Finally, the *user/cforw* configuration has user-space PAN nodes running on the sender and receiver and has a simple user-space UDP forwarder written in C running on the middle node. In this configuration, all traffic between the sender and the receiver pass through the UDP forwarder. Comparing results between the different configurations provides insight into where various overheads are coming from.

The UDP forwarder uses the same system calls as the user-space PAN implementation, but doesn't do any active processing. This configuration gives insight into how much of the cost of user-space PAN is due to active processing as opposed to how much of the cost is due to the overhead of transferring data to and from user-space.

The PAN kernel implementation uses most of the same code path as Linux IP forwarding, but adds hooks in to perform active processing of the capsule, allowing it to take actions other than just forwarding itself on towards a destination. As a result, the base case performance of PAN will always have slightly lower performance than standard IP forwarding.

In addition to the seven basic configurations, latency experiments are also run in a *kernel/activecopy* configuration that is identical to the *kernel/active* configuration except for using a slightly modified PAN kernel node for the middle node only. Normally, PAN doesn't need to copy or touch the contents of capsules, except for capsule headers. The middle node used in the *kernel/activecopy* configuration copies all data before sending it in order to see how much is actually gained by PAN's memory management system.

To see how packet size effects performance, all experiments are run across a range of packet sizes:

Packet Sizes Used In Experiments			
128	bytes	1504	bytes
256	bytes	2048	bytes
512	bytes	4096	bytes
1024	bytes	8192	bytes
1500	bytes		

In experiments with the kernel implementation of PAN, these sizes include the 20 byte IP header. In experiments with the user-space implementation, these sizes include both the 20 byte IP header and the 8 byte UDP header. This is done to avoid overly penalizing the user-space implementation for having a larger header size. Because 1500 bytes is the MTU of Ethernet, the kernel's IP layer fragments and later reassembles any packets larger than 1500 bytes. Measurements are taken with both 1500 and 1504 byte packets in order to better see the discontinuity caused by packet fragmentation. Note that the Linux kernel needs to copy packet data in order to fragment packets.

The current implementation of PAN is essentially untuned. It should be possible to achieve substantial performance gains by analyzing and reducing existing performance bottlenecks.

The current implementation of PAN doesn't provide safety guarantees or interoperability across different types of processors. However, the current PAN implementation performs many of run-time bounds checks that a bytecode system would need to perform. As a result, the performance of a safe and interoperable node that uses a load-time compiled bytecode shouldn't be significantly worse than the performance of the existing system. The computation involved in loading code objects and compiling bytecode only happens the first time that a node sees a new type of capsule. As a result, this computation isn't likely to be in the critical path.

5.2 Measuring latency

Latency is measured by using an active ping application which was written explicitly for this purpose. The active ping application behaves similarly to the UNIX *ping* utility that uses ICMP ECHO responses to measure round-trip times. Active ping works by sending a ping capsule from a sender to a receiver. On reaching the receiver, the ping capsule sets a state variable indicating that it has reached its destination and then sends itself back towards the source. When it reaches the source, the capsule uses the *portmapper* code object to deliver the ping response back to the application. In order to obtain more accurate measurements with lower variances, the ping capsule time stamps itself within the sender node and then calculates the round trip time on arriving back at the sender node. Each ping capsule consists of a 100 byte header followed by a data body that fills up the rest of the packet.

In addition to being a benchmark, this active ping application demonstrates how an active network can provide functionality that is special-cased in traditional networks (ICMP ECHO responses) without requiring any special support in the network infrastructure.

For each testbed configuration and packet size, 10,000 ping capsules are sent and their round trip times are averaged. These experiments are repeated three times each, the median of the three trials is taken and used. Before any experiments are performed, the code objects needed by ping are loaded into all of the network nodes.

5.2.1 Latency results

Figure 5-2 shows the end-to-end round trip times of ping capsules under different network configurations. The discontinuity between 1500 and 1504 bytes is due to packets being fragmented.

In Figure 5-3, the per-capsule latency incurred by the middle forwarding node is shown. This latency is calculated by halving the difference between the round trip times for the *none* configurations and the corresponding *active* and *passive* configurations. With 128 byte packets, just passive IP forwarding takes about 50 microseconds.

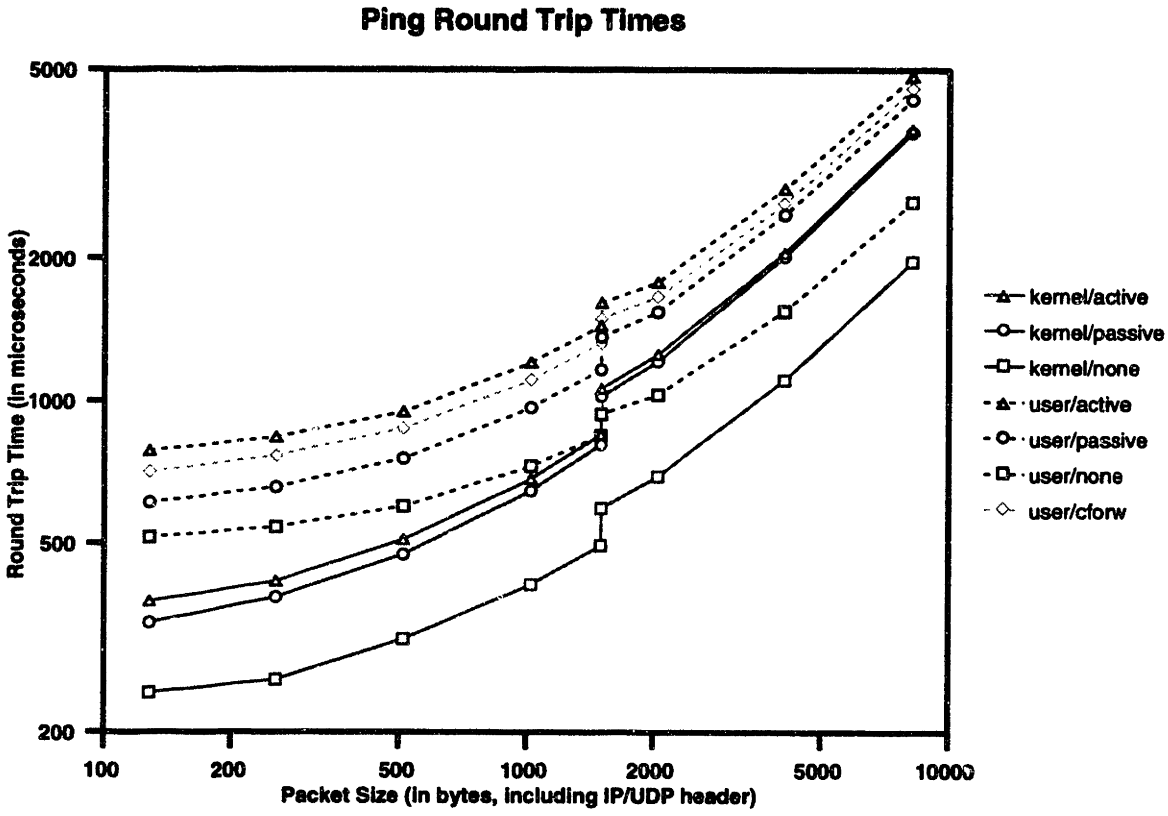

Figure 5-2: End-to-end ping round trip times. Both axes use a logarithmic scale.

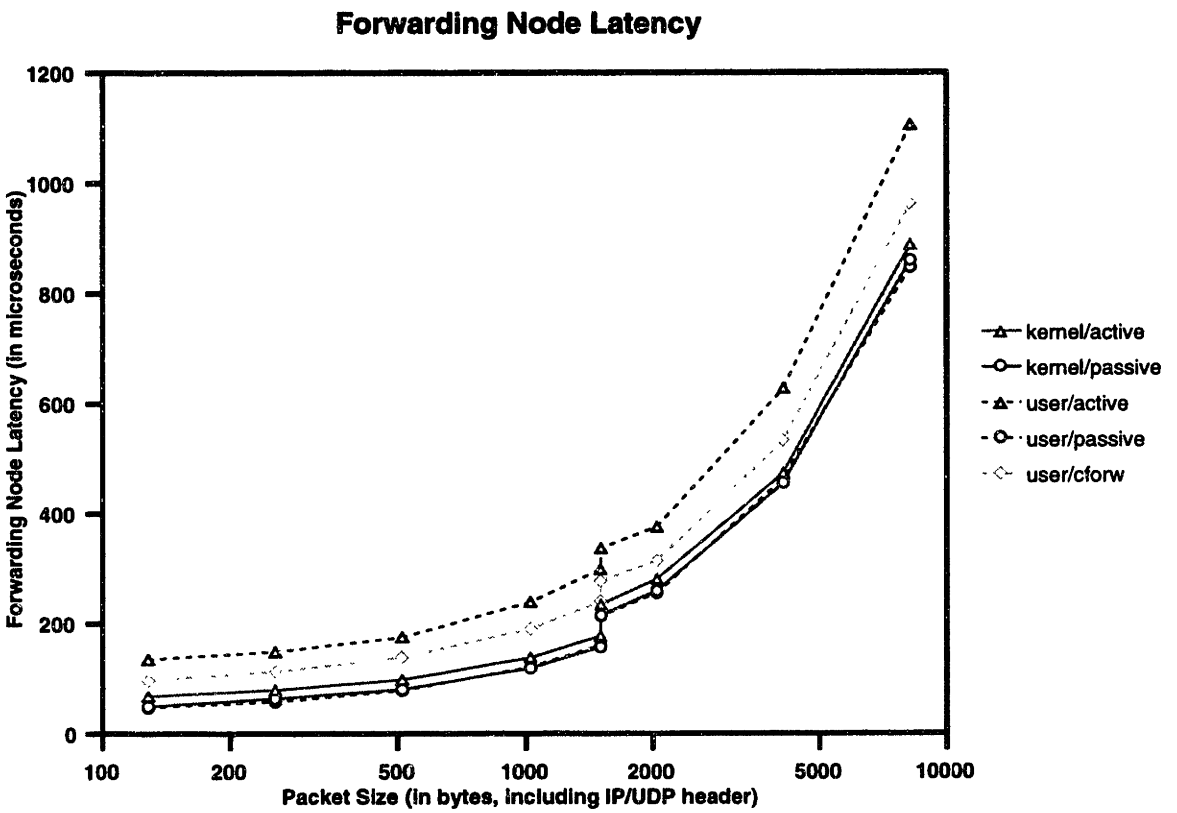

Figure 5-3: Latency per capsule incurred by forwarding node. The horizontal axis uses a logarithmic scale. The difference between *kernel/passive* and *user/passive* is due to noise in the measurements since both are measuring the same quantity.

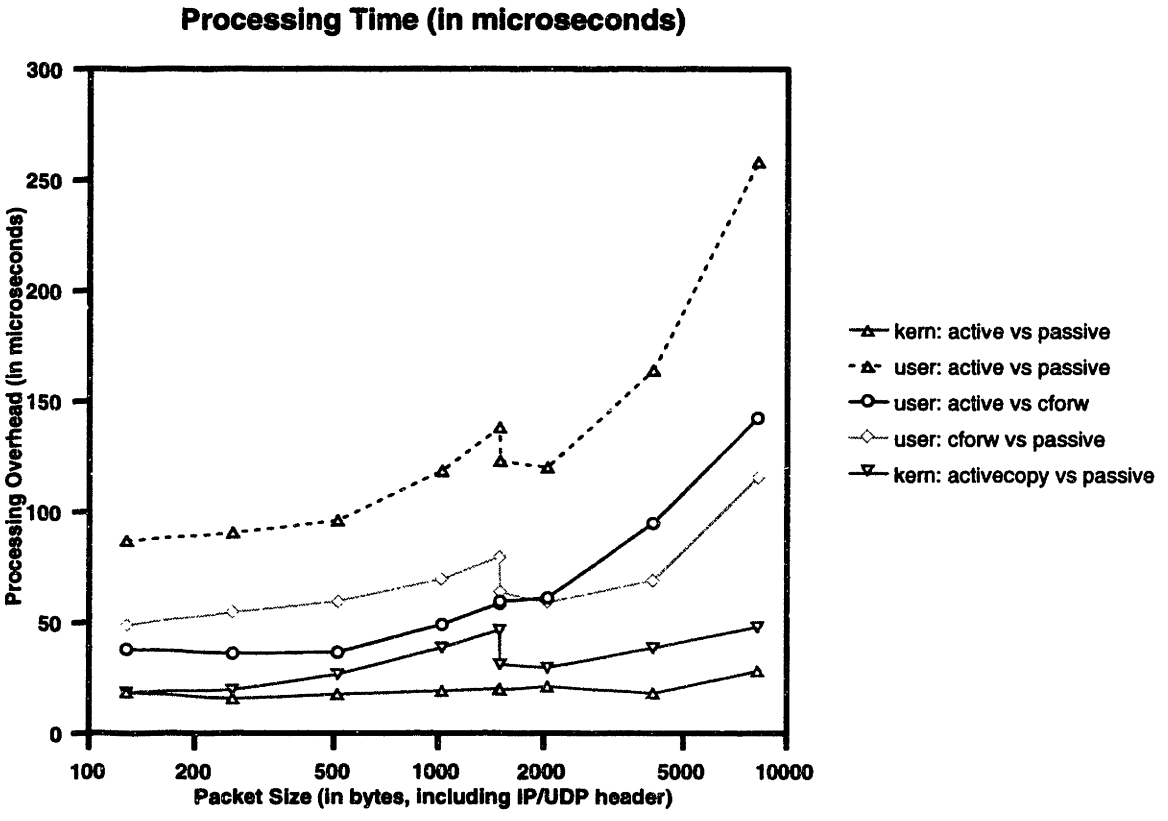

Figure 5-4: Overhead (in microseconds) for forwarding each capsule, relative to passive or C forwarder. The horizontal axis uses a logarithmic scale.

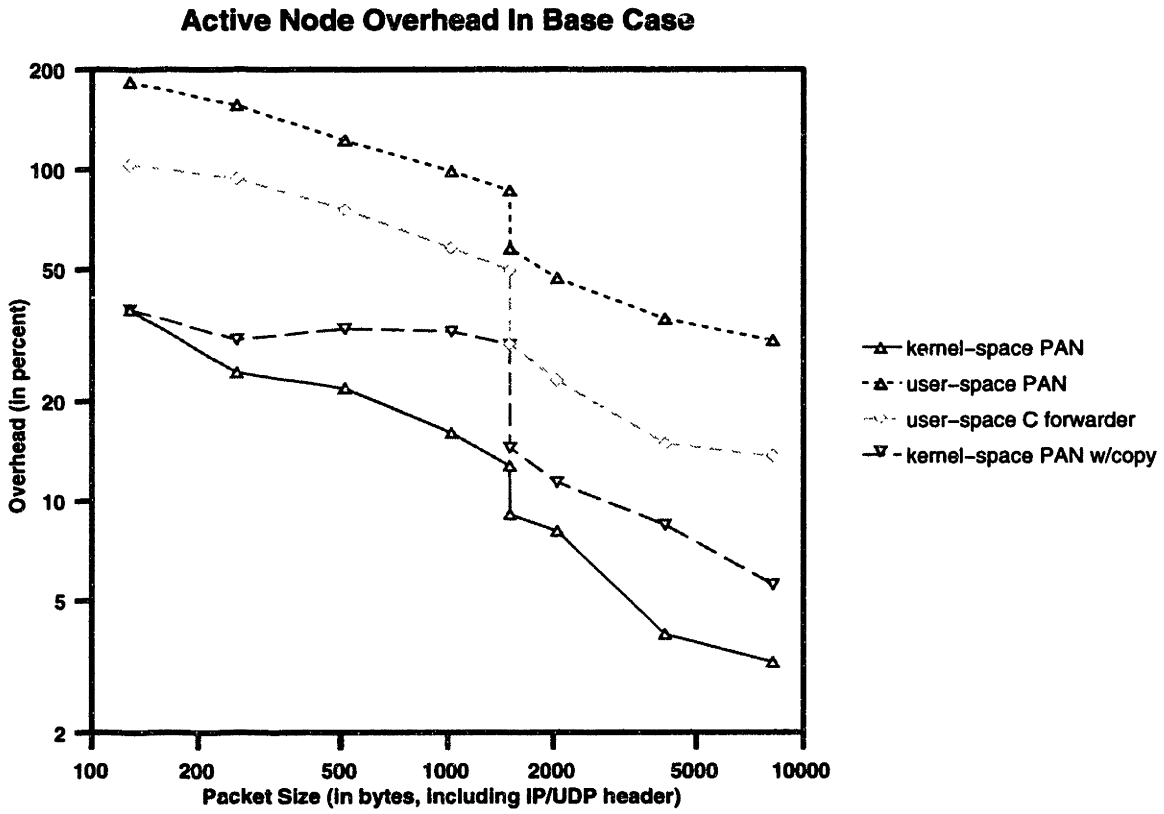

Figure 5-5: Percent overhead for forwarding, relative to passive forwarder. Both axes use a logarithmic scale.

For 1500 byte packets the passive forwarding time increases to about 160 microseconds.

Figure 5-4 shows the overheads incurred by various configurations relative to either the *passive* or *cforw* configuration. Because the PAN kernel implementation doesn't touch the contents of capsules, the overhead of *kernel/active* relative to *kernel/passive* remains around a constant 20 microseconds, regardless of packet size.

This can be contrasted to the *kernel/activecopy* configuration which has an overhead that grows with packet size. For 1500 byte packets, the overhead of *activecopy* relative to passive forwarding grows to over 45 microseconds. Thus, the cost of just copying a packet (about 25 microseconds for a 1500 byte packet) is larger than the entire overhead of active capsule processing. The cost of copying data actually drops for packets larger than the MTU because the overheads are computed relative to the passive forwarding node which is incurring the cost of dealing with IP fragmentation.

The user-space PAN implementation has a higher overhead than the kernel implementation, especially for large packet sizes. Because of the need to copy each packet to and from userspace, the shape of the curves for the user-space implementation's overhead and the UDP forwarder's overhead are similar to the shape of the curve in the *kernel/activecopy* implementation. For 128 byte packets, the user-space implementation has an overhead relative to the IP forwarder of about 87 microseconds, or over four times the overhead of the kernel implementation. For 1500 byte packets, this overhead increases to almost 140 microseconds, or seven times the overhead of the kernel implementation. Even when compared against the user-space UDP forwarder, the user-space implementation still has a considerably higher overhead than the PAN kernel implementation. It is not immediately clear why this is the case.

Finally, Figure 5-5 shows the overheads as percentages relative to the passive IP forwarder. This shows that the relative cost of using an active network drops substantially as packet size grows. This happens because the passive IP forwarding latency increases with packet size while the active processing latency remains constant. At its worst, the kernel node has an overhead of 38 percent for 128 byte packets. For 1500 byte packets, the kernel-space node is only incurring a 13 percent overhead. For

8192 byte packets, this overhead drops to only 3 percent.

5.3 Measuring throughput

The throughput of PAN nodes is measured using an active flood application that pushes capsules across the network as rapidly as possible. The flood application first inserts a *flood* capsule into the source node with an indicator that marks it as the start of a flow. This capsule forwards itself to the receiver node where it creates an entry for the flow in soft state, keyed by the address and port number of the originating application. In this entry, the capsule places the starting time of the flow. The flood application continues to send capsules towards the receiver where each of them accesses the soft state entry for the flow and increases the count of received capsules. At intervals specified in the starting capsule, status capsules are sent from the receiver towards the source that contain the elapsed time since the start of the flow along with the number of capsules received since that time. Only periodic updates are sent to minimize the effect these replies have on measurements being made. Each flood capsule contains a 100 byte header followed by a data body that fills up the rest of the packet. This application is another good demonstration of how active networks make it easy to write network applications and protocols that would otherwise require extending protocols or deploying servers.

Because flow control and reliable communications protocols have not yet been implemented on top of PAN, measuring throughput can become difficult due to packet loss. Experiments were performed by sending as many packets as could be sent without experiencing significant amounts of packet loss that interfered with the measurements. Measurements were taken three times for each network configuration and for each packet size. The median of these three measurements was then taken and used. So that code object load time isn't a factor, all code objects needed for the tests are loaded into all of the nodes before running the experiments.

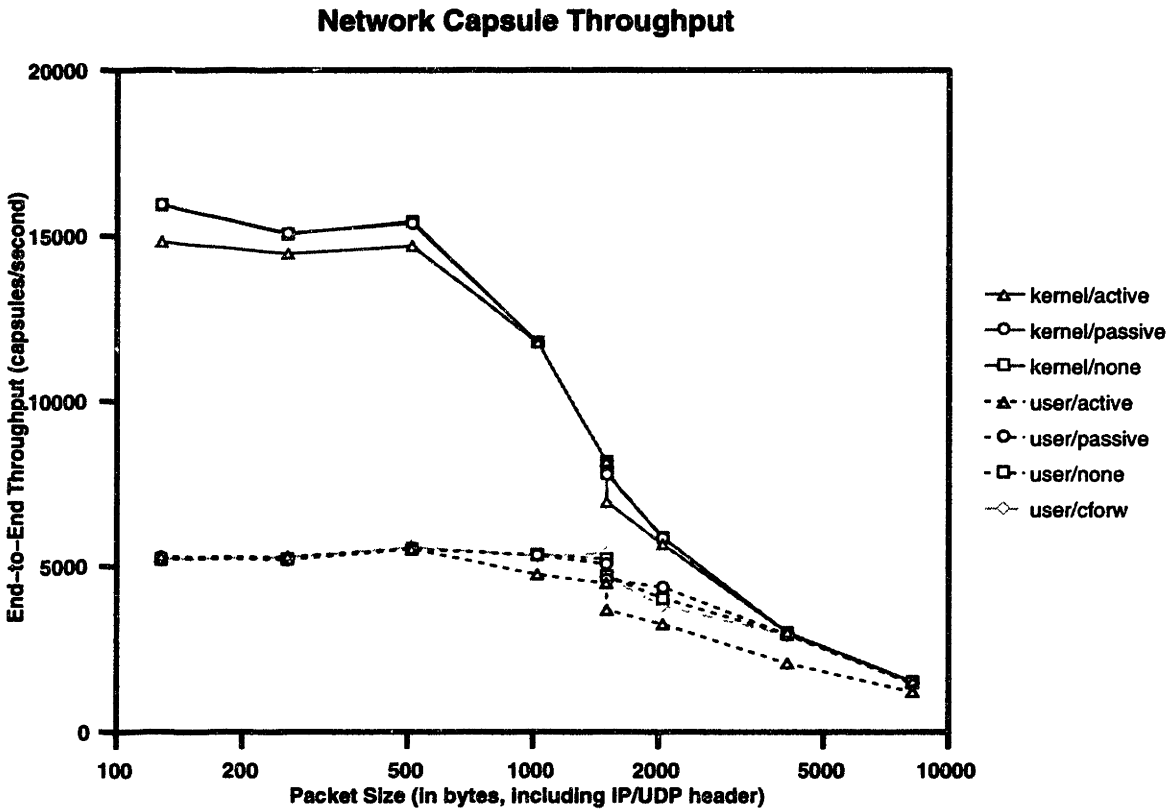

Figure 5-6: Flood throughput in capsules per second. The horizontal axis uses a logarithmic scale.

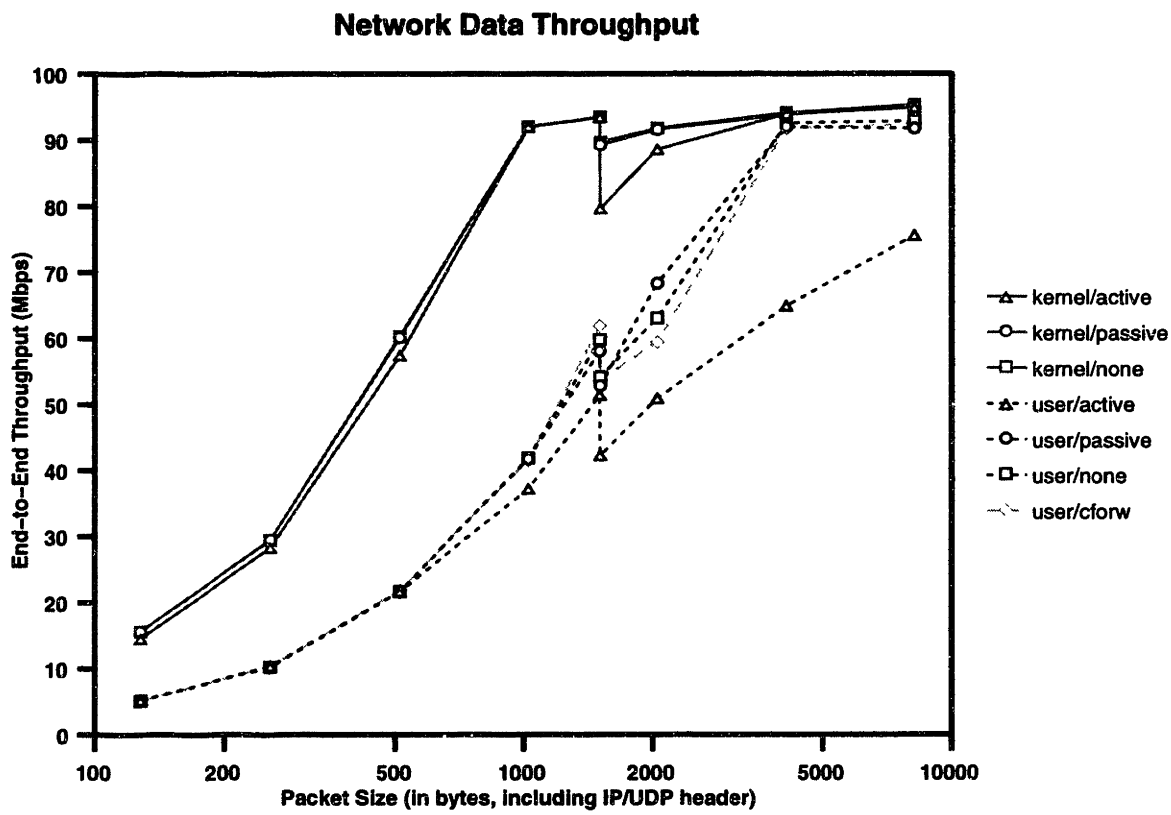

Figure 5-7: Flood throughput in megabits per second. The horizontal axis uses a logarithmic scale.

5.3.1 Throughput results

Figure 5-6 shows the throughput measurements in capsules per second while Figure 5-7 shows the throughput measurements in megabits per second.

For smaller packets, the measurements of the user-space implementation are limited by the rate at which the sender was able to insert capsules into the network and therefore does not provide a good indication of the potential throughput of the middle forwarding node. This is indicated by the measurements showing that the *user/passive* and *kernel/passive* configurations had significantly different throughputs for packets smaller than about 4096 bytes. In a system where end nodes and network collisions are not limiting factors, both should have had the same throughputs.

For both the kernel and user-space nodes, throughput for small capsules is limited by the processing time required for the node to process the capsules. For the kernel node, these processing times are on the same order as those measured in the latency experiments in the previous section. For capsules larger than about 800 bytes, the throughput of over 90 Mbps is high enough to effectively saturate the Fast Ethernet network. Prior to this saturation, a small performance difference of under 8 percent can be seen between the throughput of the PAN node and the IP forwarding throughput. After the network saturates, both configurations have very similar throughputs.

Chapter 6

Future Work

As the field of active networks is still fairly new, there is a huge amount of work that needs to be done before an active network infrastructure can be globally deployed. Even in the much shorter term, there are many things that can be done with and to PAN to both better understand its performance characteristics and to enhance its functionality.

6.1 Further experiments and optimizations

The current implementation of PAN is still untuned. By performing microbenchmarks and profiling on various subsystems of PAN, it should be possible to find out the sources of current performance bottlenecks. In addition, some of these bottlenecks may give insight into what aspects of PAN are causing the current performance overheads. For example, by modifying the serializer generator to generate code that is better optimized for the common case, it should be possible to eliminate a substantial number of function calls and bounds checks.

This thesis also does not analyze the performance of the object code loader and linker. However, current benchmarks seem to indicate that loading a code object into the node takes on the order of one or two milliseconds. Further analysis still needs to be performed on the overhead that code object loading has on the overall performance of the system.

It would also be worthwhile to compare the performance of PAN and ANTS using the same experimental setup for both systems. Preliminary measurements indicate that the PAN kernel implementation is considerably faster than ANTS. However, controlled experiments have not yet been performed to compare the performance of the systems.

The reliability of the throughput benchmarks presented in this thesis is somewhat limited by the fact that only one sender was available for pushing packets through the middle node. At some point it would be worthwhile to run the experiments in an environment that provided more reliable measurements of the sources of throughput limitations.

6.2 A bytecode language for safety and interoperability

PAN's use of Intel ix86 object code inhibits code object interoperability and safe execution. The current system is designed so that a safe bytecode language could be added to the system without substantial redesign. However, any intermediate bytecode used should be capable of high performance. In the context of PAN, this means that the bytecode must be rapidly translatable to executable code for a variety of hardware architectures, the system must be small and simple enough to fit into an operating system kernel, and the system must be able to create protected execution environments quickly with very little overhead. In addition, it must be able to manage memory in a way that allows packets to be received, shared, and sent without requiring their entire contents to be copied or touched. The system must also be able manage memory in a way that both works in a real-time environment and is efficient in a very high throughput environment where memory regions are being rapidly created and released. Non-asynchronous garbage collectors will not work in the PAN environment because of the latency they would add to capsules trying to pass through a node while it was garbage collecting.

Although the Java Virtual Machine (JavaVM) is popular and trendy, it is a large system (the book that informally specifies it is over 450 pages) that may be hard to fit easily into an operating system kernel. In addition, it is not as easy to verify as some other systems and it has a number of security problems inherent to its design[7]. It may also not be possible to achieve many of the active network performance requirements within Java. On the other hand, using the JavaVM bytecode would not only increase PAN's compatibility with ANTS, but could greatly increase the size of PAN's audience. There are also a large number of development tools already available for working with JavaVM bytecode.

There are a large number of other possible mobile code systems, such as Juice[13] [18], that might be useful, if only for inspiration. Other mobile code and dynamic code generation technologies that may be useful include OmniWare[1], Proof-Carrying Code[22], Sandboxing[31], and VCODE[9]. These are compared in a number of active networking papers including [34] and [29].

In the end, the best solution may be to design a mobile code system designed specifically to be simple, flexible, and fast in an active network environment.

6.3 Safety and security

The current implementation of PAN does not yet provide either node or global network safety or security. It is critical that this be added at some point. Using a safe bytecode system for interoperability will also provide node safety and security. In addition, cryptographic signing of code objects may be used to provide a degree of security. In particular, it will allow the creation of securely sealed protocols.

Providing global network security is a much harder challenge. In particular, being able to distinguish bad behavior from good behavior can be very challenging when only provided with a local view of a single node. A number of approaches are under examination, including using an extension of the concept of a time-to-live (TTL) field. The problem with this is allowing valid protocols to scale without also allowing worms to scale as well, whether they are maliciously written or just the result of

buggy coding.

6.4 Resource management

The current implementation of PAN does not manage resources terribly well. Some of this is because it is not entirely clear what the best ways of performing some sorts of resource management are. In particular, the code cache needs a replacement algorithm that periodically evicts unused code objects. The memory management system for evicting data from soft state, described in subsection 4.7.5 also needs to be implemented and experimented with.

In order to allow capsules to be processed in parallel on SMP machines, a system needs to be developed for forcing code objects to safely share data in a synchronized fashion (for example, setting appropriate locks and avoiding deadlock conditions).

In addition to memory management, nodes need to a way of managing network bandwidth. Some system of flow control needs to be implemented to deal with the case where an application is inserting capsules faster than they can be sent out across the network. A system also needs to be implemented to handle capsules that arrive faster than they can be processed. Ideally, PAN would allow code objects to have control over how to handle capsules, code objects, and application links in these situations. This would allow code objects to take actions such as dropping certain frames of video sequences but not others.

CPU utilization also needs to be bounded in some way, either by having timers abort code objects that have been running for too long or by having the safe code system check how long it has been running whenever a backward jump or costly operation is performed.

6.5 Applications and programming models

Before active networks can be successful, they need applications. PAN provides a good testbed for developing new active network protocols. It should be possible to

write a wide range of new protocols on top of PAN for applications ranging from multicast to convergecast to network management to data caching. Some of these applications may demonstrate deficiencies in the existing system.

Over the years, much research has been performed about improving performance at end nodes by providing extensible network layers[33][10]. Some of these have demonstrated substantial performance gains over existing systems. It may be possible to achieve similar performance gains by using active network protocols.

Active networks should also be useful for implementing easily customizable replacements for existing protocols. For example, it should be possible to implement a reliable communications protocol similar to TCP using PAN. Applications could then modify or extend this protocol to suit their needs. It might be interesting to see what sorts of performance gains could be obtained by integrating HTTP with a reliable protocol optimized for the characteristics of HTTP.

It may turn out that one approach to writing PAN protocols is to take fully-functional modular protocols and then to modify them to suit a particular application by removing, modifying, and replacing functionality. Rather than just writing large monolithic code objects in C, it may make sense to write code objects in a language such as Prolac[19] that is designed specifically for the purpose of writing and extending protocols in a modular fashion.

Although active networks allow new protocols to be deployed dynamically, the current system is designed around tightly associating applications with a protocol or a small collection of protocols. However, many factors may contribute to the most appropriate protocol to use. It might be interesting to develop a system that would allow parts of protocols supplied by clients, servers, and intermediary nodes to be dynamically integrated within the network. This may be necessary in order to allow active networks to be used for applications such as enabling mobile networking and for making it easier for protocols to work over very high latency or very low bandwidth links (such as satellite or wireless links).

Chapter 7

Conclusions

The current PAN implementation only performs minimal resource management and does not yet fully address the safety, security, and interoperability issues. Bearing that in mind, the results of experiments performed on PAN indicate than an active network may be able to provide significant flexibility with only a small performance overhead over a traditional network node. The untuned kernel implementation of PAN is able to saturate a 100 Mbps Fast Ethernet with 1500 byte capsules while having an overhead of less than 15 percent when compared to the time it takes a traditional network node to process capsules.

Achieving high performance in an active networking node isn't at all hard and doesn't require fancy tricks. The implementation just needs to run within the kernel, not copy or touch capsule data whenever possible, and evaluate capsules using code objects that have been converted into native executable code at load time.

Active networking nodes should also be designed with the end-to-end argument in mind: applications should have as much control as possible over what happens to their data within the network. This means providing a simple but powerful interface to nodes that provides code objects with as much flexibility as possible. In theory, this should eventually be able to improve end-to-end performance by allowing protocols to do exactly what they need to do within the network and no more.

There are still quite a few significant challenges facing active networks, however. Interoperability must be provided, possibly through a bytecode system. Resource

management problems must be solved in a way that doesn't overly constrain the flexibility of the system. Safety and security guarantees must be provided, both at the node and network-wide levels. Eventually, a standard will need to be implemented and widely deployed. Possibly the greatest challenge will come at this point: preventing the very creeping featurism in active networking standards that active networks are themselves supposed to avoid.

If these challenges can be addressed, and if processing power scales along with network capacity demands, there doesn't seem to be an immediate reason why active networks shouldn't be a practical solution for making the future network infrastructure far more flexible.

Bibliography

[1] Ali-Reza Adl-Tabatabai, Geoff Langdale, Steven Lucco, and Robert Wahbe. Efficient and Language-Independent Mobile Programs. In *PLDI '96*, pages 127–136, Philadephia, Pennsylvania, May 1996.

[2] D. Scott Alexander, Marianne Shaw, Scott M. Nettles, and Jonathan M. Smith. Active Bridging. In *Proceedings of the ACM SIGCOMM'97 Conference on Communication Architectures, Protocols, and Applications*, Cannes, France, September 1997.

[3] Michael Beck, Harald Böhme, Mirko Dziadzka, Ulrich Kunitz, Robert Magnus, and Dirk Verworner. *Linux Kernel Internals.* Addison-Wesley, Harlow, England, second edition, 1998.

[4] Brian N. Bershad, Stefan Savage, Przemysław Pardyak, Emin Gün Sirer, Marc Fiuczynski, David Becker, Susan Eggers, and Craig Chambers. Extensibility, Safety and Performance in the SPIN Operating System. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles (SOSP)*, pages 267–284, Copper Mountain Resort, Colorado, December 1995.

[5] Samrat Bhattacharjee, Kenneth L. Calvert, and Ellen W. Zegura. Implementation of an Active Networking Architecture. Networking and Telecommunications Group, College of Computing, Georgia Tech, White paper presented at Gigabit Switch Technology Workshop, Washington University, St. Louis, July 1996.

[6] William Clinger, Jonathan Reese, et al. Revised^4 Report on the Algorithmic Language Scheme. *ACM Lisp Pointers IV*, 4(3):1–55, July–September 1991.

[7] Drew Dean, Edward W. Felten, and Dan S. Wallach. Java Security: From HotJava to Netscape and Beyond. In *Proceedings of the 1996 IEEE Symposium on Security and Privacy*, pages 190–200, Oakland, California, May 1996.

[8] Giovanna DiMarzo, Murhimanya Muhugusa, Christian Tschudin, and Jürgen Harms. The Messenger Paradigm and its Implications on Distributed Systems. In *Proceedings of the ICC'95 Workshop on Intelligent Computer Communication*, pages 79–94, June 1995.

[9] Dawson R. Engler. VCODE: A Retargetable, Extensible, Very Fast Dynamic Code Generation System. In *Proceedings of the 23rd Annual ACM Conference on Programming Language Design and Implementation*, pages 160–170, Philadephia, Pennsylvania, May 1996.

[10] Dawson R. Engler and M. Frans Kaashoek. DPF: Fast, Flexible Message Demultiplexing using Dynamic Code Generation. In *Proceedings of the ACM SIGCOMM'96 Conference on Communication Architectures, Protocols, and Applications*, pages 53–59, Stanford, California, August 1996.

[11] Dawson R. Engler, M. Frans Kaashoek, and James O'Toole, Jr. Exokernel: An Operating System Architecture for Application-Level Resource Management. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles (SOSP)*, pages 251–266, Copper Mountain Resort, Colorado, December 1995.

[12] FIPS 180-1. Secure Hash Standard. U.S. Department of Commerce/N.I.S.T., National Technical Information Service, Springfield, Virgnia, April 1995.

[13] Michael Franz. *Code-Generation On-the-Fly: A Key for Portable Software.* PhD thesis, Institute for Computer Systems, ETH Zurich, 1994.

[14] James Gosling and Henry McGilton. The Java Language Environment (White Paper). October 1995.

[15] John Hartman, Udi Manber, Larry Peterson, and Todd Proebsting. Liquid Software: A New Paradigm for Networked Systems. Technical Report TR 96-11, The University of Arizona Department of Computer Science, November 1996.

[16] Norman C. Hutchinson and Larry L. Peterson. The *x*-Kernel: An Architecture for Implementing Network Protocols. *IEEE Transactions on Software Engineering*, 17(1):64–67, January 1991.

[17] Alden W. Jackson and Craig Partridge. Smart Packets, A DARPA-Funded Research Project. Slides from 2nd Active Nets Workshop, March 1997.

[18] T. Kistler and M. Franz. A Tree-Based Alternative to Java Byte-Codes. In *Proceedings of the International Workshop on Security and Efficiency Aspects of Java '97*, Eilat, Israel, January 1997. Also in UC Irvine Technical Report No. 96-58.

[19] Eddie Kohler. Prolac: A Language for Protocol Compilation. Master's thesis, Massachusetts Institute of Technology, August 1997.

[20] Samuel J. Leffler, Marshall Kirk McKusick, Michael J. Karels, and John S. Quarterman. *The Design and Implementation of the 4.3BSD UNIX Operating System.* Addison-Wesley Publishing Company, Reading, Massachusetts, 1989.

[21] Li-Wei H. Lehman. Active Reliable Multicast. Draft Paper, Telemedia Networks and Systems Group, MIT Laboratory for Computer Science, December 1996.

[22] George C. Necula and Peter Lee. Safe Kernel Extensions Without Run-Time Checking. In *Second Symposium on Operating Systems Design and Implementation (OSDI '96)*, pages 229–243, Seattle, Washington, October 1996.

[23] Ron Rivest. The MD5 Message-Digest Algorithm. Network Working Group Request for Comments, April 1992. Internet RFC 1321.

[24] Jerome H. Saltzer, David P. Reed, and David D. Clark. End-to-end Arguments in System Design. *ACM Transactions on Computer Systems*, 2(2):277–286, November 1984.

[25] J. M. Smith, D. J. Farber, C. A. Gunter, S. M. Nettles, D. C. Feldmeier, and W. D. Sincoskie. SwitchWare: Accelerating Network Evolution (White Paper). June 1996.

[26] W. Richard Stevens. *UNIX Network Programming.* P. T. R. Prentice Hall, Englewood Cliffs, New Jersey, 1990.

[27] W. Richard Stevens. *TCP/IP Illustrated, Volume 1.* Addison-Wesley Publishing Company, Reading, Massachusetts, 1994.

[28] D. L. Tennenhouse, S. J. Garland, L. Shrira, and M. F. Kaashoek. From Internet to ActiveNet. MIT Laboratory for Computer Science, Request for Comments, January 1996.

[29] David L. Tennenhouse, Jonathan M. Smith, W. David Sincoskie, David J. Wetherall, and Gary J. Minden. A Survey of Active Network Research. *IEEE Communications Magazine*, 35(1):80–86, January 1997.

[30] David L. Tennenhouse and David J. Wetherall. Towards an Active Network Architecture. *Computer Communication Review*, 26(2):5–18, April 1996.

[31] Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham. Efficient Software-Based Fault Isolation. In *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles (SOSP)*, pages 203–216, Asheville, North Carolina, December 1993.

[32] Larry Wall, Tom Christiansen, and Randal L. Schwartz. *Programming Perl.* O'Reilly & Associates, Inc., Sebastopol, California, second edition, 1996.

[33] Deborah A. Wallach, Dawson R. Engler, and M. Frans Kaashoek. ASHs: Application-Specific Handlers for High-Performance Messaging. In *Proceedings of the ACM SIGCOMM'96 Conference on Communication Architectures, Protocols, and Applications*, Stanford, California, August 1996.

[34] David Wetherall. Safety Mechanisms for Mobile Code. Area Exam Paper, MIT Laboratory for Computer Science, November 1995.

[35] David J. Wetherall, John V. Guttag, and David L. Tennenhouse. ANTS: A Toolkit for Building and Dynamically Deploying Network Protocols. In *IEEE OPENARCH'98*, San Francisco, California, April 1998.

[36] David J. Wetherall and David L. Tennenhouse. The ACTIVE IP Option. In *Proceedings of the 7th ACM SIGOPS European Workshop*, Connemara, Ireland, September 1996.

[37] Edward Wobber, Martín Abadi, Michael Burrows, and Butler Lampson. Authentication in the Taos Operating System. *ACM Transactions on Computer Systems*, 12(1):3–32, February 1994.

[38] Yechiam Yemini and Sushil da Silva. Towards Programmable Networks (White Paper). In *IFIP/IEEE International Workshop on Distributed Systems: Operations and Management*, L'Aquila, Italy, October 1996.

[39] Zander and Forchheimer. Softnet: An Approach to High-Level Packet Communication. In *ARRL 2nd Computer Networking Conference*, 1983.

THESIS PROCESSING SLIP

FIXED FIELD: ill. _____ name _____

index _____ biblio _____

▶COPIES: (Archives) Aero Dewey (Eng) Hum

Lindgren Music Rotch Science

TITLE VARIES: ▶☐ _____

_____

_____

NAME VARIES: ▶☐ _____

_____

IMPRINT: (COPYRIGHT) _____

▶COLLATION: 89p _____

_____

▶ADD. DEGREE: _____ ▶DEPT.: _____

SUPERVISORS: _____

_____

_____ _____

_____ _____

_____ _____

_____ _____

_____ _____

_____ _____

_____ _____

NOTES:

cat'r: date:

page: F 40

▶DEPT: E.E.

▶YEAR: 1993 ▶DEGREE: M. Eng

▶NAME: NYGREN, Erik L.