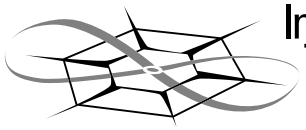
The University of Kansas



Information and
Telecommunication
Technology Center

Technical Report

Implementation and Performance of an Integrated OCaml-Based Active Node

Ravi Shankar, Gary Minden,
and Joseph Evans

ITTC-FY2003-TR-19740-07

February 2003

Defense Advanced Research Projects Agency and the
United States Air Force Research Laboratory,
contract no. F30602-99-2-0516

Abstract

The field of Active Networks has seen exciting developments over the past few years. The Active Node architecture describes an active node as a collection of a NodeOS, a runtime Environment, an execution environment (EE) and services deployed therein. This thesis describes the integration of University of Utah's OSKit (NodeOS), OCaml (Runtime) and the University of Pennsylvania's PLAN (EE). It is now time to pull together these components into an integrated system and establish its performance and behavior. In this thesis, we present the integration of the OCaml language system with OSKit. We use this foundation to build the PLAN execution environment on OSKit and examine its performance.

Table of Contents

Table of Figures

List of Tables

Chapter 1

Introduction

The rapid growth in the popularity of the Internet has brought about a need for faster and newer services. The traditional network infrastructure is unsuitable for fast deployment of services. Services in the existing network are rigidly built into the embedded software and hence are subject to a very slow standardization process. Efforts such as RSVP and IPv6 stand testimony to this statement. It is out of this need that a new paradigm known as Active networks takes shape. It involves the ability to program a network node dynamically. Such a network, known as Active network, would be able to better accommodate the need for faster services. Hence it will also become possible to deploy application-specific protocols, which the application will then be able to take advantage of.

The packets carried in an Active network are different from those of a traditional network. These special packets, known variously as SmartPackets, capsules etc., contain additional information in the form of program code. The nodes of an Active Network (called Active Nodes) are programmable in the sense that when a SmartPacket reaches an Active Node, the code inside the SmartPacket is extracted and executed.

The functionality of an active network node is divided among the Node Operating System (NodeOS) and the Execution Environments (EEs). Several active network prototypes, known as Execution Environments, exist. They include ANTS [16], PLAN

[5] and Magician [17]. Each EE exports an API that can be leveraged by Active Applications (AAs) to program the active node. It is possible for multiple EEs to exist in a single active node. These EEs offer services such as dynamic loading that enable SmartPackets to be loaded into the node. Thus new services can be deployed on the fly. Each of these differ in their details, but they all define the interfaces required by an active network that runs at user-level on general purpose operating systems. The EE has an interface to the NodeOS by means of which the active node can be modified according to the needs of the AA.

The node operating system of an Active Network deals with how packets are processed and how local resources are managed. It provides adequate support for services deployed and controls the resources consumed by the various components of an Active Node. It has the capability to terminate an EE or an AA if it believes that certain resource limits such as CPU usage or bandwidth consumed have been exceeded. The NodeOS also encompasses the Security Enforcement engine that authenticates and verifies the credentials of the entity that loads code onto the active node.

The NodeOS exports an interface to the EEs that is described in a companion document [7]. The NodeOS Interface defines the minimal fixed point for the Active Network architecture. Knowledgeable EEs should be able to exploit the advanced functionality provided by this interface although it should be possible for EEs to exist on just the minimal services.

1.1 Motivation

In the past few years, a number of Execution Environments have been developed which have explored the concept of Active Networks. Most of these EEs are written in Java™, which seems to be a popular language of implementation. However, to ensure that an active network execution environment is able to maintain state of correctness it is necessary to ensure that a given code does not change the system integrity of an active node. The PLAN EE developed by the University of Pennsylvania, as part of the SwitchWare [11] project is a step in this direction. It provides a type safe environment that ensures that all programs will terminate. The OSKit is a toolkit provided by the University of Utah to help build custom kernels that can be linked against the components provided. It can be used to custom build language kernels that can be directly booted over the hardware. The OSKit provides us with a simple OS structure that being highly modular and separable is easy to understand. It will be a part of our effort to build the PLAN EE over OSKit. Preliminary integration of the PLAN EE with the Moab NodeOS [6] will also be explored.

1.2 Proposed Study

We propose to integrate the PLAN EE from the University of Pennsylvania [5] with Moab/OSKit developed at the University of Utah [6]. This represents necessary work in the direction of building a fully integrated active node. The PLAN EE is written in OCaml, which is an object oriented implementation of the Caml dialect of ML. The OSKit is a set of modular libraries that can be used to build language kernels in a single

address space directly over hardware. The Moab NodeOS (from the University of Utah) is a resource control mechanism that provides the NodeOS interface API to EEs. Bringing these components together will help us study the performance and behavior of such an integrated node. This, we believe, is the way forward to build a large active network testbed.

1.3 Organization of the thesis

Chapter 2 discusses related work in the area of development of integrated active nodes. This is followed by Chapter 3, which explains the various components of our integrated node, namely the PLAN EE, OSKit kernel and the Moab NodeOS. Chapter 4 delves into the integration of these components and issues involved therein. The implementation of the PLAN active router is discussed here. The tests performed and the results obtained are discussed in Chapter 5. Finally, in Chapter 6 conclusions have been drawn based on the test results and possible future work has been suggested.

Chapter 2

Related Work

2.1 RCANE: Resource Controlled Active Network Architecture

In an active network, hostile or careless code could potentially consume all available resources at a node. The task of allocating resources to each flow according to a QoS policy is complicated by lack of knowledge about the behavior of the user-supplied code. The RCANE design [20] provides for protection against such possibilities at two levels – it limits what the code can do and how much effect its activities can have on the node.

RCANE, developed at the University of Cambridge, UK, is a resource control framework developed to help active network nodes control the services deployed on these nodes. It supports an active network programming model over the Nemesis Operating system, providing robust control and accounting of system resources, including CPU and I/O scheduling, and garbage collection overhead. The first requirement is achieved by making use of a safe and restricted language like OCaml. Fine-grained scheduling and accounting help in controlling the resources consumed by a particular application.

The RCANE paradigm leverages software protection vis-à-vis hardware. It depends on a safe language to do much of the protection checking at compile time or load time. This allows for much lighter-weight barriers between two principals (entities executing code on the node).

A *Session* is the primary abstraction for resource control in the RCANE framework. It represents a principal with resources reserved on the node. This is similar to the flow/domain model of the NodeOS specification. Session creation requires the owner of the session to present his credentials, resources required and the code at the time of creation. Resources include CPU, network I/O and memory usage. The concept of a virtual processor (VP), threads and threadpools represent CPU usage. A VP represents a regular guaranteed allocation of CPU time, according to some scheduling policy. Current NodeOS implementations make use of threads and threadpools only. The network flows associated with RCANE are mapped directly to Nemesis I/O channels. Each channel is associated with a demultiplexing key and buffers. Packets are demultiplexed by the keys associated with these channels and then handed off to the receive handler.

Memory managed by RCANE falls into four categories: network buffers, thread stacks, dynamically loaded code and heap memory. Network buffers and thread stacks are accounted to the owning session in proportion to the memory consumed. Code modules kept in memory may be charged depending on the system specific policy. Different heaps are recommended for different sessions so that Garbage Collection (GC) work for one session does not affect the other.

2.2 Janos: A Java-oriented OS for Active Network Nodes

The major goal of Janos [8] is to provide comprehensive and precise resource control over the execution of untrusted Java bytecode in an active network. The Janos architecture and the corresponding DARPA active network node architecture are shown

in Figure 2.1. Moab forms the NodeOS layer and the EE layer is comprised of the JanosVM, the Janos Java NodeOS, and the ANTS with Resource Management (ANTSR). The Active applications (AAs) are hosted on this foundation. The ANTSR runtime relies on Java bindings to the NodeOS API, while the JanosVM relies on the C implementation and provides the Java bindings.

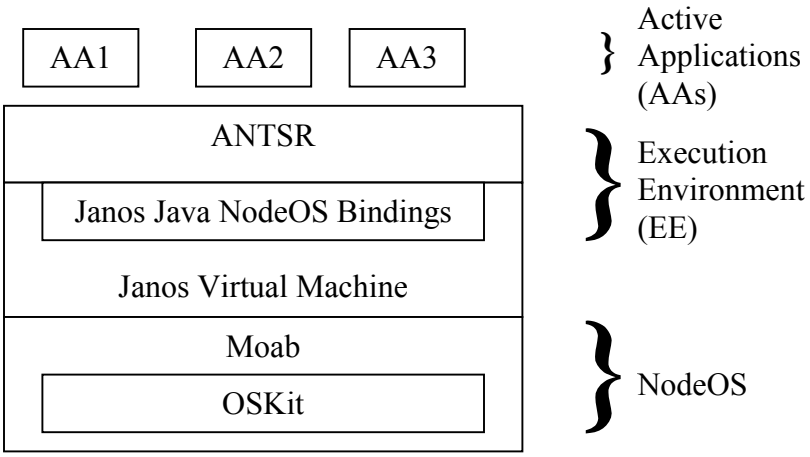


Figure 2.1 Janos Architecture

Janos leverages the type-safety of Java to provide memory safety and allows safe, fine-grained resource sharing across the user/kernel boundary. It differs from previous Java operating systems in its customization for the active network domain. JanosVM is the virtual machine that accepts Java bytecodes and executes them on Moab. It provides access to underlying NodeOS API through simple Java wrapper classes. In terms of resource controls, the CPU and network controls offered by Moab remain unchanged. The major changes introduced are the per-domain memory control through the use of the garbage collection mechanism. It no longer supports the shared heaps of KaffeOS.

Instead it supports separate GC threads for each heap. This provides for strict separation of flows.

The ANTSR system, based on ANTS 1.1, provides the interfaces necessary to execute untrusted, potentially malicious Active Applications. The use of the NodeOS API with ANTS helps add many significant features to it. These include domain-specific threads, separate namespaces, improved accounting of code loading and a simple administrator's console.

2.3 Magician/Scout

One of the early experiments in the field of integrated nodes was the integration of the Magician EE [17] with the Scout OS [18] and Joust VM [19]. This built up an active network node from the ground up that runs stand alone on PC's.

Magician is a toolkit for creating a prototype active network, developed at the University of Kansas. Magician provides a platform on which active applications can run, along with tools and interfaces for creating new SmartPackets that deploy new services and protocols in the active network. It is implemented in Java™ and uses UDP/IP for transport and routing.

Developed at the University of Arizona, Scout is an operating system architecture that is designed specifically to accommodate the needs of communication-centric systems. Scout has a modular structure that is complemented by an abstraction called the *path*, which is essentially an extension of a network connection into the host operating

system. The modular structure enables the efficient building of systems that are tailored precisely to the requirements of a particular application. Scout employs paths as the primary means to communicate data through a system.

Joust, is a configuration of Scout that includes a module that implements the Java Virtual Machine (JVM) along with those modules that implement the underlying services upon which JVM depends, such as TCP, IP, NFS, etc.

The idea behind building such an active network kernel on a modular, communication-oriented operating system (OS) such as Scout was that - for network intensive applications such as active networks, it is often the entire data-path that forms the application. Knowing the specific purpose for which an active network kernel is intended, allows the building of such a specialized system wherein computation does occur, but is incidental to moving data efficiently throughout the system. Such tailoring of resources to the application's specific needs has a direct, positive consequence on its performance.

Chapter 3

Components of Integrated Active Node

3.1 PLAN System

The PLAN system is part of the SwitchWare [11] active networking project at the University of Pennsylvania. Its architecture contains two basic components. It defines a high level interoperable layer wherein lie the active packets based on a new language called PLAN. Below this layer, exists a layer that provides node-resident services. These node resident services can be written in a general purpose programming language such as Java or OCaml. A typical PLANet (PLAN Active network) node would look as in Figure 3.1.

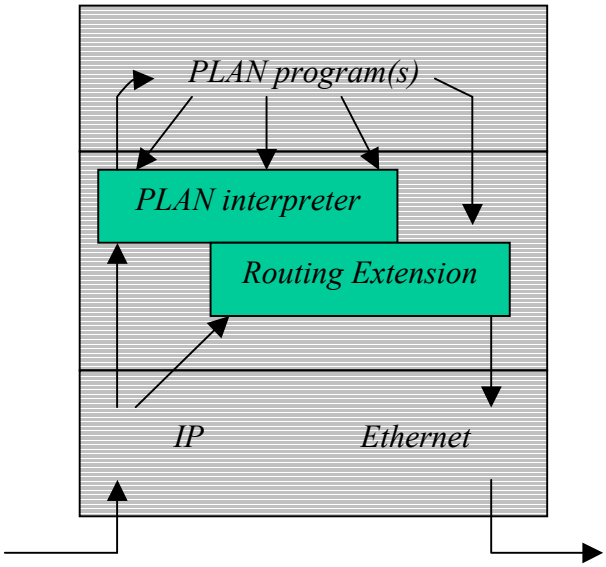


Figure 3.1 PLANet node architecture

The PLAN architecture [5] is designed to provide programmability at two levels. It can support both programmable (or active) packets and downloadable router extensions. This basic structure, as discussed below, follows a model of distributed computing that is based on remote evaluation. This sort of communication is asynchronous and unreliable. The extensible router infrastructure is used to provide support to heavyweight operations. These extensions can be dynamically installed, but are not mobile once installed.

3.1.1 The PLAN language

An active networking approach must tread a fine line among the following issues: *flexibility*, *safety and security*, *performance* and *usability*. Increased flexibility is the primary motivation for active networks. PLAN does not have to be too general because it adopts a two-level approach. This is because the service language helps in providing general-purpose expressibility. Hence PLAN has been able to express itself in 'little' programs and acts as glue between router resident services. By safety, we mean reducing the risk of mistakes or unintended behavior, and security encompasses the concept of privacy, integrity and availability in the face of malicious attack. To address some of these issues, PLAN was made a functional, stateless and strongly typed language. This ensures that PLAN programs are pointer-safe and concurrently executing programs do not interfere with each other. Since network operations involve changing the state of nodes in some way, some sort of authentication is required. However, packet-based authentication is very costly and hence PLAN pushes these features on to the node-resident services. PLAN programs are also statically typeable and are guaranteed to terminate as long as they use services which terminate. Basic error-handling facilities are also provided. These help in improving the usability of PLAN programs.

PLAN design is based on remote evaluation, rather than on remote procedure call. Specifically, child active packets may be spawned and asynchronously executed on

Field	Explanation
`chunk` code	Top-level functions and values
entry point	First function to execute
Bindings	Arguments for entry function
EvalDest	Node on which to evaluate
RB	Global resource bound
RoutFun	Routing function name
Source	Source node of initial packet
Handler	Function for error-handling

Figure 3.2 The PLAN packet format

remote nodes. Each packet may further create packets provided it adheres to a global resource bound. The PLAN packet format is shown in Figure 3.2

The primary component of each packet is its *chunk* (code *hunk*), which consists of code, a function to serve as an entry point, and values to serve as bindings for the arguments of the entry function. The `EvalDest` field specifies the active node on which this packet is to be evaluated. The packet is transported to the `EvalDest` active node by means of a routing function specified by the field `RoutFun`. The resource bound field specifies the total number of hops the packet and its subsequent child packets can take before evaluation. The source and handler fields represent the source node of initial packet and the handler function for error handling respectively. A host application

constructs the PLAN packet. It then injects it into the local PLAN router via a well-known port, say 3324. Remote execution is achieved by making calls to network primitives such as `OnRemote` or `OnNeighbor`. These are services written at the lower level in a general-purpose programming language. A PLAN program can be better explained by a simple *Ping* example as shown in Figure 3.3.

```
fun ping (src:host, dest:host) : unit =
  if (not thisHostIs(dest)) then
    OnRemote(|ping|(src,dest), dest, getRB(), defaultRoute)
  else
    OnRemote(|Ack| (), src, getRB(), defaultRoute)

fun ack() : unit = print("Success")
```

Figure 3.3 PLAN ping code

This program is placed in an active packet that executes '`ping`' at the source. The arguments for the ping function include the source and destination active nodes. The program then proceeds to execute as follows: If the packet is not at the destination, the `OnRemote` call is activated which creates a new packet and sends it over to the destination using the `defaultRoute`, which is RIP here. Once the packet is at the destination, it invokes another `OnRemote` that executes an '`ack`' function at the source. This completes the required operation for ping.

The PLAN language is characterized by the semantic basis provided by the theory of lambda calculus. However, in order not to compromise on security, PLAN does not include many features common to functional languages. In keeping with this idea, PLAN

has simple programming constructs: statement sequencing, conditional execution, iteration over lists with *fold* and exceptions. The lack of recursion and unbounded iteration (as well as the monotonically decreasing resource bound) ensure that all PLAN programs terminate. Its type system is strong and static but is dynamically checked. This arises from the necessity of distributed programming wherein static checking is necessary for debugging purposes whereas dynamic checking ensures safety of the code. It also helps that PLAN does not provide user-defined mutable state, although some aspects of PLAN, such as the resource bound, are stateful. In addition to the general exception based error handling mechanism, PLAN also provides an `abort` service that allows the program to execute a *chunk* on the source node. A major feature of PLAN is that chunks can be encapsulated in one another providing for protocol layering within PLAN.

Another issue is the choice of implementation language for PLAN at the service level. Such a language must to able to provide services to make code dynamically loadable. To enable such modules to work on heterogeneous types of machines, it must be portable. Thirdly, in order to provide guarantees for safe execution and termination it must be a safe language. PLAN has been implemented in OCaml [3] and the Pizza extension [21] to Java. Our efforts at an integrated node revolve around the use of OCaml as a safe language. Hence we will be discussing it alone.

3.1.2 OCaml

OCaml provides several of the design goals required for a service level language. Some of these have been outlined above. It has been developed at INRIA, Rocquencourt within the "Cristal project" group. The OCaml language system is an object-oriented implementation of the Caml dialect of ML.

All programming in OCaml is dominated by the use of functions. These first-class functions can be passed to other functions, received as arguments or returned as results. A powerful type system is another inherent feature of OCaml. It comes along with parametric polymorphism and type inference. Functions also may have polymorphic types. It is possible to define a type of collections parameterized by the type of elements, and functions operating over such collections. For instance, the sorting procedure for arrays is defined for any array, regardless of the type of its elements.

OCaml provides several common data-types such as int, float, char and string. Other data-types that include records and variants are standard features of functional languages. OCaml's type system is extensible by using user-defined data-types. New recursive data-types can be defined as a combination of records and variants. More importantly, functions over such structures can be defined by pattern matching: a generalized case statement that allows the combination of multiple tests and multiple definitions of parts of the argument in a very compact way.

OCaml is a safe language. The compiler performs many sanity checks on programs before compilation. That's why many programming errors such as data type confusions, erroneous accesses into compound values cannot happen in Caml. The compiler carefully verifies all these points, so that data accesses can be delegated to the compiler code generator and thus ensures that data manipulated by programs may never be corrupted. The perfect integrity of data manipulated by programs is hence granted for free in Caml. These features are extremely important for an active network where mobile code generated at one end gets evaluated on other machines.

Another feature of relevance to the active networking community would be the strong type safety provided by OCaml. OCaml is statically type-checked, hence there is no need to add type information in programs (as in Pascal, or C). Type annotations are fully automatic and handled by the compiler.

The OCaml compiler frees the programmer of all memory management. All memory allocations and de-allocations are handled automatically by the compiler. This way the programs are much safer, since spurious memory corruption never occurs. The memory manager works in parallel with an application, thereby improving the efficiency of execution of OCaml bytecodes.

In addition to these features, OCaml provides an expressive class-based object oriented layer that includes traditional imperative operations on objects and classes,

multiple inheritance, binary methods, and functional updates. Error handling is achieved by an exception mechanism as in other object-oriented languages.

The OCaml distribution comes with general-purpose libraries that facilitate arbitrary precision arithmetic, multi-threading, graphical user interfaces, etc. It also offers Unix-style programming environment including a replay debugger and a time profiler. Objective Caml programs can easily be interfaced with other languages, in particular with other C programs or libraries. This feature has been extensively used while building our wrapper around the Moab C libraries.

3.2 OSKit

3.2.1 Design and Rationale

The Flux research group at the University of Utah has developed OSKit. It provides a set of modularized libraries with straightforward and documented interfaces for the construction of operating system kernels, servers, and other core OS functionality. It is not an OS in *itself* and does not define any particular set of "core" functionality, but merely provides a suite of components from which real OS's can be built directly on hardware. The OSKit is considered self-sufficient in that it does not use or depend on existing libraries or header files installed on the host system.

Building the OSKit is no different than any other user-level system. Installing the OSKit causes a set of libraries to be created in a user-defined location. They can then be linked into operating systems just like ordinary libraries are linked into user-level applications. The most important goal of the OSKit is to be as convenient as possible for

the developer to use. This has led to its modular structure. It is also highly separable in that inter-module dependencies are very thin and managed through "glue" layers that provide a level of indirection between a component and the services it requires. The structure of the OSKit is shown in Figure 3.4.

For usability, it is critical that OSKit components have clean, well–defined interfaces. To provide this sort of abstraction, the Flux project adopted a subset of the Component Object Model (COM) [22] as a framework in which to define the OSKit's component interfaces. COM is a language-independent protocol that allows software components within an address space to rendezvous and interact with one another efficiently, while retaining sufficient separation so that they can be developed and evolved independently. The obvious advantage of COM is that it makes the OSKit interfaces consistent with one another. The other major technical advantages it brings about are implementation hiding and interface extension. COM helps us to define interfaces independently of their implementation. Hence, several different implementations of the same interface can exist together. COM also allows for interface extension and evolution. An object can export any number of COM interfaces, each of which can be defined independently by anyone with no chance of accidental collisions. Given a pointer to any COM interface, the object can be dynamically queried for pointers to its other interfaces. This mechanism allows objects to implement new or extended versions of existing interfaces safely. One of the other abstraction features of COM that is made use of in OSKit is that interfaces can be completely "standalone" and do not require any common infrastructure or support code that the client OS must use in order to make

use of the interfaces. This is markedly different from traditional operating systems like BSD or Linux. Consider the networking stacks of these systems. Though they are highly modular, each of their interfaces depend on a particular buffer management abstraction which are `mbufs` and `skbufs` respectively. The OSKit's corresponding interfaces, on the other hand, rely on no particular common implementation infrastructure.

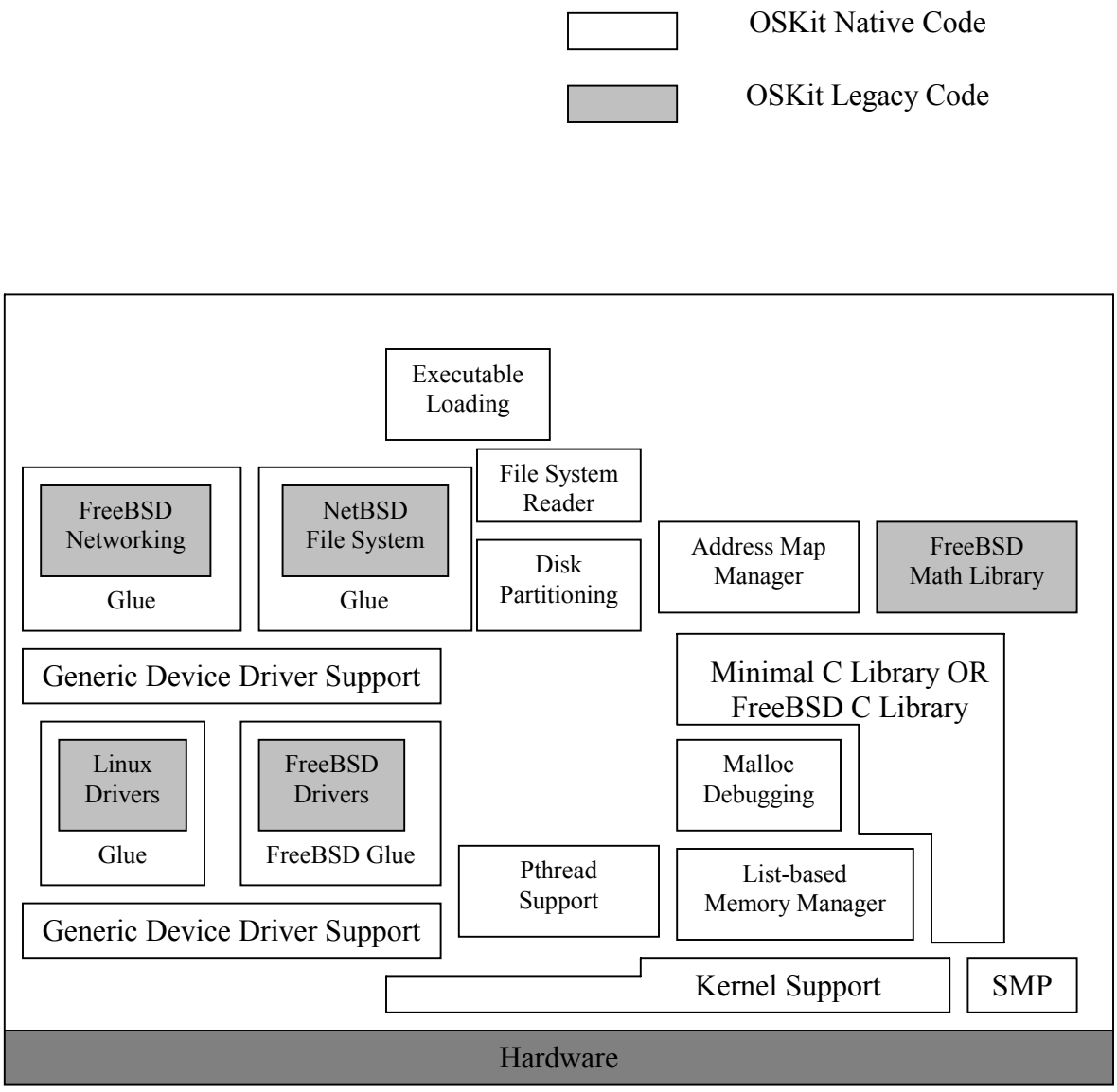


Figure 3.4 Sample OSKit Configuration

3.2.2 The OSKit Structure

Much of the OSKit code is derived directly or indirectly from existing systems such as BSD, Linux, and Mach. The OSKit uses a two pronged approach towards legacy code. For small pieces of code, that aren't expected to change much in the original source base, it is simply absorbed into the OSKit source tree, modifying it as necessary. The OSKit uses an encapsulation method towards large blocks of code. These include code borrowed from existing systems such as device drivers, file systems, and networking protocol stacks. The OSKit defines a set of COM interfaces by which the client OS invokes OSKit services. The OSKit components implement these services in a thin layer of *glue* code, which in turn relies on a much larger mass of encapsulated code, imported directly from the donor OS largely or entirely unmodified. The glue code translates calls on the public OSKit interfaces such as the `bufio` interface into calls to the imported code's internal interfaces. It also in turn translates calls made by the imported code for low-level services such as memory allocation and interrupt management into calls to the OSKit's equivalent public interfaces. Although tricky to implement, this simplifies the task of making modifications to each block of code. Modifications to blocks of code remain insulated from other components. For example, the OSKit's Linux device driver set has already tracked the evolution of the Linux kernel though several versions, starting with Linux 1.3.68. The encapsulation technique described above has made this relatively straightforward.

3.2.3. Overview of OSKit components

3.2.3.1 Bootstrapping

Most operating systems come with their own boot loading mechanisms, which are largely incompatible with those used by other systems. This diversity is attributed to the fact that not much time is spent on designing boot loaders which are relatively uninteresting when compared to actual OSs themselves. The OSKit, on the other hand, subscribes to the Multiboot standard [23]. The Multiboot standard provides a simple but general interface between OS boot loaders and OS kernels. Hence any compliant boot loader will be able to boot any compliant OS. Using the OSKit, it is very easy to create kernels that operate with a variety of existing boot loaders that support the Multiboot standard. Another key feature of the Multiboot standard is the ability of the boot loader to load additional files in the form of boot modules. The boot loader does not interpret the module in any way at the time of loading the kernel. It however provides the kernel with a list of physical addresses and sizes of all the boot modules. It is upto the kernel to interpret these modules in the way it deems fit.

3.2.3.2 Kernel Support Library

The primary purpose of the OSKit's kernel support library is to provide easy access to the raw hardware facilities without obscuring the underlying abstractions. Most of the definitions and symbols defined here are highly specific to supervisor-mode code. This is in contrast to the most other OSKit libraries that are specific to user-mode code. Another difference worth to note is that that most of the code here is architecture specific. No attempt has been made to hide machine-specific details so that the client OS may directly manipulate these. Other OSKit libraries build upon these machine-specific details and

provide higher architecture-neutral interfaces to higher layers. However these machine-specific details remain directly accessible.

The default behavior of the kernel support library is to do everything necessary to get the processor into a convenient execution environment in which interrupts, traps and other debugging facilities work as expected. The library also locates all associated modules loaded with the kernel and reserves the physical memory in which they are located. The client OS need only provide a main function in the standard C style. After everything is setup, this library calls the client OS with any arguments passed by the boot loader.

3.2.3.2 Memory Management Library

Memory management implementation typically used in user space, such as the `malloc` implementation in a standard C library, is not suitable for kernels because of the special requirements of the hardware on which they run. For example, device drivers need to allocate memory with specific alignment properties and space constraints. To address these issues, the OSKit includes a pair of simple but flexible memory management libraries. The *list-based memory manager*, or LMM, provides powerful and efficient primitives for managing allocation of either physical or virtual memory, in kernel or user-level code, and includes support for managing multiple "types" of memory in a pool, and for allocation with various type, size and alignment constraints. The *address map manager*, or AMM, is designed to manage address spaces that don't necessarily map directly to physical or virtual memory. It provides support for other aspects of OS implementation such as the management of process address spaces, paging partitions or

free block maps. Although these libraries can be used in user space, they are specifically designed to satisfy the needs of OS kernels.

3.2.3.4 C Libraries

The OSKit provides two different C libraries – one a minimal C library native to OSKit and another imported from the FreeBSD C library. The OSKit's minimal C library is designed around the principle of minimizing dependencies rather than maximizing functionality and performance. For example the standard I/O calls do not do any buffering, instead relying directly on underlying `read` and `write` operations. Dependencies between C library functions are minimized. This approach is followed since the standard C library running on a full-function OS, such as Linux, makes too many assumptions to be reliable in a kernel environment.

The FreeBSD C library provides an alternative to the OSKit's minimal C library so that sophisticated applications can be built using it. In addition to the standard single threaded version of the library, a multi-threaded version is also built which relies on the pthread library to provide the necessary locking primitives. Like the minimal C library, the FreeBSD C library depends on the POSIX library to provide mappings to the appropriate COM interfaces. For example, `fopen` in the C library will chain to open in the POSIX library, which in turn will chain to the appropriate `oskit_dir` and `oskit_file` COM operations.

3.2.3.5 Debugging support

The OSKit provides the developer with a full source-level kernel-debugging environment. The OSKit's kernel support library includes a serial-line stub for the GNU debugger, GDB. The stub is a small module that handles traps in the client OS environment and communicates over a serial line with GDB running on another machine, using GDB's standard remote debugging protocol.

3.2.3.6 Device Driver Support

One of the most expensive tasks in OS development and maintenance is supporting the wide variety of available I/O hardware. The OSKit avoids direct maintenance by leveraging the extensive set of stable, well-tested drivers developed for existing kernels such as Linux and BSD. The OSKit uses the technique of encapsulation discussed earlier to integrate these various code bases into the OSKit. Currently, most of the Ethernet, SCSI and IDE disk device drivers from Linux 2.2.14 are included. Eight character device drivers that manage the standard PC console and the serial port are imported from FreeBSD in the same way.

3.2.3.7 Protocol Stacks

The OSKit provides a full TCP/IP network protocol stack. It incorporates the networking code by encapsulation. The TCP/IP stack is borrowed from the FreeBSD system that is generally considered to have a more mature network protocol implementation. This demonstrates another advantage of using encapsulation. Two different systems, namely Linux device drivers and FreeBSD TCP/IP can coexist with one another. With this

approach, it is possible to pick the best components from different sources and use them together.

However this approach is also fraught sometimes with inefficiency. The networking stack is an excellent example of this. When a packet arrives at an OSKit node, it is initially picked by the Linux device drivers and represented as the Linux packet buffer, `skbuff`. The OSKit represents all packets as COM `bufio` objects. Due to contiguous nature of Linux packet buffers, they can be directly passed to the FreeBSD TCP/IP stack as `bufio` objects. The FreeBSD code internally repackages them as `mbufs`, which is the FreeBSD abstraction for packet buffers. However, the situation reverses in the case of outgoing packets. `mbuf`s consist of multiple discontiguous buffers chained together. Hence when they are passed to the Linux driver code as `bufio` objects, the Linux code has to resolve these into contiguous buffers. This mismatch sometimes requires extra copying between these two modules on the send path.

3.2.3.8 File Systems

The OSKit incorporates standard disk-based file system code, again using encapsulation, this time based on NetBSD's file systems. The choice of NetBSD was influenced by the fact that it had one of the best-separated interfaces of the available systems. FreeBSD and Linux file systems are more tightly coupled with their virtual memory systems.

3.3 Moab NodeOS

3.3.1 The NodeOS Interface

The NodeOS interface defines the boundary between the EE and the NodeOS. Generally speaking, the NodeOS is responsible for distributing the node's resources among the various packet flows, while the EE's role is to offer AAs a sufficiently high-level programming environment. The design of the NodeOS is influenced by three major considerations:

1. The interface's primary role is to support packet forwarding. Hence the interface is designed around network packet flows: packet processing, accounting for resource usage, and admission control are all done on a per-flow basis. No single definition is attributed to the flow.

2. All NodeOS implementations need not export the same set of interfaces. Some NodeOS implementations can have advanced features such as hardware transfer of non-active IP packets. However, these features must be exported to EEs so that they may make use of it. The NodeOS may also be extensible. Exactly how a particular OS is extended is an OS-specific issue.

3. Whenever the NodeOS requires a mechanism that is not particularly unique to active networks, the NodeOS interface borrows from established interfaces, such as POSIX.

3.3.1.1 NodeOS Abstractions

The NodeOS defines five primary abstractions: thread pools, memory pools, channels, files and domains. The first four encapsulate a system's four types of resources:

computation, memory, communication, and persistent storage. The domain abstraction encapsulates all the above and is used to aggregate control and scheduling of the other four abstractions.

Domains:

The domain is the primary abstraction for accounting, admission control, and scheduling in the system. A domain typically contains the following resources: a set of channels on which messages are received and sent, a memory pool, and a thread pool. Active packets arrive on an input channel (`inChan`), are processed by the EE using threads and memory allocated to the domain, and are then transmitted on an output channel. One can think of a domain as encapsulating resources used across both the NodeOS and an EE on behalf of a packet flow. Domain creation is hierarchical. This is used solely to constrain domain termination. A domain can be terminated by the domain itself, or by the NodeOS because of some policy violation. The domain hierarchy is independent of resource allocation. That is, each domain is allocated resources according to credentials presented to the NodeOS at domain creation. These resources are not deducted from the parent domain.

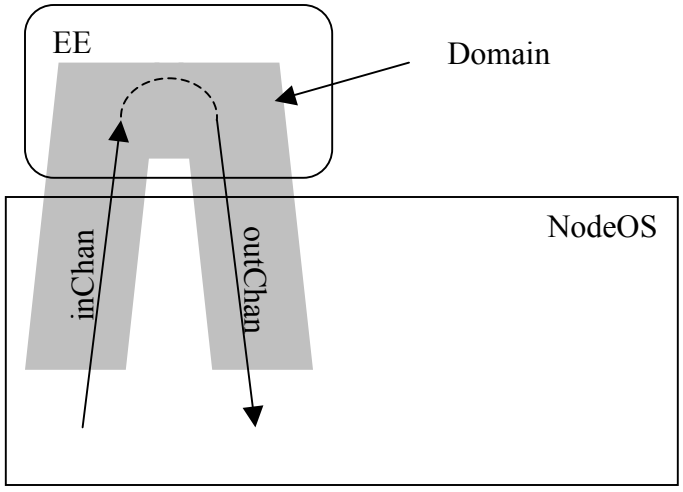


Figure 3.5 A typical NodeOS domain

Thread Pools:

Thread pools are the primary abstraction for computation and exist for accounting purposes. A thread pool is initialized at the time of domain creation and threads run "end-to-end", i.e., to forward a packet they typically execute input channel code, EE-specific code and output channel code. Threads in the pool are implicitly activated and scheduled to run in response to certain events like message arrival, timers firing and kernel exceptions. The entire domain is terminated if a thread misbehaves. There is no explicit operation of killing a thread.

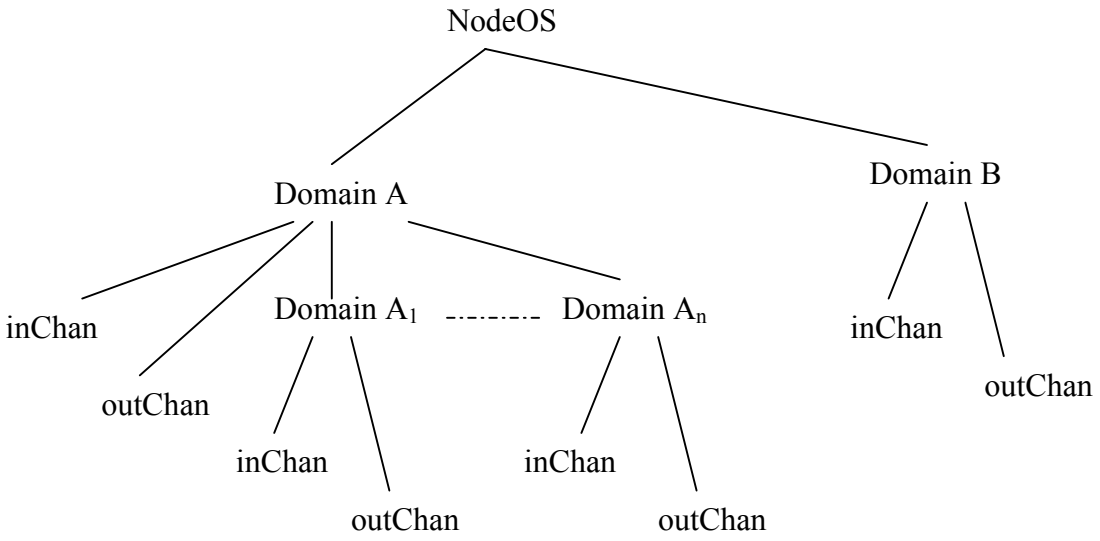


Figure 3.6 NodeOS Domain Hierarchy

Memory Pools:

The memory pools are the primary abstraction for memory in the NodeOS. It is used to implement packet buffers, the NodeOS abstraction for network flow buffers. A memory pool combines the memory associated with the domains. This many-to-one mapping between domains and memory pools accommodates EEs who would like to manage

memory resources themselves. Memory pools have an associated callback function that is invoked by the NodeOS whenever the resource limits of the pool have been exceeded. The corresponding domains are terminated by the NodeOS if the EEs do not handle resource violation in a timely manner. Memory pools can be arranged hierarchically though this is not used to control the propagation of resources.

Channels:

Domains create channels to send, receive and forward packets. Some channels are anchored in an EE i.e. they are used to send packets from an EE to the underlying physical layer and vice versa. They can hence be classified into two types-`inChan` and `outChan`. When creating an `inChan`, a domain must specify the following things: (1) which arriving packets are to be delivered on this channel; (2) a buffer pool that queues packets waiting to be processed by the channel; and (3) a function to handle the packets. Packets to be delivered are described by a protocol specification string, an address specification string, and a demultiplexing (demux) key. On the other hand the requirements for an `outChan` include (1) where the packets are to be delivered and (2) how much link bandwidth the channel is allowed to consume. Another type of channel known as `cutChan` is defined which forwards packets through the active node without being processed by an EE. This might correspond to a standard forwarding path that the NodeOS implements very efficiently. It can also be created by concatenating an existing `inChan` to an existing `outChan`.

A packet is demultiplexed by specifying the protocol and addressing information. For example, the protocol specification "if0/ip/udp" specifies incoming UDP packets tunneled through IP. The address specification defines destination-addressing information like the destination UDP port number. The NodeOS designers realized that simply specifying the protocol and addressing information is insufficient when an EE wants to demultiplex multiple packet flows out of a single protocol. Hence a `demux` key is used which is passed on to the inChan. It can specify a set of (`offset, length, value, mask`) 4-tuples. These tuples are compared in an obvious way to the "payload" of the protocol.

Files:

Files are provided to support persistent storage and sharing of data. The file system interface loosely follows the POSIX specification and is intended to provide a hierarchical namespace to EEs that wish to store data in files.

3.3.2 The Moab NodeOS

The Moab [6] is a C implementation of NodeOS developed at the University of Utah. It comprises of a multi-threaded fully-preemptible single address-space operating environment implementing the NodeOS abstractions. Moab is not an operating system in the strict sense of the word. This is because invocations of NodeOS functions are direct function calls and do not "trap" into the OS.

Moab is built using the OSKit. This helps it to leverage many of the components such as the device drivers, a networking stack, and a thread implementation, as well as a host of support code for booting and memory management. The following paragraphs describe the Moab implementation of the NodeOS API that help us to understand the advantages and disadvantages of using OSKit:

Threads:

The implementation of NodeOS threads directly leverages the POSIX thread library. This was possible because of the similarity between the NodeOS and POSIX APIs. This direct mapping between NodeOS and POSIX threads caused some performance problems. The NodeOS' thread-per-packet model of execution led to creation and destruction of pthreads, which imposed a lot of overhead. This was avoided by creating and maintaining a cache of active pthreads in every thread pool.

Memory:

The OSKit made it easy to track the memory allocated and freed within its components such as the networking stack. However, it was difficult identifying the correct user to charge the memory. Right now, the Moab charges the memory allocated to the "root flow". The alternative would be to charge it to the thread doing the allocation. The OSKit's memory interfaces are being modified to bring it inline with the NodeOS' needs.

Channels:

Channels provide the path necessary for execution of a packet flow. Anchored channels, namely `inChan` and `outChan`, are implemented in two ways depending on their

protocol specification. Raw interface ("`if`") channels are directly implemented over the stock Linux device drivers. All the other types of protocols use the OSKit's socket interface and its network stack to deliver UDP/TCP packets to Moab. This only partially implements the NodeOS API requirements since direct "IP" packet delivery is not supported. Cut-through channels are implemented as unoptimized concatenation of NodeOS `inChan`/`outChan` pairs and perform no additional protocol processing.

Chapter 4

Implementation of PLAN Router on OSKit

4.1 Motivation

The OSKit provides us with a platform wherein single address-space based kernels can be directly executed on bare hardware. It is with this idea that we proceed to build a PLAN router that operates directly over the OSKit. The OSKit provides all the essentials required for the router to operate. These include a C library, a TCP/IP protocol stack and a pthread library. The OSKit's modular design helps us in choosing the components we would like the router to be integrated with. This modularity also implies configurability. Hence, given a choice, we could link the PLAN router with a different set of components at system build time.

The PLAN router is built in a vertically integrated fashion over the OSKit in a single address space. In a traditional operating system, such as Linux, we tend to differentiate between the user space and the kernel space. This is done to operate the two levels with different levels of trust. This again is due to the fact that programs loaded and unloaded at runtime should be operated only with certain privileges. When a single application such as a PLAN router is intended to run on a system, it is no longer necessary to differentiate between processes. Hence, the distinction between kernel-space and application becomes fuzzy. In our implementation, we will study the effect of OSKit's components on our PLAN router implementation.

4.2 Implementation of PLAN router

4.2.1 Porting OCaml to OSKit

The first task involved in the implementation of a PLAN active node on OSKit was the porting of the OCaml language to OSKit. A normal install of OCaml on Linux would look like in Figure 4.1.

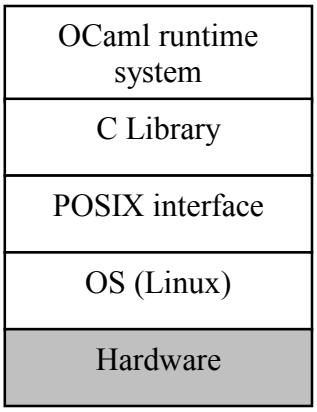


Figure 4.1 OCaml Installation on Linux

In order to integrate OCaml with OSKit, the interfaces required by OCaml from the underlying operating system have to be figured out. It is also necessary to examine the interfaces provided by the OSKit in order to match them with those required by OCaml. In this process, the dependencies of the libraries linked to provide the interfaces have to be taken care of. The timeline of development of a custom OCaml kernel over the OSKit is shown in Figure 4.2.

A regular OCaml system is installed over Linux, in order to build the various OCaml tools such as `ocamlc`[1], `ocamlopt`[2], etc. The OCaml-on-Linux compiler (`ocamlc` or `ocamlopt`) thus built is used to custom-compile OCaml sources to generate a C object file instead of compiled object bytecode executables. The objective Caml runtime system comprises three main parts: the bytecode interpreter, the memory manager, and a set of C functions that implement the primitive operations. In the default mode, the Caml linker produces bytecode executables for the standard runtime system, `ocamlrun`, with the standard set of primitives. In the "custom runtime" mode, we may generate a C object file that contains the list of C primitives required or an executable file that contains both the runtime system and the bytecode for the program.

The OCaml runtime system was built using OSKit libraries. We then use the C object file generated above and link it against a runtime system built using OSKit components. If we are using the native code compiler, the `-custom` flag is not necessary, as it can directly produce a C object file with the help of the `-output-obj` option. An OSKit interface file is then compiled which initializes the OCaml component. This is linked with OSKit libraries and the modified OCaml runtime system to form an executable. Using OSKit's image generating tools, we generate a multi-boot image of this kernel. This image is then booted as a kernel using a multi-boot loader such as GRUB (Grand Unified Boot-loader). The entire process has been described in Appendix B.

[1] OCaml bytecode compiler
[2] OCaml native code compiler

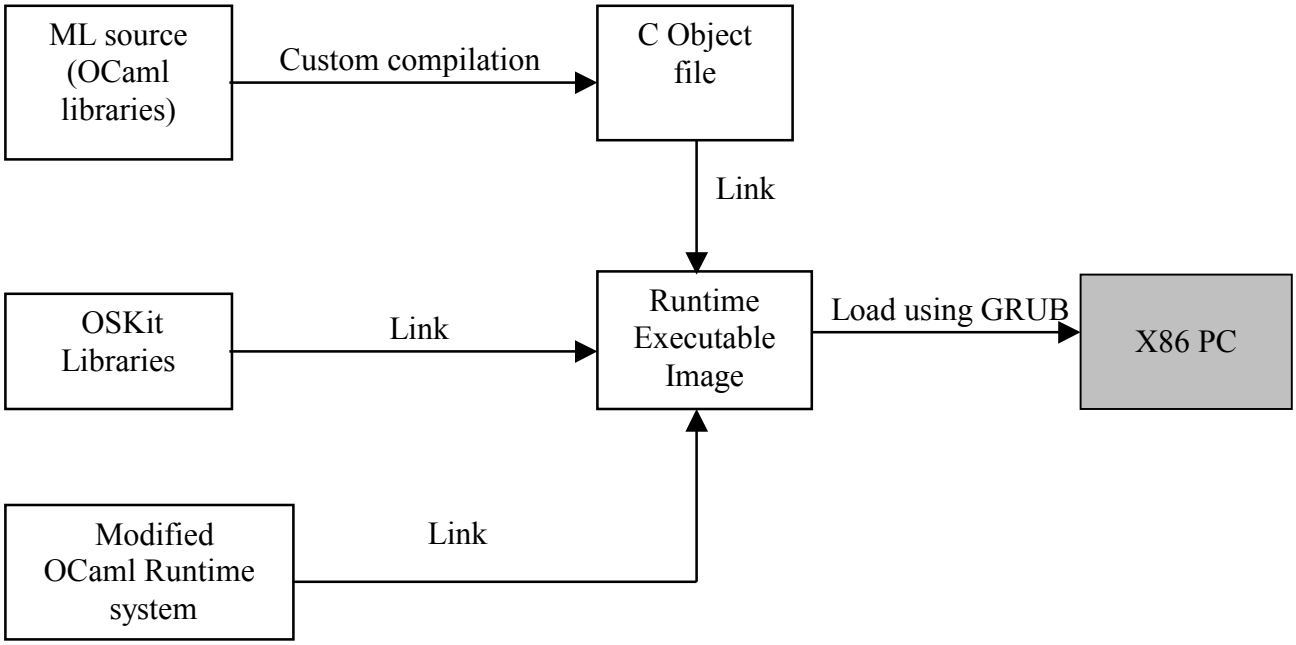


Figure 4.2 An OCaml based OSKit kernel

4.2.2 Integration of PLAN with OSKit

The integration of PLAN with OSKit involved matching all the OCaml interfaces used by PLAN with those provided by OSKit. Fixes were required in areas such as file operations and usage of loopback interfaces. To fix this, it was decided to convert all file-based operations to string operations. Most of these files were configuration files and DNS files. PLAN operates its own DNS system which makes use of a file similar to the standard '/etc/hosts' file on Unix systems. The PLAN code thus modified was then custom compiled to generate object code. This wass then linked against OSKit libraries to form an executable that in turn was used to generate a multiboot image. This installation is as shown in Figure 4.3.

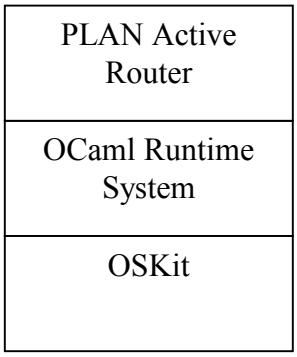


Figure 4.3 PLAN installation on OSKit

The PLAN router protocol graph as used on OSKit is shown in Figure 4.4.

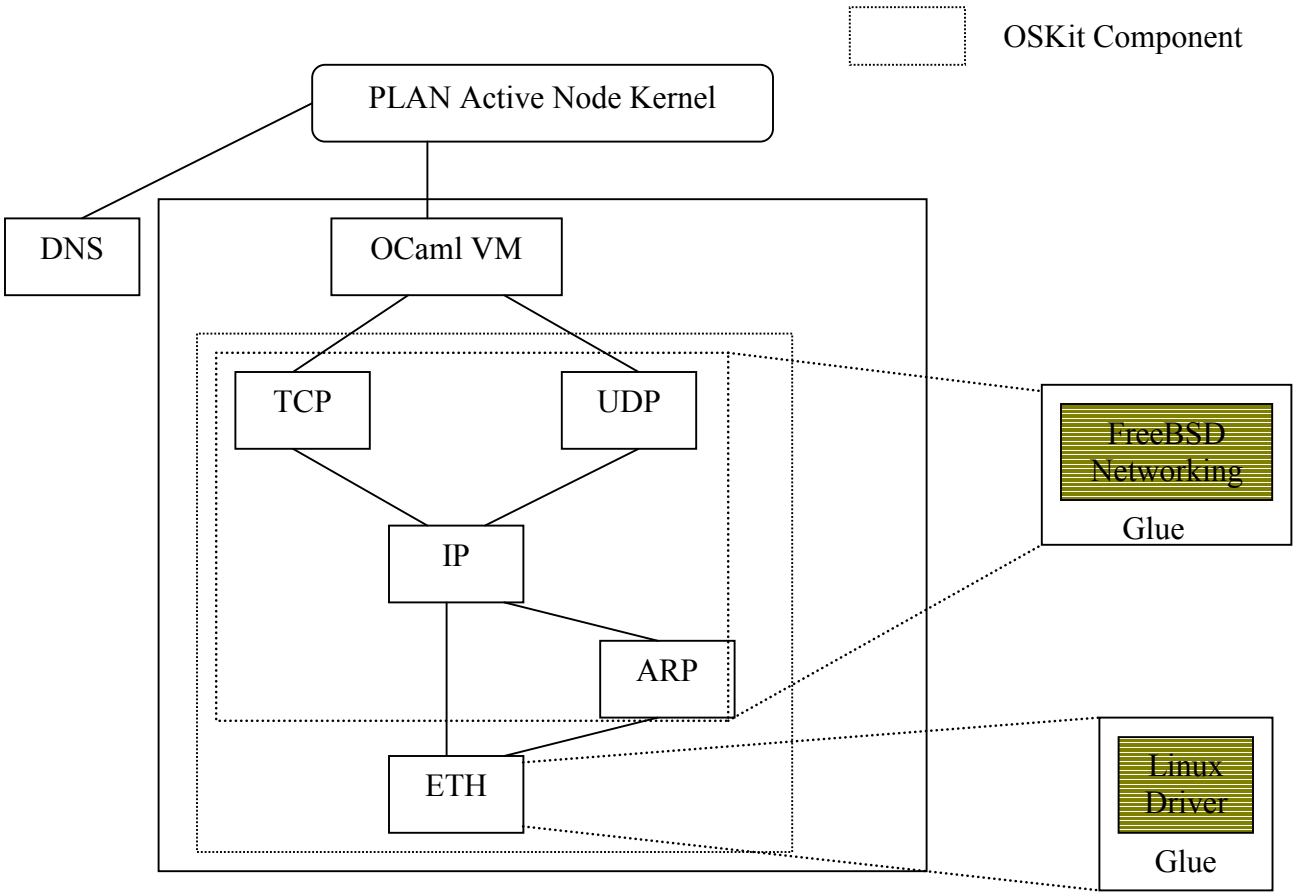


Figure 4.4 PLAN Active Node Protocol graph

The DNS shown in Figure 4.4 is implemented by PLAN in a user-level file that maps host names with their IP addresses. The OCaml VM is the OCaml runtime system linked against OSKit libraries that can load and interpret bytecode modules. The TCP, UDP and IP modules of OSKit are derived from the FreeBSD TCP/IP stack of OSKit. The Ethernet module uses the device drivers of Linux to probe and operate network cards. Such a system is not highly optimized for network operations. We shall further investigate this mismatch during the discussion of our results. Minor fixes were required to the Pthread library to ensure that it gets linked in properly.

4.2.3 Integration with the Moab NodeOS

As described earlier, the Moab NodeOS is a C implementation of the NodeOS API. The Moab environment is first setup by means of the interface `an_moab_setup`. This function initializes the Moab by first initializing the memory associated with the NodeOS flow. The various hardware devices are probed and filesystems initiated. The networking framework is initialized and can be configured manually. The Moab now reaches a state where its bare setup is done. The NodeOS root flow is now started in this context. The parameters passed to it include the root flow thread count and the root flow stack size. The function for flow initialization and it's associated arguments is also passed. The various NodeOS components such as resources, credential objects and threads are initialized. The setup of Moab is now complete.

With the initialization of Moab complete, the root flow is then fired off as a thread in this context. There is no callback associated with the root flow. It can be killed only by an explicit call to `ani_moab_shutdown`. New credentials are created in the root flow that may correspond to child flows. A new flow is now created in the context of root flow. Since we have only one flow subordinate to the root flow, all the root credentials are transferred to the new flow. The flow initialization function and the corresponding arguments are now passed to this flow. This flow is used to start the OCaml virtual machine. A resource specification is also passed to the flow that specifies the number of threads and the stacksize allocated for each thread. A thread is then fired off which starts up the Caml flow. This hierarchical structure is shown in Figure 4.5

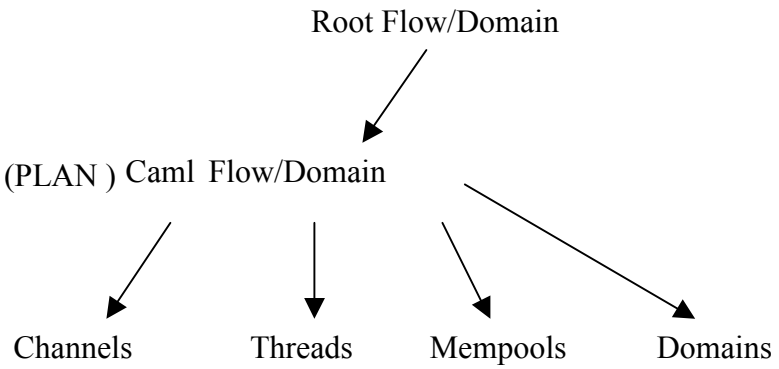


Figure 4.5 Hierarchical structure of NodeOS flows

Chapter 5

Testing and Discussion of Results

The following test framework was used to evaluate our implementation. Figure 5.1 describes a linear topology used to test the performance of OSKit as a node operating system for an OCaml based active node.

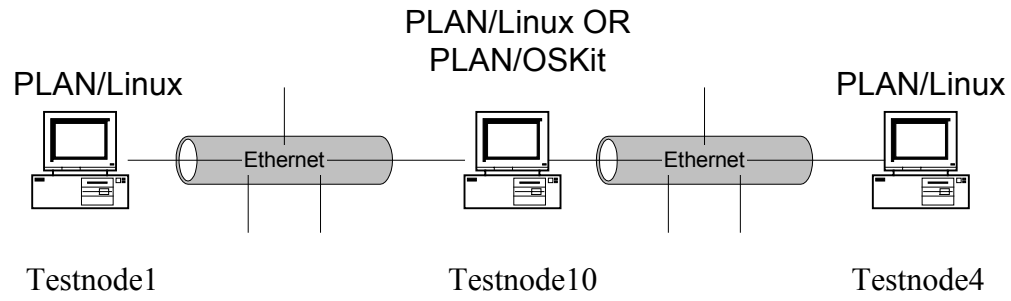


Figure 5.1 Test topology

The machines used in the above tests were equipped with 530 MHz Pentium III processors and 128 MB RAM. Their network interfaces were connected to 100 Mbps Ethernet segments. The Caml benchmarking tests were done on a single machine featuring both Linux and OSKit versions. Table 5.1 compares results obtained using Linux 2.2.13 and OSKit version 20010214. The tests have been performed both with the bytecode and native code versions of OCaml. All measurements are made using the `rdtsr` function provided for *i386* machines.

Most of the benchmarks are self-explanatory. We observe that the performance provided by OSKit is not better than that provided by Linux.

Benchmark Test	OCaml-bc[3]/ Linux (Time in ms)	OCaml-Nat[4]/ Linux (Time in ms)	OCaml-bc/ OSKit (Time in ms)	OCaml-Nat/ OSKit (Time in ms)
Array Access 1000000 times in a tight loop	876.25	215.992	851.441	199.035
Array Access (Test 2) 1000000 times after unrolling of loops	757.45	215.891	742.894	199.167
Fibonacci Series N = 32	1,159.94	140.16	1,151.54	149.053
Hash Access With 80000 entries	1,320.33	692.093	1,084.01	518.831
Heap Sort of 80000 randomly created entries	1,876.98	138.048	1,914.63	135.223
Various List operations for 16 lists each of size 10000	991.441	184.252	990.477	171.724
Matrix Multiplication of 2 matrices of size 30x30	5,645.93	198.179	5,824.22	197.17
1000000 Method Calls on the same object	1,429.35	129.09	1,388.04	129.192
Loop overhead for 16 nested loops	3,861.07	169.358	3,668.71	170.669
100000 thread synchronizations between 2 producer/consumer threads using a mutex and a condition variable	3,649.96	3,436.02	1640.25	1488.68
Generated 900000 Random numbers	1,085.74	164.336	1,089.89	164.168
Sieve of Eratosthenes Counts primes from 2 to 8192 , 300 times	3,202.76	145.933	3,170.74	155.087
String concatenates 40000 strings(using `Buffer.add_string`)	84.231	11.286	83.338	11.173
String Append (using the string concatenation operator)	44,959.89	43,684.61	41,592.80	41,274.74

Table 5.1 Comparison of OCaml benchmarks

[3] OCaml Bytecodes
[4] OCaml Native code

5.1 Baseline Performance

We study the performance of our implementation of the PLAN active network node on OSKit using the test framework in Figure 5.1 and compare it with PLAN running over Linux. The test framework is used as it is for the throughput measurements. However, for the latency measurements, a two-node setup is used. Specifically, we measure the latencies observed during the execution of the PLAN ping program.

The PLAN ping experiment will serve to demonstrate the difference in latencies between Linux and OSKit. It consists of a simple application program, which injects a PLAN packet that contains the ping code described earlier. This packet is injected at Testnode1. The packet is evaluated at its destination (Testnode10) which is across a single hop and sent back to the source.

5.2 Latency measurements

For the above PLAN ping experiment, the results are as shown in Figure 5.2. The PLAN ping times have been measured as the average round trip time taken by the packets sent from a Linux host to an OSKit router. Each individual test involved 100 round trip times. It illustrates the performance of this active protocol both on Linux and OSKit. To enable us to compare these figures with standard benchmarks, the ICMP and C-level ping times are also shown. The C level ping consisted of a simple UDP client which sends a packet to a UDP server that returns the packet back to the client. All the tests were done with minimum packet sizes. It was also noticed that there is a perceptible difference between

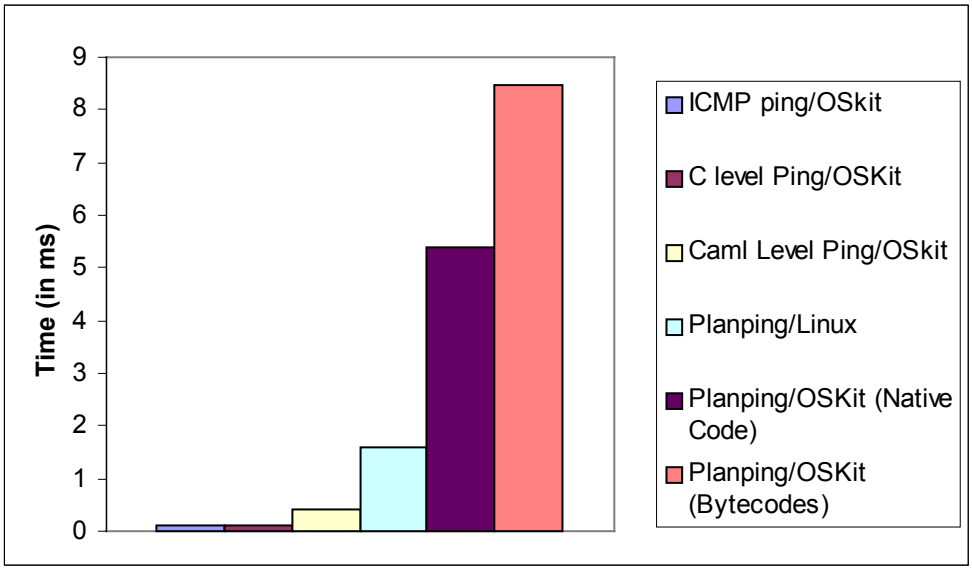


Figure 5.2 Latency Measurements

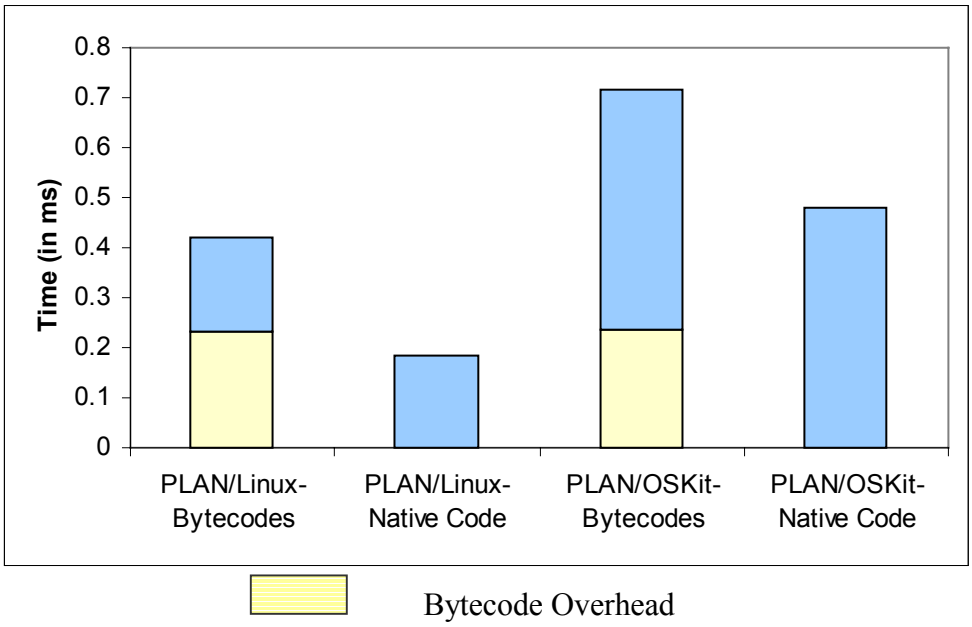


Figure 5.3 PLAN ping packet evaluation overhead

ping times of the bytecode and native versions of PLAN/OSKit. The Linux version does not show such a large difference between the two versions. The time taken for packet evaluation is shown in Figure 5.3. These times reflect the per-packet switching overhead. The packet concerned contains the PLAN ping packet code. Figure 5.3 measures the time taken by a PLAN Active node to unmarshall the packet, interpret the PLAN code and send it over to its next destination, which in our case, is the source. The processing overhead is considerably larger for PLAN/OSKit when compared to PLAN/Linux. However this accounts for only a small proportion of the total delay.

In order to further understand the delays seen, we ran an unoptimized version of PLAN/OSKit. This version forks an extra thread that waits for packets from the local PLAN port. It was noticed that the delays experienced increased considerably in this case. These results are tabulated in Table 5.2. This is probably due to the poor performance of the OSKit Pthread library.

Switch	Optimized PLAN	Unoptimized PLAN
PLAN/OSKit – Native Code	5.4 ms	14.3 ms
PLAN/OSKit – Bytecodes	8.45 ms	37.91 ms

Table 5.2 Comparison between Optimized and Unoptimized versions of PLAN

5.3 Throughput Tests

5.3.1 Application-level Exchange

The following test framework measures the forwarding capacity of a PLAN node. We make use of a linear back-to-back connectivity between 3 nodes in order to compare the forwarding performance of a PLAN switch. The test topology is shown in Figure 5.4.

In order to do these tests, PLAN routers are set up on the three nodes. We then make use of a Caml program that generates load of a given payload on the sender side. This program injects these packets into the local PLAN router through its PLAN port. The destination of these packets is on the far end of the topology, which is Testnode 4. These packets are initially evaluated at Testnode1 and routed through the active network by Testnode10. The PLAN router on Testnode 4 later receives them and hands over the packets to a receiver program running on Testnode 4 which waits for these packets on its PLAN port. A total of 80,000 packets were sent from Testnode1 through Testnode11 onto Testnode4.

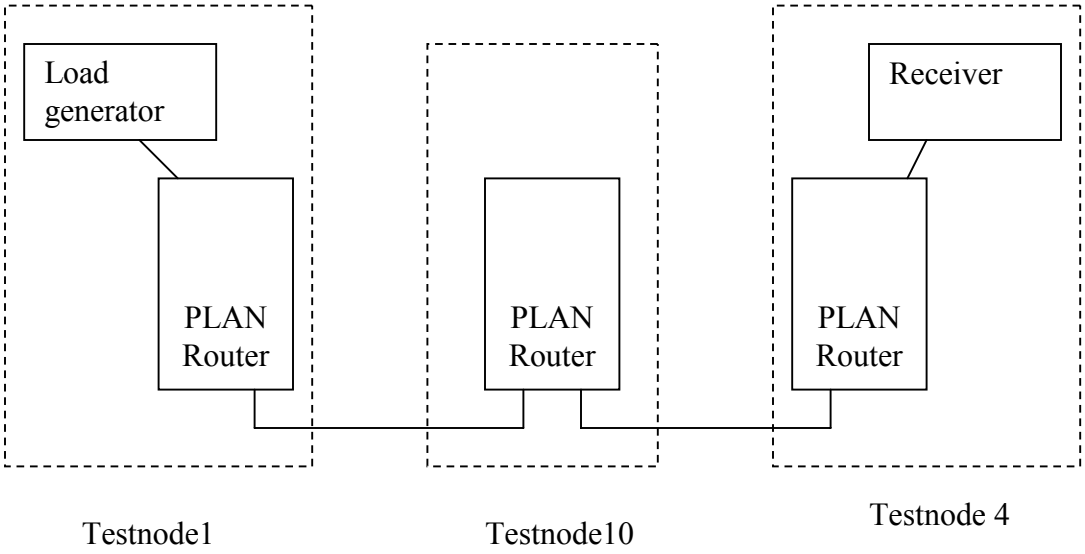


Figure 5.4 Application level bandwidth tests

The results of these tests are as shown in Figure 5.5. As seen from the Figure the PLAN/OSKit shows significant loss of packets as compared to the PLAN/Linux switch. This could be attributed to a less than tighter integration of PLAN over the OSKit framework.

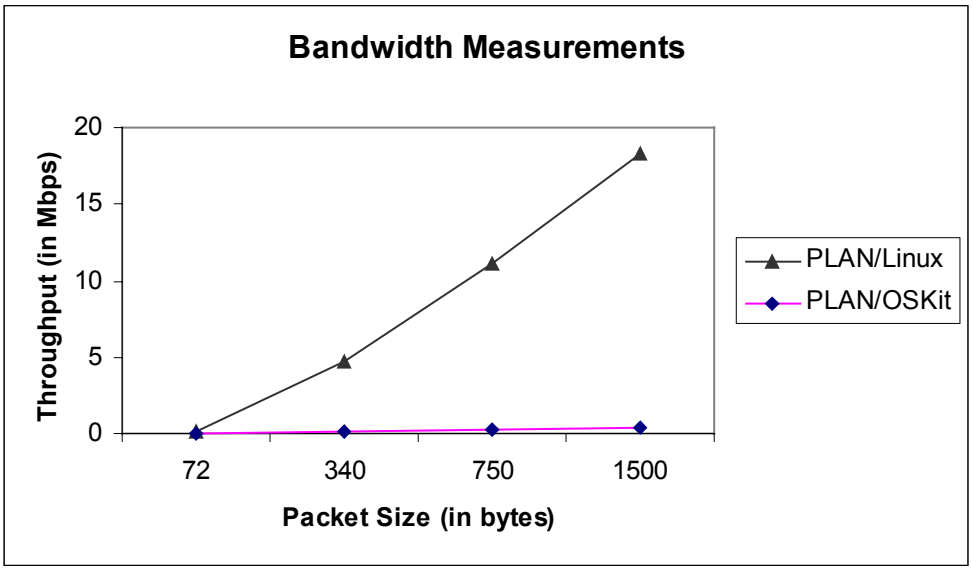


Figure 5.5 Comparison of forwarding performance of PLAN

The above tests do not entirely isolate our problem of interest. The PLAN router communicates with a local application by means of PLAN ports that are implemented using Unix domain sockets. In addition to the forwarding overhead of the switch used on Testnode10, these tests also carry the overhead of transfer of packets between PLAN routers and the active applications, namely the load generator and the receiver. The tests shown in Figure 5.4 reflect the performance achieved by such an application.

5.3.2 Router-level Exchange

The tests described here measure the performance achieved by a PLAN networking system. It makes use of an inbuilt "bandwidth service" provided with the PLAN router. Hence packets are dispatched directly to and from the PLAN routers, avoiding the costs of copying to and from the application. The test framework is shown in Figure 5.6.

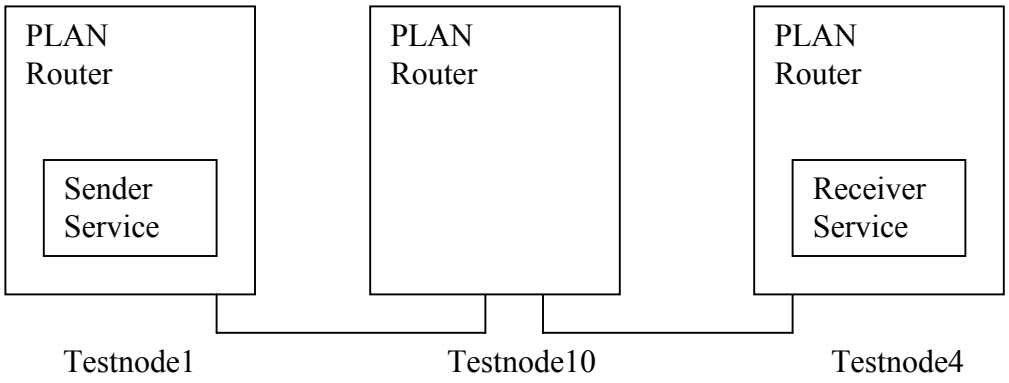


Figure 5.6 Router-level bandwidth tests

The sender service makes use of a script provided with the PLAN code to initiate the service. The sender is made to send 100000 packets with a delay of 100 iterations of an empty loop between each send. The sender initially sends out a connection packet to the destination (testnode4) which starts a receiver thread there. The receiver counts the number of packets arrived and reports the bandwidth depending on the payload of the packet. The results of the test are shown in Figure 5.7

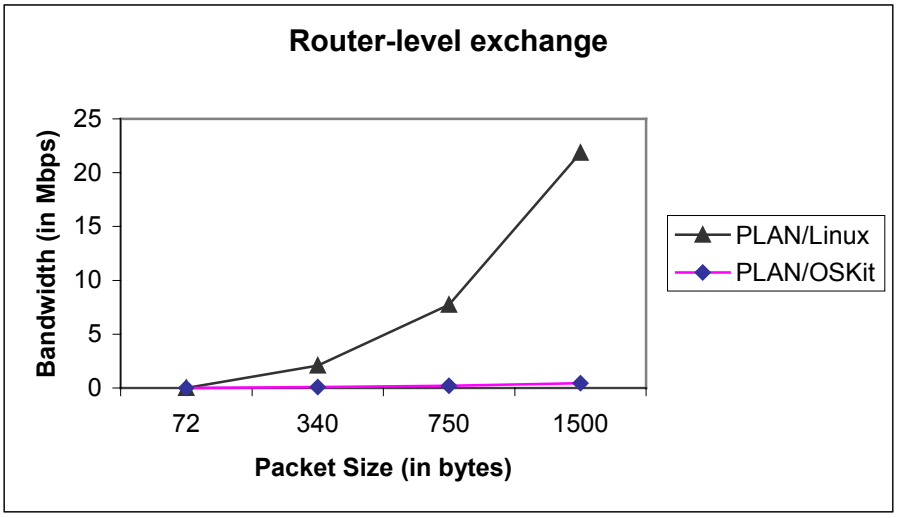


Figure 5.7 Comparison of Routing performance of PLAN

The performance of the PLAN router on OSKit is significantly affected. In the PLAN/Linux case, for an Ethernet packet size of 750 bytes, only 9.4 % of the packets made it to the receiver. The figures for the PLAN/OSKit case were worse. Of the 100000 packets dispatched from Testnode1, the receiver reported only around 370 packets. The above tests involving PLAN were conducted using a version that makes use of queues between its network layer and linklayer. In order to reduce the synchronization overhead associated with queues, the router-level exchange tests were repeated with a new configuration.

The topology for the tests remains the same, as shown in Figure 5.6. The new configuration of PLAN uses direct upcalls between its network layer and linklayer. This helps in removing a lot of thread synchronization overhead. The following results, shown in Figure 5.8, verify this position.

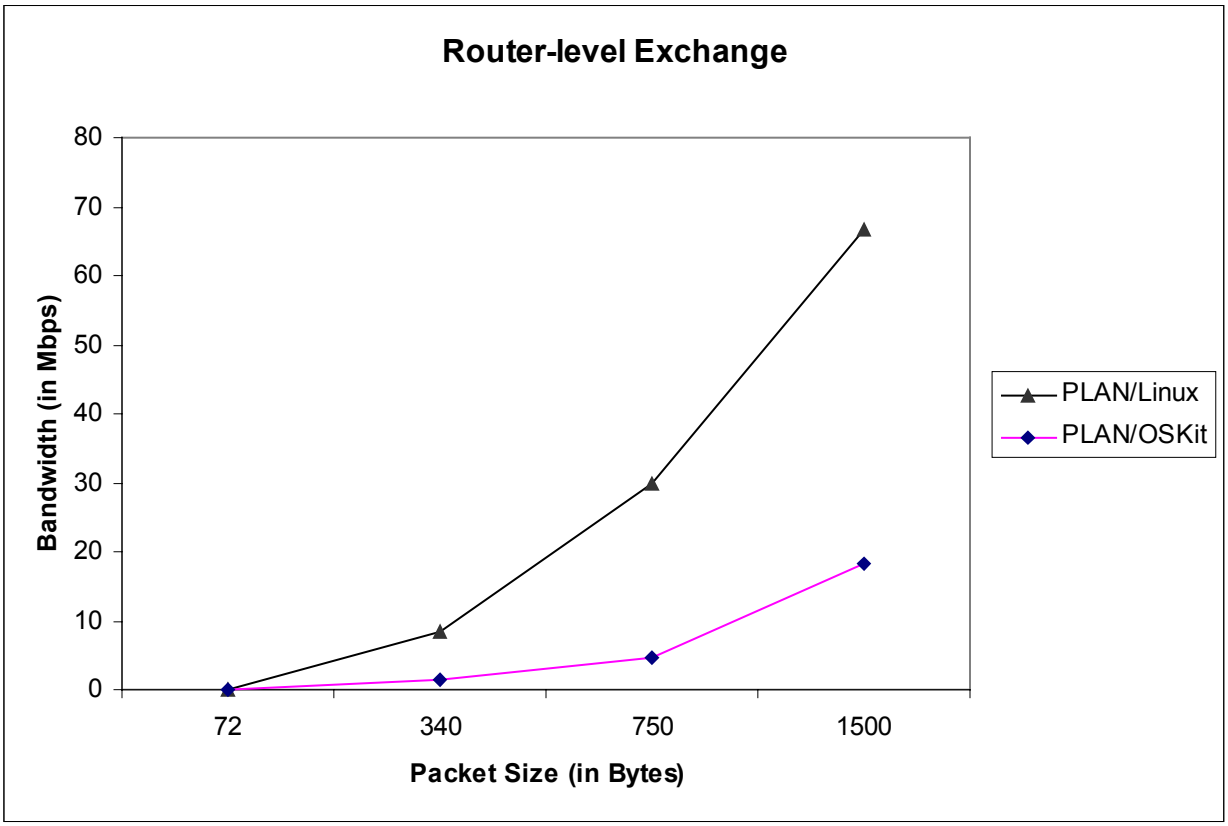


Figure 5.8 Comparison of Routing performance of PLAN (using upcalls)

Figure 5.8 illustrates the performance advantage gained by operating PLAN using upcalls. The numbers of packets that make it to the receiver also register a significant increase in both the cases. In the case of PLAN/Linux, the number of packets received at Testnode4 increases by 4 times. The improvement in performance is more marked in the PLAN/OSKit. As many as 5,600 packets make it to the receiver which is a 14-fold increase over earlier numbers.

A variation was introduced in the above case to ensure that the results scale at smaller numbers. The above test was repeated by sending 1000 packets only instead of

blasting the PLAN router with 100000 packets. All the other parameters associated with the test remained the same. The numbers seen during the test are almost proportionate. The results are summarized in Figure 5.9.

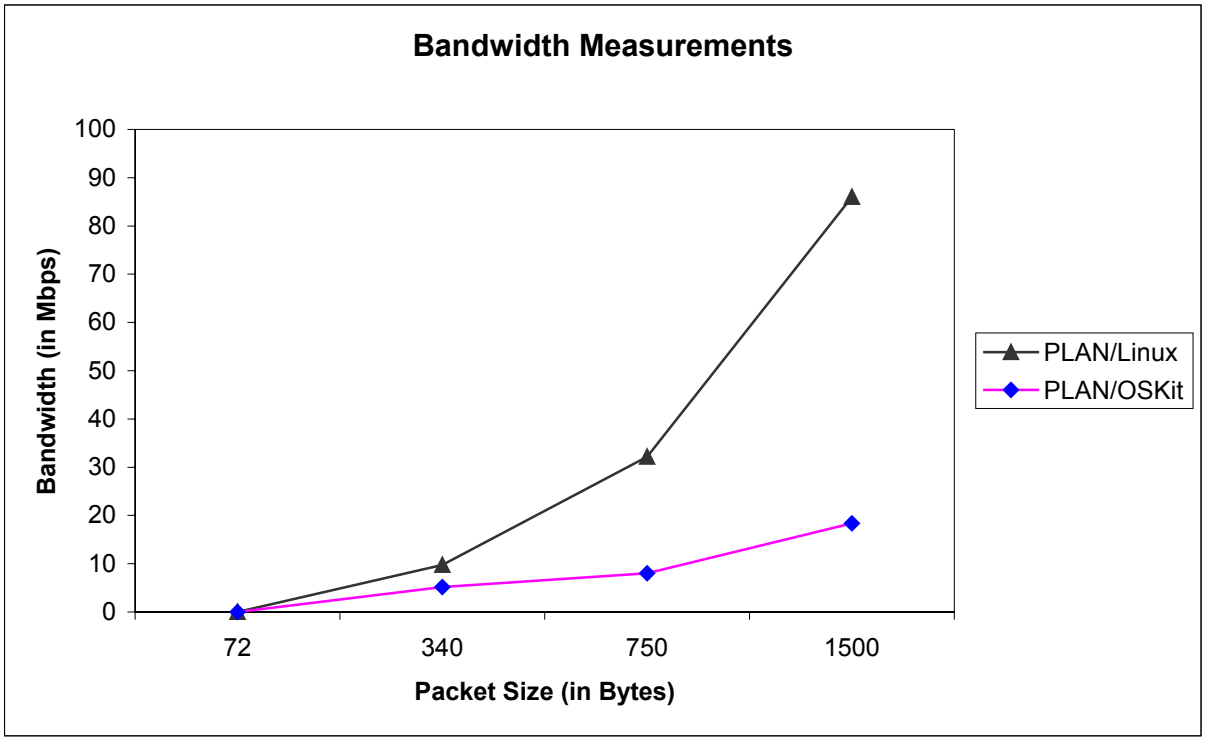


Figure 5.9 Comparison of routing performance of PLAN (under lighter load)

Chapter 6

Conclusions and Future Work

6.1 Summary

We have implemented an active network prototype on a non-conventional experimental operating system. We have ported the OCaml runtime system (both bytecode and native code versions) to OSKit and built the PLAN execution environment of top of this structure. The various components of the system are written in C and OCaml. They are ultimately statically linked into the OSKit kernel. This thesis describes the implementation of such a system and quantifies the performance obtained by such integration.

The performance of OCaml on OSKit has been benchmarked and has been found to be not exceedingly better than that of Linux. Most of the benchmarks do not show any substantial improvement over their Linux counterparts. This is because most of the OSKit uses legacy code used on systems such as Linux or FreeBSD that are bound together by glue code. There is also no provision for optimized data-paths for providing improved performance. For example, the mismatch between the FreeBSD TCP/IP stack and the Linux device driver places an upper bound on its networking performance. Improvement in the benchmark performance can be brought about if we are able to integrate such components into the OSKit framework. The threads benchmark shown tests the synchronization between two producer/consumer threads synchronized by a mutex and condition variable. The bytecode overhead associated with Linux is retained into the OSKit kernel since the OSKit has to run through all the C primitives associated with the

bytecode runtime system. The only way the bytecode overhead is avoided is to compile the OCaml sources to native code.

The effectiveness of the implementation was tested by means of injecting code carrying PLAN packets into an active network and observing the latency. The latency of PLAN/OSKit has been found to be more than that achieved by PLAN/Linux. The forwarding capacity of a PLAN node was measured both on Linux and OSKit. The results show perceptible difference between the two versions of PLAN – namely, queue-based and upcall-based. The queue-based version introduces additional synchronization overhead. The upcall-based version performs better due to the removal of this overhead. In both these versions, the OCaml native threads do not seem to run in parallel on OSKit. These threads prevent the execution of other threads during blocking operations. This results in the dropping of a number of packets due to filling up of socket buffers on OSKit. The OSKit's structure, though modular and separable in nature has been found not thoroughly suitable for an OCaml-based active node.

6.2 Future Work

Another important concept of the Active Networks includes the dynamic deployment of services. It may be necessary to quantify the performance limitations of PLAN on such a system. Further work is also required in integrating the PLAN execution environment with the Moab NodeOS. The integration of the Moab NodeOS should see an improvement in network performance over that shown by the OSKit due to the

availability of optimized data paths which are executed end-to-end. However it will be necessary to map OCaml native threads to NodeOS threads that operate in Moab.

OCaml provides a simple interface to query the status of its garbage collector. PLAN has extended this so as to provide for dynamic querying of the GC status of an active router. The PLAN/OSKit version can be studied to determine the performance of the OCaml garbage collector.

Appendix A

The BlueBox

In our need for the realization of large-scale active network nodes, the Bluebox is an essential tool. The Bluebox, as the name suggests, consists of a set of neatly stacked machines laid out in a blue colored box-on-wheels. It came out of an earlier research project at ITTC, KU. It consists of a compact arrangement of 13 nodes with one of them functioning as a control node. The control computer is a 600 MHz PIII processor with 256MB of memory. In addition to these, it has a 20GB hard disk and can be connected to the other testnodes through twelve 100-Mbps, Ethernet ports. A Netgear 516 16-porter unmanaged Ethernet switch is used to lay out the physical connectivity between the testnodes and the control computer.

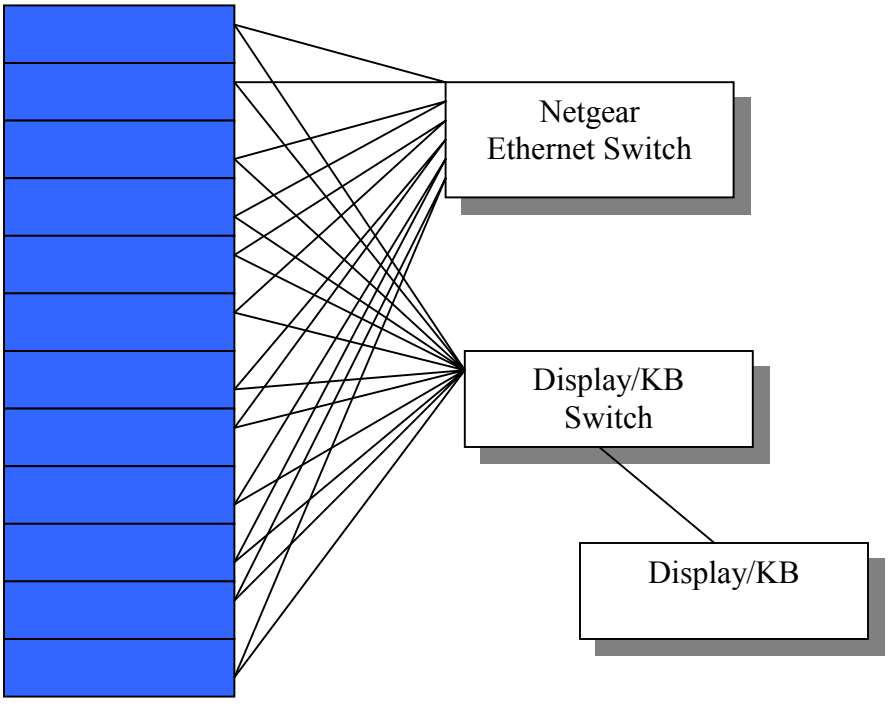


Figure A Physical Interconnect of BlueBox

The testnodes themselves are each equipped with a 533 MHz PIII processor and a 20 GB hard disk. Each of these testnodes also carries 128 MB RAM on it. All the testnodes are equipped with a minimum of two 100 Mbps Ethernet ports. The primary port is used to connect to the Netgear switch that enables it to maintain connectivity with the control computer at all times, which is essential for it to function as a management port. Four of these machines also have 4-port Ethernet cards on them, which enables us to create a number of complex topologies among the testnodes. The secondary ports on these nodes are used to create these complex topologies. A Cisco C2924 switch, with 24 ports, capable of managing these connections is also provided. All the machines are connected to a single LCD monitor and a keyboard through an APEX 16-port KVM switch. A WaveLAN access port is also provided which in conjunction with two laptops completes the emulation of a complex network with a host and a sink as well. The physical interconnections of the Bluebox are shown in Figure A. The secondary connections are not shown in the figure.

The BlueBox had been demonstrated at the DARPA Active Network Conference at Atlanta, Georgia in December 2000 and has won laurels for its compact structure. It will be the primary tool for constructing a large-scale active network testbed at ITTC.

Appendix B

Building of PLAN/NodeOS/OSKit

OSKit:

1. Located at /projects/IANS/oskit-20010214
2. Version 20010214
3. Changes incorporated in modified libraries:
   - `liboskit_threads_KU.a`
   - `liboskit_startup_KU.a`
   - `liboskit_linux_dev_KU.a`
4. All changes have been recorded in README_KU files in respective component directories
5. Extra libraries to measure some net statistics
   - `liboskit_freebsd_net_KU.a`
   - `liboskit_linux_dev_KU_stats.a`
6. Begin by using the makefiles provided to build standard OSKit system
   - cd /projects/IANS/oskit-20010214
   - ./configure --prefix=/projects/IANS/oskit-20010214 -enable-examples
   - make all
   - make install
7. Use the new makefiles listed in the toplevel file README_KU to build the above modified libraries

OCaml:

1. Version 3.00 (Uses pthreads to implement multi-threading)

2. Original source code available at /projects/IANS/caml-world/ocaml-3.00
3. Build process:
    - cd /projects/IANS/caml-world/ocaml-3.00
    - ./configure –prefix /projects/IANS/caml-world/3.00 –with-pthread
    - make world
    - make opt
    - umask 022
    - make install
4. Camlp4
    - cd /projects/IANS/caml-world/camlp4-3.00
    - ./configure –prefix projects/IANS/caml-world/3.00
    - make world
    - make opt
    - make install
5. Modified OCaml runtime system available at /projects/IANS/caml-world/ocaml-3-on-oskit
    - Run configure-oskit in the toplevel directory
    - Build the following runtime libraries using the Makefiles created
        - `libcamlrun.a` in the directory, `byterun`
        - `libasmrun.a` in the directory, `asmrun`
    - Other OS dependent libraries to be rebuilt include:
        - `libunix.a` in the directory, `otherlibs/unix`
        - `libnums.a` in the directory, `otherlibs/nums`

- `libthreads.a` and `libthreadsnat.a` in the directory `otherlibs/systhreads`
- `libstr.a` in the directory `otherlibs/str`

- Build process:
  - cd /projects/IANS/caml-world/ocaml-3-on-oskit
  - ./configure-oskit
  - cd byterun; make libcamlrun.a
  - cd asmrun; make libasmrun.a
  - cd otherlibs/systhreads; make –f Makefile_KU libthreads.a
  - cd otherlibs/str; make libstr.a
  - cd otherlibs/unix; make libunix.a
  - cd otherlibs/nums; make libnums.a

6. OCaml with wrappers around NodeOS API is located at /users/ravi/thesis/ocaml-3.00
7. This version may be used if building a Caml EE that needs to access to NodeOS API.

PLAN:

1. Version 3.21

2. Original PLAN code located at /projects/IANS/plan-3.21. Builds using ocaml-3.00 and camlp4-3.00.

- cd /projects/IANS/plan-3.21
- make all

3. The modified PLAN system code located in /projects/IANS/plan-on-oskit. Code may have to be modified to run on different machines.

- Make all

- ./scr.sh (to custom compile PLAN sources)

Moab NodeOS:

1. Version 20010214
2. Original source code located at /projects/IANS/nodeos-20010214
3. Build process:
   - ./configure --with-oskit=/projects/IANS/oskit-20010214 --host=i386-oskit
   - make all
4. `Libnodeos.a` will be the library built

Bringing together the above components:

1. Makefile located at /projects/IANS/oskit-20010214/examples/nodeos_plan_thesis
2. Building it will link all libraries with an interface file. The interface file is a C file that initializes the Moab machine on OSKit.
   - make
3. The executable formed is used to create a multiboot image using the script provided in the OSKit '`bin`' directory.

Booting the image:

1. The image is copied onto the hard disk of the machine.
2. The image is booted using a GRUB bootloader.
3. Version 0.5.92
4. Source is located at /projects/IANS/oskit_utils/grub-0.5.92
5. A Grub floppy is prepared
6. GRUB commands:

- `kernel  = (hd0,0)/Plan_Image [any options]`
    - This command loads the primary multiboot image from a file.
- `boot`
    - This command boots the kernel loaded through ‘kernel’ command and also loads any modules needed by the kernel.

4. The exact parameters of the ‘kernel’ command depend on the partitions in the hard disk and file name of the image. The bootloader passes any options it receives to the OSKit kernel that in turn passes it to the OCaml virtual machine.

Fixes to various components:

1. OSKit:

Threads Library

File: threads/osenv_lock.c

The following functions were commented:

`osenv_process_lock`

`osenv_process_unlock`

`osenv_process_locked`

File: threads/osenv_sleep.c

The following functions were commented:

`osenv_sleep_init`

`osenv_sleep`

`osenv_wakeup`

Startup Library:

File: startup/start_conf_network.c

This file has to be modified to suit the Ethernet interfaces for a particular machine

2. PLAN:

PLANport makes use of loopback address. This has been changed to make use of the interface INADDR_ANY since OSKit does not support the loopback interface.

All file operations changed to string operations

3. OCaml:

File: byterun/debugger.c

Changed '`wait`' system call to map to that of OSKit.

File:byterun/sys.c

Changed '`system`' function to return 1.

File:asmrun/sys.c

Changed '`clock`' function to return 1

File:otherlibs/unix/getgroups.c

Changed '`getgroups`' system call to return 1

File:otherlibs/unix/getlogin.c

Changed '`getlogin`' to return 1

File:otherlibs/unix/nice.c

Changed '`getpriority`' and '`setpriority`' to return 1

File:otherlibs/unix/setsid.c

Changed '`setsid`' to return 1

File:otherlibs/unix/signals.c

Changed '`sigpending`' to return 1

File:otherlibs/unix/sioc.c

Changed the function '`unix_siocgifhwaddr`' to return 1.

File:otherlibs/unix/wait.c

Changed '`wait`' system call to map to that of OSKit.

References

[1] B. Ford, G. Back, G. Benson, J. Lepereau, A. Lin, O. Shivers, "*The Flux OSKit: A Substrate for kernel and Language Research,*" Proceedings of the 16th ACM Symposium on Operating Systems Principles, October 1997.

[2] Didier Remy, Xavier Leroy, Pierre Weis, *"Objective Caml – A general purpose high-level programming language*," INRIA Rocquencout, France, ERCIM News, (36), January 1999.

[3] "The Caml Language," http://caml.inria.fr/

[4] "The OSKit Homepage," http://www.cs.utah.edu/flux/oskit/

[5] Michael Hicks, Pankaj Kakkar, Jonathan T. Moore, Carl A. Gunter, and Scott Nettles, "*PLAN: A packet language for Active Networks,*" Proceeding of the Third ACM SIGPLAN International Conference on Functional Programming Languages, 1998, pp. 86-93, ACM.

[6] Larry Peterson, Yitzchak Gottlieb, Mike Hibler, Patrick Tullmann, Jay Lepreau, Stephen Schwab, Hrishikesh Dandekar, Andrew Purtell, and John Hartman, "*An OS Interface for Active Routers,*" IEEE Journal on Selected Areas in Communications, 2001.

[7] AN Node OS Working group, NodeOS Interface specification, January 2001.

[8] Patrick Tullmann, Mike Hibler, Jay Lepreau, "*Janos: A Java-oriented OS for Active Network Nodes,*" IEEE Journal on Selected Areas in Communications, Active and Programmable Networks, 2001.

[9] Michael Hicks, Jonathan T. Moore, D. Scott Alexander, Carl A. Gunter, and Scott M. Nettles, "*PLANet: An Active Internetwork,*" IEEE InfoComm, 1999.

[10] David J. Tennenhouse, Jonathan M. Smith, W. David Sincoskie, David J. Wetherall, and Gary J. Minden, "*A survey of Active Network research*", IEEE Communications Magazine, January 1997.

[11] D. Scott Alexander, William A. Arbaugh, Michael Hicks, Pankaj Kakkar, Angelos Keromytis, Jonathan T. Moore, Carl A. Gunter, Scott M. Nettles, and Jonathan M. Smith. "*The switchware active network architecture,*" IEEE Network Magazine, 1998.

[12] Michael Hicks, Pankaj Kakkar, Jonathan T. Moore, Carl A. Gunter, and Scott Nettles, "*Network programming using PLAN*," In IPL Workshop '98, 1998.

[13] Michael Hicks, Pankaj Kakkar, Jonathan T. Moore, Carl A. Gunter, D. Scott Alexander and Scott Nettles, "*The PLAN system for building Active Networks,*" February 1998.

[14] "The Great Computer Language Shootout," http://www.bagley.org/~doug/shootout/.

[15] Isfahan Deendar, "*Implementation and Performance Evaluation of an Active Network on a Communication-oriented Operating system,*" Master's Thesis, September 1999.

[16] David Wetherall, John Guttag and David Tennenhouse. *"ANTS: A toolkit for building and dynamically deploying network protocols,*" IEEE Openarch 98, April 1998.

[17] A. Kulkarni, G. Minden, R.Hill, Y.Wijata, A.Gopinath, S.Sheth, F.Wahhab, H.Pindi and A.Nagarajan, "*Implementation of a Prototype Active Network,*" Openarch 98, April 1998.

[18] David Mosberger, "*SCOUT: A Path-based Operating System,*" Ph.D. Dissertation, Department of Computer Science, University of Arizona, November 1997.

[19] John Hartman, Larry Peterson, Andy Bavier, Peter Bigot, Patrick Bridges, Brady Montz, Rob Plitz, Todd Proebsting and Oliver Spatscheck, "*JOUST: A Platform for Liquid Software,*" IEE Computer 1999.

[20] Paul Menage, "*RCANE: A Resource Controlled Framework for Active Network Services,*" IWAN 99, June 1999.

[21] Pizza home page. http://www.math.luc.edu/pizza.

[22] Microsoft Corporation and Digital Equipment Corporation. *"Component Object Model Specification,"* October 1995.

[23] Multiboot Standard, http://www.nilo.org/multiboot.html.