An OS Interface for Active Routers

Larry Peterson, Yitzchak Gottlieb, Mike Hibler, Patrick Tullmann, Jay Lepreau, Stephen Schwab, Hrishikesh Dandekar, Andrew Purtell, and John Hartman

Abstract— **This paper describes an operating system interface for active routers. This interface allows code loaded into active routers to access the router's memory, communication, and computational resources on behalf of different packet flows. In addition to motivating and describing the interface, the paper also reports our experiences implementing the interface in three different OS environments: Scout, the OSKit, and the exokernel.**

Keywords— **Active networks, programmable networks, operating systems.**

I. Introduction

Active networks offer the promise of being able to customize the network on an application-by-application basis [26]. This is accomplished by allowing packet flows to inject code into the routers they traverse. One of the central challenges in designing an active network is to define the interface that this code is written to. This interface specifies the services and resources the active code is able to access on every node (router) in the network.

The design space for customizing networks is large, and includes approaches commonly referred to as "active networks" (the focus of this paper) and "programmable networks" (an approach being pursued in industry). The latter approach—as exemplified by the IEEE P1520 working group [12] and Nortel's Open IP [16]—involves a more restrictive programming environment, in which the network's signalling and control functions are programmable, but the data transfer functions are fixed. In contrast, active networks permit applications to customize both the control plane and the data plane.

A general architecture for active networks has evolved over the last few years [24]. This architecture identifies three layers of code running on each active node (Figure 1). At the lowest level, an underlying operating system (NodeOS) multiplexes the node's communication, memory, and computational resources among the various packet flows that traverse the node. At the next level, one or more execution environments (EE) define a particular programming model for writing active applications. To date, several EEs have been defined, including ANTS [30], [29], PLAN [3], [11], and CANES [5]. At the topmost level are the active applications (AA) themselves.

Authors' current addresses: L. Peterson and Y. Gottlieb, Department of Computer Science, 35 Olden Street, Princeton, NJ 08544 ({llp,zuki}@cs.princeton.edu); M. Hibler, P. Tullmann and J. Lepreau, University of Utah School of Computing, 50 S. Central Campus Drive Rm. 3190, Salt Lake City, UT 84112 ({mike,tullmann,lepreau}@cs.utah.edu); S. Schwab and H. Dandekar and A. Purtell, NAI Labs, Network Associates, 3415 S. Sepulveda Blvd., Suite 700, Los Angeles, CA 90034 ({sschwab,hdandeka,apurtell}@nai.com); and J. Hartman, Department of Computer Science, University of Arizona, Tucson, AZ 85721 (jhh@cs.arizona.edu).

This work was supported in part by DARPA contracts N66001–96–8518, N66001–97–C–8514, and DABT63–94–C–0058, and NSF grants ANI–99–06704 and ANI–00–82493.

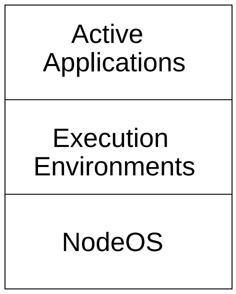

Fig. 1. Software layers running on an active router.

We were major contributors to developing and documenting the interface [1] between the bottom two layers in Figure 1. This paper focuses on that interface, making two contributions. The first is to motivate and describe the NodeOS interface. While similar in many respects to a standard API such as POSIX, the emphasis of an active router on forwarding packets makes this interface unique in many ways. The second contribution is to report our experiences implementing the interface in three different OS environments: within the Scout kernel [21], using the OSKit component base [9], and above the exokernel [13]. Although none of these implementations is complete, each exposes an interesting set of implementation issues for a significant subset of the NodeOS interface.

II. Design Rationale

The goal of active networks is to make the network as programmable as possible, while retaining enough common interfaces so that active applications injected into the network can run on as many nodes as possible. In this context, it is not obvious where to draw the line between the EEs and the NodeOS. One answer is that there is no line: a single layer implements all the services required by the active applications. This is analogous to implementing a language runtime system directly on the hardware, as some JavaOSs have done. However, separating the OS from the runtime system makes it easier for a single node to support multiple languages. It also makes it easier to port any single language to many node types. This is exactly the rationale for defining a common NodeOS interface.

Deciding to separate the NodeOS and the EEs is only the first step; the second step is to decide where the EE/NodeOS boundary should be drawn. Generally speaking, the NodeOS is responsible for multiplexing the node's resources among various packet flows, while the EE's role is to offer AAs a sufficiently high-level programming environment. This is loosely analogous to the distinction between an exokernel and an OS library [13]. Beyond this general goal, the NodeOS interface is influenced by both a set of high-level design goals, and our experiences implementing the interface on a collection of OS platforms. The rest of this section identifies the high-level design decisions that gave the interface its general shape, while Sections IV–VI discuss how various implementation factors influenced particular details of the interface.

The first, and most important design decision was that the interface's primary role is to support packet forwarding, as opposed to running arbitrary computations. As a consequence, the interface is designed around the idea of network packet flows [6]: packet processing, accounting for resource usage, and admission control are all done on a per-flow basis. Also, because network flows can be defined at different granularities—e.g., port-to-port, host-to-host, per-application—the interface cannot prescribe a single definition of a flow.

Second, we do not assume that all implementations of the NodeOS interface will export exactly the same feature set—some implementations will have special capabilities that EEs (and AAs) may want to take advantage of. The interface should allow access to these advanced features. One important feature is the hardware's ability to forward certain kinds of packets (e.g., non-active IP) at very high speeds. Said another way, packets that require minimal processing should incur minimal overhead. A second feature is the ability to extend the underlying OS itself, i.e., extensibility is not reserved for the EEs that run on top of the interface. The NodeOS interface must allow EEs to exploit these extensions, but for reasons of simplicity, efficiency, and breadth of acceptable implementations, the NodeOS need not provide a means for an EE to extend the NodeOS directly. Exactly how a particular OS is extended is an OS-specific issue.

Our final design decision was a pragmatic one: whenever the NodeOS requires a mechanism that is not particularly unique to active networks, the NodeOS interface should borrow from established interfaces, such as POSIX.

III. Architecture

The NodeOS interface defines five primary abstractions: *thread pools, memory pools, channels, files*, and *domains* [1]. The first four encapsulate a system's four types of resources: computation, memory, communication, and persistent storage. The fifth abstraction, the domain, is used to aggregate control and scheduling of the other four abstractions. This section motivates and describes these five abstractions, and explains the relationships among them. Of the five abstractions, domains and channels are the most novel (NodeOS-specific), threads and memory are variations on traditional designs, and files are mostly standard.

A. Domains

The domain is the primary abstraction for accounting, admission control, and scheduling in the system. Domains directly follow from our first design decision: each domain contains the resources needed to carry a particular packet flow. A domain typically contains the following resources (Figure 2): 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 (dotted arc), and are then transmitted on an output channel (outChan).

Note that a channel consumes not only network bandwidth, but also CPU cycles and memory buffers. The threads that shepherd messages across the domain's channels come from the domain's thread pool and the cycles they consume are charged to that pool. Similarly, the I/O buffers used to queue messages on a domain's channels are allocated from (and charged to) the domain's memory pool. In other words, one can think of a domain as encapsulating resources used across both the NodeOS and an EE on behalf of a packet flow, similar to resource containers [4] and Scout paths [25].

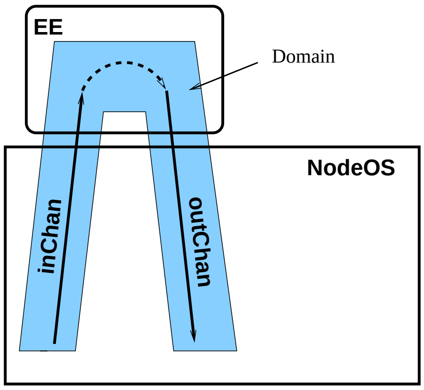

Fig. 2. A domain consists of channels, memory, and threads needed for EE-specific processing.

A given domain is created in the context of an existing domain, making it natural to organize domains in a hierarchy, with the root domain corresponding to the NodeOS itself. Figure 3 shows a representative domain hierarchy, where the second level of the hierarchy corresponds to EEs and domains at lower levels are EE-specific. In this example, the EE implemented in Domain A has chosen to implement independent packet flows in their own domains (Domain C through Z), while the EE running in Domain B aggregates all packets on a single set of channel, memory, and thread resources. The advantage of using domains that correspond to fine-grained packet flows—as is the case with the EE contained in Domain A—is that the NodeOS is able to allocate and schedule resources on a per-flow basis. (Domain A also has its own channels, which might carry EE control packets that belong to no specific sub-flow.)

The domain hierarchy is used solely to constrain domain termination. A domain can be terminated by the domain itself, by the parent domain that created it, or by the NodeOS because the domain has violated some resource usage policy. Domain termination causes the domain and all its children to terminate, the domain's parent to be noti-

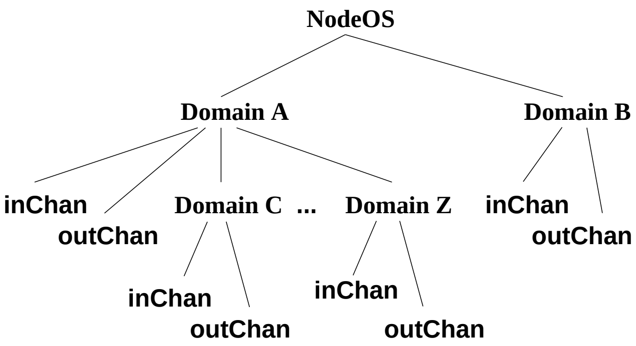

Fig. 3. Domain hierarchy.

fied, and all resources belonging to the terminated domains are returned to the NodeOS.

Each parent domain contains a handler that is invoked just before a child domain is terminated by the NodeOS. This "imminent termination" handler allows the parent domain (generally running the EE) to reconcile any state it may have associated with the dying domain and free any resources it may have allocated on the child domain's behalf. The handler is invoked in the context of a thread in the parent domain; thus the parent domain pays for cleaning up an errant child domain. The handler is given a small, fixed amount of time to complete its cleanup. If the thread exceeds this limit, it, and the domain in which it runs, are terminated.

In contrast to many hierarchical resource systems (e.g., stride CPU schedulers [28]), 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; resources given a child domain are not deducted from the parent's allocation, and resources belonging to a child domain are not returned to the parent domain when the child terminates. This design was based on the observation that requiring resources to be allocated in the same hierarchical manner as domains results is an overly restrictive resource model. For example, suppose an ANTS EE runs in a domain and creates new (sub)domains in response to incoming code capsules. These new domains should be given resources based solely on their credentials (identity). They should not be restricted to some subset of the ANTS EE's resources, which they would be if resources followed the domain hierarchy.

B. Thread Pool

The *thread pool* is the primary abstraction for computation. Each domain contains a single thread pool that is initialized when the domain is created. Several parameters are specified when creating a thread pool, including the maximum number of threads in the pool, the scheduler to be used, the cycle rate at which the pool is allowed to consume the CPU, the maximum length of time a thread can execute between yields, the stack size for each thread, and so on.

Because of our decision to tailor the interface to support packet forwarding, threads execute "end-to-end"; that is, to forward a packet they typically execute input channel code, EE-specific code, and output channel code. Since a given domain cuts across the NodeOS and an EE, threads must also cut across the NodeOS/EE boundary (at least logically). This makes it possible to do end-to-end accounting for resource usage. Note that from the perspective of the NodeOS interface, this means that the thread pool primarily exists for accounting purposes. Whether or not a given NodeOS pre-allocates the specified number of threads is an implementation issue. Moreover, even if the NodeOS does pre-allocate threads, these threads may not be able to handle all computation that takes place on behalf of the thread pool; for example, they may not be allowed to run in supervisor mode. Any thread running on behalf of the thread pool, no matter how its implemented, is charged to the pool.

The fact that a thread pool is initialized when a domain is created, and threads run end-to-end, has two implications. First, there is no explicit operation for creating threads. Instead, threads in the pool are implicitly activated, and scheduled to run, in response to certain events, such as message arrival, timers firing, and kernel exceptions. Second, there is no explicit operation for terminating a thread. Should a thread misbehave—e.g., run beyond its CPU limit—the entire domain is terminated. This is necessary since it is likely that a thread running in an EE has already executed channel-specific code, and killing the thread might leave the channel in an inconsistent state.

As just described, threads are short-lived, "data driven" entities with no need for explicit identities. While this is sufficient for many environments, our experience with Janos, detailed in Section V, indicates that some EEs require "system" threads that are long-lived and not associated with any particular packet flow. For example, a JVM-based EE might have a global garbage collection thread that, when it runs, needs to first stop all other threads until it is done. To support these environments, the API defines a small set of pthread-inspired operations for explicit thread manipulation: sending an interrupt, blocking and unblocking interrupts, changing a scheduler-interpreted priority value, and attaching thread-specific data.

C. Memory Pool

The *memory pool* is the primary abstraction for memory. It is used to implement packet buffers (see Section III-D) and hold EE-specific state. A memory pool combines the memory resources for one or more domains, making those resources available to all threads associated with the domains. Adding domains to a pool increases the available resources while removing domains decreases the resources. The amount of resources that an individual domain can contribute to a pool is either embodied directly in the domain's credentials or explicitly associated with the domain at creation time. The many-to-one mapping of domains to memory pools accommodates EEs that want or need to manage memory resources themselves. For example, as illustrated in Section V-A, this mapping is needed by a

JVM-based EE that shares objects and JIT'ed code between domains.

Memory pools have an associated callback function that is invoked by the NodeOS whenever the resource limits of the pool have been exceeded (either by a new allocation or by removing a domain from the pool). The callback function is registered when a memory pool is created by an EE. The NodeOS relies on the EE to release memory when asked; i.e., the NodeOS detects when a pool is over limit and performs a callback to the EE to remedy the situation. If the EE does not free memory in a timely manner, the NodeOS terminates all the domains associated with the pool. The rationale for these semantics is similar to that for domain termination give above: the EE is given a chance to clean up gracefully, but the NodeOS has fallback authority.

Memory pools can be arranged hierarchically to allow constrained sharing between pools. The hierarchy of mempools is not used to control the propagation of resources; rather, it is intended as an access control mechanism. Specifically, a "child" mempool does not inherit its memory resources from its "parent"; those resources come from domains that are attached to the pool. Instead, the mempool hierarchy allows for sharing of memory between pools: a parent may see all of a child's memory while limiting what the child may see of its own. The semantics of the mempool hierarchy are motivated by the desire to support multiple address spaces.

D. Channels

Domains create channels to send, receive, and forward packets. Some channels are *anchored* in an EE; anchored channels are used to send packets between the execution environment and the underlying communication substrate. Anchored channels are further characterized as being either incoming (inChan) or outgoing (outChan). Other channels are *cut-through* (cutChan), meaning that they forward packets through the active node—from an input device to an output device—without being intercepted and processed by an EE. Clearly, channels play a central role in supporting our flow-oriented design. We crystallize this role at the end of this subsection; first we describe the various types of channels in more detail.

When creating an inChan, a domain must specify several 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. The buffer pool is created out of the domain's memory pool. The packet handler is passed the packet being delivered, and is executed in the context of the owning domain's thread pool.

When creating an outChan, the domain must specify (1) where the packets are to be delivered and (2) how much link bandwidth the channel is allowed to consume (guaranteed to get). Packet delivery is specified through a protocol specification string coupled with an address specification string. The link bandwidth is described with an RSVP-like QoS spec [34].

Cut-through channels both receive and transmit packets. A cutChan can be created by concatenating an existing inChan to an existing outChan. A convenience function allows an EE to create a cutChan from scratch by giving all the arguments required to create an inChan/outChan pair. Cut-through channels, like input and output channels, are contained within some domain, that is, the cycles and memory used by a cutChan are charged to the containing domain's thread and memory pool. Figure 4 illustrates an example use of cut-through channels, in which "data" packets might forwarded though the cut-through channel inside the NodeOS, while "control" packets continue to be delivered to the EE on an input channel, processed by the EE, and sent on an output channel.

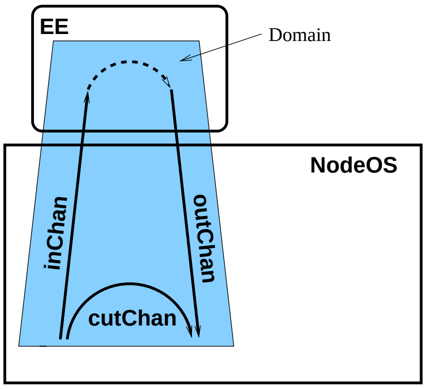

Fig. 4. A domain with a cut-through channel.

The protocol and address specifications for inChans and outChans are similar, and are largely adapted from Scout's path abstraction (Section IV). The protocol specification is composed of modules built into the NodeOS. For example, "ip", "udp", or "anep". Components are separated in the specification string by the '/' character. Included at one end of a protocol specification is the interface on which packets arrive or depart. Thus, a minimal specification is "if" (for all interfaces) or "if*N*" where *N* is the identifier of a specific interface. For example "if0/ip/udp/anep" specifies incoming ANEP packets tunneled through IP, while "ip/if" specifies outgoing IP packets. The address specification defines destination addressing information (e.g., the destination UDP port number). The format of the address is specific to the highest level protocol in the protocol specification (e.g., describing UDP addresses). cutChan protocol specifications have an identical syntax with the addition of a '|' symbol to denote the transition from incoming packet processing to outgoing packet processing; e.g., example, "ip/udp|udp/ip".

Simply specifying the protocol and addressing information is insufficient when an EE wants to demultiplex multiple packet flows out of a single protocol (e.g., from a single UDP port). The demux key passed to the inChan specifies a set of (offset, length, value, mask) 4-tuples. These tuples are compared in the obvious way to the "payload" of the protocol. The "payload" is defined as the non-header portion of the packet for whatever protocol specification was given. For example, with a raw "if" specification, the payload is everything after the physical headers; with an "if/ip/udp" specification the payload is the UDP payload. Convenience functions are provided for creating filters that match well-known headers.

Note that demux keys and protocol specifications logically overlap. The distinction is in the processing done on the packets by the NodeOS. For example, an EE can receive UDP port 1973 packets by creating an inChan with a protocol of "if0" and demux key that matches the appropriate IP and UDP header bits, or by creating an inChan with a protocol of "if0/ip/udp". The important and critical distinction is that the former case will not catch fragments at all, while the latter will perform reassembly and deliver complete UDP packets. Additionally, the former will provide the IP and UDP headers as part of the received packet where the latter will not.

We conclude our description of channels by revisiting our design goals. First, it is correct to view channels and domains as collectively supporting a flow-centric model: the domain encapsulates the resources that are applied to a flow, while the channel specifies what packets belong to the flow and what function is to be applied to the flow. The packets that belong to the flow are specified with a combination of addressing information and demux key, while the function that is to be applied to the flow is specified with a combination of module names (e.g., "if0/ip/udp") and the handler function.

Second, cut-through channels are primarily motivated by the desire to allow the NodeOS to forward packets without EE or AA involvement. Notice that a cutChan might correspond to a standard forwarding path that the NodeOS implements very efficiently (perhaps even in hardware), but it might also correspond to a forwarding path that includes an OS-specific extension. In the former case, the EE that creates the cutChan is able to control the channel's behavior, similar to the control allowed by API defined for programmable networks [12], [16]. In the latter case, the EE that creates the cutChan is able to *name* the extension (e.g., "if0/ip/extension/if1") and specify parameters according to a standard interface, but exactly how this extension gets loaded and its interface to the rest of the kernel is an OS-specific issue; the NodeOS interface does not prescribe how this happens. In other words, cut-through channels allow EEs to exploit both performance and extensibility capabilities of the NodeOS.

E. Files

Files provide persistent storage and coarse-grained sharing of data. Because we did not view active networks as requiring novel file system support, we adopted an interface that loosely follows POSIX 1003.1. Each EE sees a distinct view of the persistent filesystem, rooted at a directory chosen at configuration time. In other words, "/" for the ANTS EE is rooted at /ANTS. This insulates EEs from each other with respect to the persistent filesystem namespace. In order to accommodate environments in which EE file sharing is desirable, however, we expect to add an interface that allows EEs to access the shared portion of the namespace.

EEs may share information through the use of shared memory regions, which are created with a combination of shm_open and mmap operations. A non-persistent file object is first created with shm_open, which allows the specification of a name, as well as access rights and other options. Once the file object is created, EEs may then mmap the object to create a region of memory that is either private (not shared), or shared among the EEs mapping that file object.

IV. Implementation I: Scout

We have implemented the NodeOS in the Scout operating system, which encapsulates the flow of I/O data through the system—from input device to output device—in an explicit path abstraction [21]. This similarity to the NodeOS interface allows Scout to implement both the traditional and active forwarding services using exactly the same mechanism. This makes it possible to integrate the NodeOS interface into a Scout-based router in a way that does not negatively impact our ability to forward non-active packets.

A. Overview

Scout is a configurable system, where an instance of Scout is constructed from a set of modules. For example, Figure 5 shows a portion of the module graph for an active router. Modules TCP, UDP, IP, and ANEP each implement a communication protocol. Modules JVM and NodeOS each implement an API—the former implements the Java Virtual Machine (see [10]) and the latter implements the NodeOS.

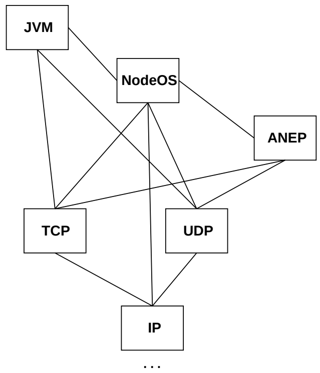

Fig. 5. Module Graph for an Active Router

Scout paths support data flows through the module graph between any pair of devices. When configured to implement a router, Scout supports network-to-network paths, which we call *forwarding paths*. For example, Figure 6 depicts a forwarding path that implements an FTP proxy.

The entity that creates a forwarding path specifies three pieces of information: (1) the sequence of modules that define how the path processes messages, (2) a demultiplexing key that identifies what packets will be processed by the

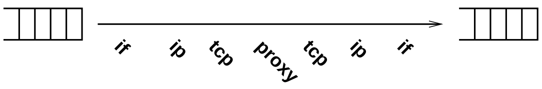

Fig. 6. Example Forwarding Path

path, and (3) the resource limits placed on the path, including how many packets can be buffered in its input queue, the rate at which it is allowed to consume CPU cycles, and the share of the link's bandwidth it may consume. This same information is required by the NodeOS: a domain is a container for the necessary resources (channels, threads, and memory), while a channel is specified by giving the desired processing modules and demultiplexing keys. As a consequence, the NodeOS module is able to implement domain, channel, thread, and memory operations as simple wrappers around Scout's path operations.

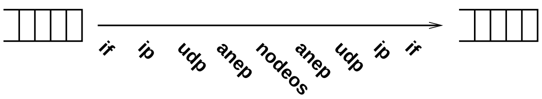

Fig. 7. In and Out Channels Connecting the NodeOS to the Network

More interestingly, a cut-through channel is simply implemented by a Scout forwarding path that does not pass through the NodeOS module (similar to the one shown in Figure 6), while in and out channels map onto a Scout path that does include the NodeOS module (as shown in Figure 7). In the latter case, the inChan corresponds the portion of the forwarding path to the left of the NodeOS module, while the outChan corresponds to the portion of the forwarding path to the right of the NodeOS module.

The only issue is how to implement demultiplexing. In Scout, each module implements two functions: one that *processes* packets as they flow along a path, and one that *demultiplexes* incoming packets to select which path should process the packet. Packet classification is accomplished incrementally, with each module's demux function making a partial classification decision using module-specific information. This approach to classifying packets causes two complications.

First, the NodeOS module's demux function must implement a programmable pattern matcher that recognizes application-specified keys. (Its processing function simply implements the wrappers described above.) Thus, to match a packet to an inChan that delivers "if/ip/udp/anep" packets for a specific ANTS protocol (e.g., protocol '8'), the interface's demux looks for type=IP, IP's demux would look for protnum=UDP, UDP's demux would look for the well-known ANEP port, and ANEP's demux would look for ANTS's well-known EE number. If we assume the ANTS protocol and capsule type fields are each four bytes, the NodeOS's demux would then match the pattern (3, 5, x08x00x00x00x02, xFFx00x00x00x02) to select the path that delivers packets to this specific ANTS protocol (i.e., to a specific active application running in the ANTS execution environment).

Second, the NodeOS does not tightly couple the demultiplex keys with the processing modules. Thus, it is possible to say that packets matching a certain IP address *and* TCP port numbers should be processed by just the IP module. This happens, for example, with transparent proxies. Scout, however, couples the two: creating a path with modules "if/ip/tcp" implies that both the demultiplexing and processing functions of all three modules are involved. Our experience implementing the NodeOS interface has caused us to change Scout so that processing and demultiplexing are not so tightly coupled. That is, path creation now takes two sets of module lists, one that identifies how the path processes packets and one that specifies how packets are classified.

B. Integrated Perspective

The goal of the Scout-based router is to provide a single framework for vanilla IP forwarding, kernel extensions to IP forwarding, and active forwarding [14]. From Scout's perspective, we can view a forwarding path as being constructed from a combination of *system* modules and *user* modules. Systems modules correspond to native Scout modules, which have two important attributes: (1) they are trusted and can be safely loaded by verifying a digital signiture, and (2) they assume the same programming environment (they are written in C and depend on Scout interfaces). In contrast, user modules are untrusted and can be implemented in any programming environment. For example, an application that wants to establish a virtual private network might create a forwarding path that includes a VPN module running over IP tunnels: "if/ip/[vpn]/ip/if", where we bracket VPN to denote that it is a user module.

Of course, the key to being able to run such user modules is to execute them in a wrapper environment, which in our case is provided by the NodeOS interface. The NodeOS, in turn, allows user-provided modules to define their own internal paradigm for extending router functionality. For example, the VPN module might be implemented in the ANTS execution environment. ANTS happens to depend on Java, which is an execution environment in its own right. In effect, if one were to "open" the user-provided VPN in this example, one might see the components shown in Figure 8, where VPN is one of possibly many active applications that ANTS might support at any given time. The only restriction on such nested environments is that the NodeOS is at the outermost level.

Given this perspective, we comment on several attributes of our design. First, some forwarding paths consist entirely of Scout modules; the vanilla IP forwarding path is a prime example. This makes it possible to implement high-performance paths that are not encumbered by the overheads of the NodeOS or Java wrapper environments. In fact, we expect that popular user-provided modules will be re-written as system modules, and migrate into the kernel over time. More generally, being able to run both system and user modules gives us a two-tier security model, allowing both trusted system adminsitrators and untrusted

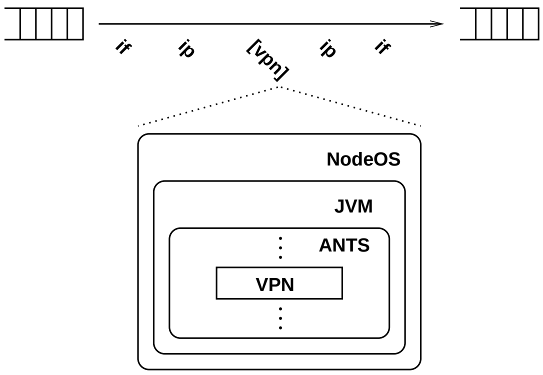

Fig. 8. Internal Structure of a User-Provided Module

users to extend a router's functionality.

Second, a Scout forwarding path contains at most one user module, but by being able to name the system modules that make up the rest of path, the user module is able to exploit legacy network modules. For example, in contrast to the overlay extension illustrated in Figure 8 that logically runs on top of the fully-connected Internet, one could implement an extensible IP user module—think of this module as implementing IP version n—that depends only on the ability to send and receive packets over the raw interface. The important point is that user-provided functionality can be inserted at any level of the network stack.

Third, there are two reasons for deciding to create a new forwarding path. One is that you want to bind a particular packet flow to a unique set of modules. For example, our router typically runs two vanilla IP forwarding paths: one that implements the fast path that consists of a single module that is optimized for moving option-free datagrams between a pair of similar devices, and a general forwarding path that deals with exceptional cases such as IP options and arbitrary interfaces. The second reason is that you want to treat a particular flow specially with respect to resource allocation. For example, there might be two forwarding paths consisting of the modules "if/ip/if" with one given best effort service and the other given some QoS reservation.

Fourth, forwarding paths exist in both the control and data plane, with control paths typically creating and controlling data paths. For example, a path constructed from modules "if/ip/udp/[rsvp]/udp/ip/if" might create the "if/ip/if" path described in the previous paragraph, and in doing so, set the amount of link and CPU resources that the flow is allowed to consume.

C. Performance

We have implemented several forwarding paths, including ones that span the NodeOS. Table I reports the performance of these paths on a 450MHz Pentium-II processor with three Tulip 100Mbps ethernet interfaces. For each type of forwarding path, we give the aggregate rate, measured in packets-per-second (pps) at which the router can forward minimim-sized packets over that path.

Forwarding Path	Rate (Kpps)
IP Fast Path	290.0
General IP	104.5
Active IP	92.5
Transparent Kernel Proxy	85.5
Transparent Active Proxy	84.9
Classical Kernel Proxy	77.1
Classical Active Proxy	75.5

TABLE I

Packet forwarding rates for various paths, measured in thousands of packets-per-second (Kpps).

The first three forwarding paths implement the standard IP forwarding function under three different scenarios. The **IP Fast Path** represents the minimal work a Scout-based router can do to forward IP packets. It is implemented by a single Scout module that is optimized for a specific source/sink device pair. The **General IP** path includes option processing and handling other exceptional cases. This path is constructed from three separate Scout modules: the input device driver, IP and the output device driver. Finally, the **Active IP** path implements IP as a user module, wrapped in the NodeOS environment. The difference between this and the previous path measures the overhead of the NodeOS module.

The next pair of paths implement a transparent UDP proxy. The first (**Transparent Kernel Proxy**) includes a null Scout proxy module, while the second (**Transparent Active Proxy**) includes a null proxy running on top of the NodeOS interface. In practice, such proxies might apply some transformation to a packet flow without the knowledge of the end points. Since the numbers reported are for null proxies, they are independent of any particular transformation.

The final two paths are for classical proxies, such as an FTP proxy that establishes an external TCP connection, receives a file name, and then establishes a internal TCP connection to the appropriate server. As before, we measure a null classical proxy running in both the kernel (**Classical Kernel Proxy**) and on top of the NodeOS interface (**Classical Active Proxy**)

Note that all of the numbers reported for the NodeOS assume the active code (user module) is written in C and compiled to the native architecture. It does not include the overhead of Java or some execution environment. We are currently porting the JVM and ANTS to the latest NodeOS interface, but based an implementation of the JVM and ANTS running on an earlier version of the interface, we expect an active Java module to add 3.5 μsec of processing time to each packet, and ANTS to add an additional 37.5 μsec to each packet. This would reduce the active IP forwarding rates, for example, to approximately 60.7k and 18.5k pps, respectively.

V. Implementation II: Janos

Our second example implementation is Janos, a "Java-oriented Active Network Operating System." Janos has two primary research emphases: (1) resource management and control, and (2) first class support for untrusted active applications written in Java. Toward these goals, Janos necessarily encompasses both the EE and NodeOS layers of the canonical active network architecture [2]. As Figure 9 shows, Janos is a layered architecture with three components: ANTSR, the Janos Virtual Machine (JanosVM), and Moab. Though this section is primarily concerned with Moab, the Janos NodeOS, we introduce the other layers to provide a bit of context. A complete discussion of the Janos architecture, focusing on other issues, appears elsewhere in this journal [27].

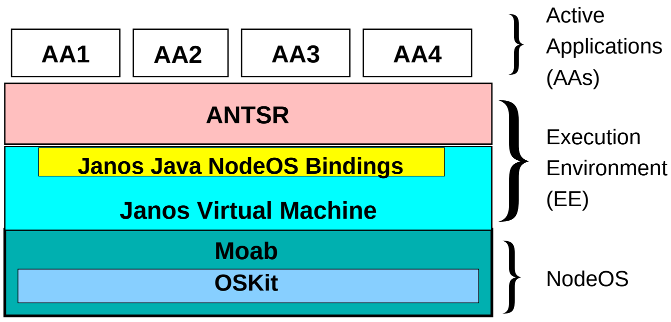

Fig. 9. The Janos Architecture and the corresponding DARPA Active Network Architecture.

Active applications for Janos are written to use the ANTSR Java runtime environment. ANTSR, or "ANTS with Resource management," exports to active applications essentially the same API as that exported by the standard ANTS active network runtime [30]. However, ANTSR is re-architected to take advantage of NodeOS and JanosVM services, and to support precise resource control. ANTSR runs atop the JanosVM, an extended Java Virtual Machine (JVM). In addition to providing the standard JVM services necessary for executing Java bytecode (e.g., threading, garbage collection, and JIT compilation), we have extended the JVM to support multiple namespaces and multiple heaps, providing independence between concurrently executing Java applications. The JanosVM also exports Java bindings for the NodeOS API, making it a potential host platform for other Java-based EEs. Together, ANTSR and the JanosVM form the EE layer of the active network architecture.

Underneath the JanosVM is Moab, a multi-threaded, fully-preemptible single-address-space operating system implementing the NodeOS abstractions. Moab is built using the OSKit [9], a toolkit of components for building systems software. The OSKit includes suites of device drivers, numerous filesystems, a networking stack, and a thread implementation, as well as a host of support code for booting, remote debugging, memory management, and enabling hosted execution on UNIX systems.

The remainder of this section discusses our experiences with the NodeOS interface in Janos. We start by describing the integration of the NodeOS interface into Janos from two perspectives. First, we look at the part of Janos above the interface boundary, discussing how the requirements of the JanosVM influenced the design of the interface and how we leverage the resulting design in the JanosVM implementation. Second, we look below the boundary, briefly describing some of the issues involved with implementing the NodeOS interface on top of the OSKit. We conclude by presenting and briefly discussing some preliminary performance results for Moab.

A. NodeOS Design Issues

The JanosVM has two key properties that influenced the design of the NodeOS specification: its JVM heritage which strongly influences memory management, and its tight coupling with the NodeOS. In the following paragraphs we explore these properties and how the NodeOS design supports them.

Memory management: Fundamentally, the JanosVM, like typical high-level language runtimes, needs to do its own memory management. Not only does the VM allow the sharing of code and data between domains, but it also maintains separate per-domain garbage-collected heaps. While it might be possible to use per-domain, NodeOS-enforced memory limits, at best these would be redundant mechanisms and at worst they would severely restrict how the JanosVM can use memory. Thus, the ability to aggregate memory from multiple domains into a single memory pool is essential to the JanosVM. This ability is enabled in the NodeOS interface by decoupling the domain and memory pool abstractions, allowing a memory pool to be specified explicitly at domain creation time. In the JanosVM, a single memory pool is created and all domains are associated with it. When a domain is created, its resources are added to this global pool. The JanosVM is responsible for making per-domain allocation decisions from this pool and enforcing per-domain memory limits. When a domain terminates, the memory pool's callback function is invoked, informing the JanosVM that it must return an amount of memory equal to that domain's share. The key point is that, with a single memory pool, the JanosVM is free to decide which memory pages are to be taken from the pool. Were there to be one memory pool per domain, the JanosVM would be forced to return the specific memory that was allocated to the terminated domain.

While the single memory pool enables the JanosVM to account for explicit allocations, one issue remains. In order for the JanosVM to accurately track all memory resources, it must also have a way to account for memory allocated within Moab on behalf of domains; that is, the implicitly allocated memory used for the internal state of NodeOS-provided objects. To accomplish this objective, the NodeOS interface was designed to provide "zero-malloc" object creation API calls in which the caller supplies pre-allocated memory for the NodeOS to use for the object state. Another advantage to this approach is that it allows the EE to embed actual NodeOS objects (as opposed to object references) in EE-provided objects, thus simplifying the code for managing these objects. The JanosVM exploits this feature by embedding NodeOS objects in the corresponding Java wrapper objects that are exposed to the ANTSR runtime via the Java NodeOS bindings. A final benefit of pre-allocation of object memory is that it makes object creation calls more predictable—they will never block for memory allocation or throw a memory exception.

Tight coupling: Another aspect of the JanosVM that had influence on the NodeOS interface is that the JanosVM and Moab are tightly coupled, with NodeOS API calls being implemented as direct function calls rather than as "system calls." The ability to share memory between the NodeOS and EE via direct pointers during object creation calls is an example of how the NodeOS interface is designed to efficiently support this model; that is, in an implementation where the NodeOS "trusts" an EE. The interfaces are designed to allow direct manipulation of all the NodeOS objects by the EE, making it possible, for example, to aggressively inline API calls in the EE code.

Another manifestation of the tight coupling of the EE and NodeOS is the way in which NodeOS exception conditions are handled. The NodeOS defers to the EE, via callbacks, whenever a domain terminates, faults, or exhausts some resource. The EE is expected to recover or clean up and destroy any affected state. Only if the EE doesn't respond in a timely manner will the NodeOS intervene, and then only in a very direct, albeit heavy-handed, way: by terminating the EE and reclaiming its resources. Graceful exception handling is the responsibility of the EE. This design is necessary because, as explained above under memory management, the EE has private knowledge of certain per-domain information.

Finally, the conscious effort to borrow from interfaces such as POSIX when designing the NodeOS interface is leveraged to great effect in the JanosVM. As the JanosVM is derived from Kaffe [31], a POSIX-hosted JVM, the similarity of the NodeOS thread, synchronization, and file interfaces to those in POSIX made porting the JanosVM to Moab much easier than it would otherwise have been.

B. NodeOS Implementation Issues

The OSKit was designed for building operating system kernels, making it an obvious choice for constructing the Moab NodeOS. The richness of the OSKit environment gave us not only the ability to run directly on two different hardware platforms (x86 and StrongARM) and on top of various UNIX-like OSes (Linux, FreeBSD and Solaris), but also provided development tools for debugging, profiling, and monitoring. Its support for standard interfaces such as POSIX threads and files made mapping the analogous NodeOS interfaces straightforward. On the other hand, not all OSKit interfaces were well suited to the task. The base memory interface was too low-level, its generic nature hindering precise control of memory resources. The networking interface was too high-level, its coarse granularity limiting the options for channel composition. In the following paragraphs we describe the Moab implementation of three NodeOS abstractions that best illustrate the good and bad characteristics of the OSKit as a NodeOS base.

Threads: The implementation of *thread pools* and the threading interface in Moab was, for the most part, straightforward due to the similarity between the NodeOS and POSIX (pthread) APIs. Just as this similarity helped "above" when mapping the JanosVM onto the API, it helps from "below" when implementing the API on top of the OSKit pthreads component. Most of the thread and synchronization primitives in the API mapped directly to pthread operations. There was one obvious performance problem caused by the direct mapping of NodeOS threads to pthreads: the NodeOS thread-per-packet model of execution led to creation and destruction of a pthread for every packet passing through the NodeOS. This was avoided by creating and maintaining a cache of active pthreads in every thread pool.

Memory: Memory pools represent one area where using the OSKit has made the implementation difficult. Tracking memory allocated and freed within OSKit components such as the network stack is easy, but identifying the correct memory pool to charge or credit for that memory is not. In particular, all allocations in OSKit components eventually funnel down to a single interface. By providing the implementation of that interface within Moab, we have control over all memory allocated anywhere in the system. However, that interface includes no specific information about what the memory is being used for, nor any explicit indication as to the principal involved. We are left with the choice of charging either the memory pool of the current thread (charge the "current user") or the "root" memory pool (charge "the system"). At the moment, Moab charges all OSKit allocations to the root pool. The solution we are pursuing is to evolve the OSKit memory interfaces, either by exposing more domain specific allocation interfaces or by passing down the necessary information to the generic interfaces.

Channels: Channels were by far the most challenging of the NodeOS abstractions to implement using current OSKit components. Anchored channels in Moab are implemented in one of two ways corresponding to the protocol specification used: raw interface ("if") channels deliver packets directly from the device driver to Moab (and on to the JanosVM) while all others use a socket interface to deliver UDP or TCP packets from the device driver, through the OSKit networking stack, and up to Moab. In both cases, the OSKit's encapsulated legacy-code components don't match well with the NodeOS networking model.

Raw interface channels are implemented directly above the OSKit's encapsulated Linux device drivers using the standard OSKit network device interface. We modified the OSKit device driver glue code to use specialized "packet memory" allocation routines to avoid the previously described problems caused by the low-level generic memory interfaces. Our only remaining concern is the inherent performance limitations caused by using stock Linux device drivers. This is discussed further in Section V-C.

All other *anchored channels* are implemented directly on a BSD socket-style OSKit interface, allowing UDP/IP or TCP/IP protocol specifications. This provides only a limited subset of what the NodeOS interface supports—in particular, it does not support "IP-only" channels. To address this drawback, we are reimplementing these channels using Click [15] routers. Click, a router component toolkit from MIT, provides a set of fine-grained *elements*— small components representing a unit of router processing— and a configuration language for combining these elements into *router configurations*. By using Click, we will be able to do the fine-grained protocol composition allowed by the NodeOS specification. Currently, we have a prototype implementation of Click-based UDP/IP inChans and outChans.

Cut-through channels are currently implemented as an unoptimized concatenation of NodeOS inChan/outChan pairs and can perform no additional protocol processing. Again, the coarse granularity of the OSKit networking interface does not allow access to individual protocols from within Moab. As with anchored channels, we have done preliminary work to make use of Click graphs to implement a more flexible form of cut-through channel. currently as an extension to the NodeOS interface. With Click cutChans, the protocol specification is a Click router description. The Click router elements used to instantiate the graph run inside Moab with standard device driver elements replaced by elements to read from an inChan and write to an outChan.

C. Performance

Our primary goals to date have been the implementation of the NodeOS abstractions and API in Moab and the integration of Moab with the JanosVM. With those goals largely met, we have now begun to look into measuring and improving the performance of Moab. In the following paragraphs we present the results of some simple packet-forwarding tests to characterize that performance.

All testing used the facilities of emulab.net, the Utah Network Emulation Testbed [8]. Testbed nodes are 600MHz PentiumIII PCs using ASUS P3B-F motherboards with 128MB of PC100 SDRAM and 5 Intel EtherExpress Pro/100+ 10/100Mbit PCI Ethernet cards all connected to a Cisco 6509 switch. Our experiment consisted of three nodes, connected in a linear arrangement: a packet producer node was connected by a private VLAN to the packet router node, which in turn was connected via a second private VLAN to a packet consumer node. The producer and consumer nodes ran custom OSKit-based kernels while the router node ran one of three routers as described below.

The test was to generate and send 18-byte UDP (64-byte Ethernet) packets at increasing rates to discover the maximum packet forwarding rate of the router node as measured at the consumer node. As in the Scout IP Fast Path experiment, only minimal IP processing was performed on the router node. The results are summarized in Table II.

Forwarding Path	Rate (Kpps)
OSKit	75.7
Moab cutChan	48.7
C-based EE	45.0

TABLE II
PACKET FORWARDING RATES AT VARIOUS LEVELS OF THE JANOS ARCHITECTURE, IN THOUSANDS OF PACKETS-PER-SECOND (KPPS).

The **OSKit** experiment establishes a performance baseline by measuring the raw packet forwarding rate of a simple OSKit-based router. This router has a single function which receives a packet pushed from the input interface driver, performs IP processing, and pushes the packet out on the output interface. This result represents the upper-bound on performance of a system based on OSKit interfaces built on top of stock, interrupt-driven Linux device drivers. Given the differences in hardware configurations, the recorded 75,700 packets-per-second (pps) is comparable to the 84,000 the Click team reported for similar stock Linux device drivers. The Click work[15] as well as the Scout team's experience also demonstrated the enormous improvement—exceeding a factor of three—in the forwarding rate of generic packets that *polled* device drivers provide. Based on those reports, converting Moab to use polled drivers should improve performance dramatically.

We then measured Moab using a cutChan to forward packets between the interfaces (**Moab** cutChan). The result was a 35% degradation of the OSKit forwarding rate, down to 48,700 pps. The bulk of the slowdown is attributable to the current unoptimized implementation of Moab cutChans. Since they are now implemented as a simple concatenation of an inChan and outChan, an actual Moab thread is dispatched for each arriving packet. This thread runs the cutChan function whose sole purpose is to send the packet on the outChan and release the packet buffer. Avoiding this scenerio is the purpose of cut-through channels, and we will optimize our implementation in the near future.

Finally, in **C-based EE**, we measured the performance of a C-language EE running on Moab using the NodeOS API. The EE consists of a domain with a single thread receiving packets on an inChan, performing the IP processing, and sending the packet on the outChan. This is exactly what the Moab cutChan forwarder does, only running outside the NodeOS API boundary. Hence, this test accurately demonstrates the overhead involved in crossing the API boundary. As one of the goals of Janos is the tight coupling of the EE and NodeOS, this result is important. In this configuration, there was an 8% drop to 45,000 pps. In absolute terms, the EE-level channel receive function took an average of 9,300 cycles (15.5 μs) per call versus 6,900 (11.5 μs) per call for the NodeOS-level function when receiving at a rate of 40,000 pps. Each EE-level invocation requires six API boundary crossings, for an average added cost of 400 cycles per crossing. This cost, which is somewhat high, will be reduced in the near future as we take further steps to optimize the implementation.

VI. Implementation III: AMP

Our third implementation, called AMP, is layered on top of the exokernel (xok) and its POSIX-like *libEXOS* library OS [13]. AMP's goal is to provide a secure platform upon which EEs and active applications can run, without unduly compromising efficiency. As illustrated in Figure 10, AMP consists of library code (libAMP), and four trusted servers, that jointly provide the NodeOS interface to an EE. One of our self-imposed design constraints was to avoid introducing new abstractions or mechanisms into xok, as we attempt to demonstrate that a secure system may be constructed entirely above an exokernel.

Comparing AMP to Janos, the same AAs and ANTSR EE are layered on the Janos Java NodeOS bindings. However, AMP implements its own subclasses of the NodeOS bindings, specialized to use the Java native method interface to make calls to the libAMP routines implemented in C. The Kaffe virtual machine has been ported to, but not specialized in any significant way for, the exokernel. LibAMP follows the exokernel library OS design approach of placing a copy of the OS in the same address space as the application. To provide protection of system-wide state information, portions of NodeOS abstractions are implemented within separate trusted servers, and libAMP invokes protected operations via cross-address space RPC. Trusted servers implement protected portions of these NodeOS abstractions: domains (Security Writer Daemon, SWTD), input channels (Dynamic Packet Filter Daemon, DPFD), output channels (Network Transmission Control Daemon, NTCD), and shared memory (Shared Memory Daemon, SHMD).

AMP shifts much of the protection burden away from the Janos VM and onto the trusted servers. There are two consequences of this design decision. First, AMP forgoes many of the opportunities for performance optimizations possible by exploiting a single-address space system. In particular, context switching costs related to RPC is a potentially significant performance bottleneck. Second, AMP is able to accomodate a wide range of EE implementations and languages. Because there is relatively little trust that must be placed in a given EE, AMP can limit the resources and operations that an EE is granted access to, thereby allowing more flexibility in configuring EEs to run within the system.

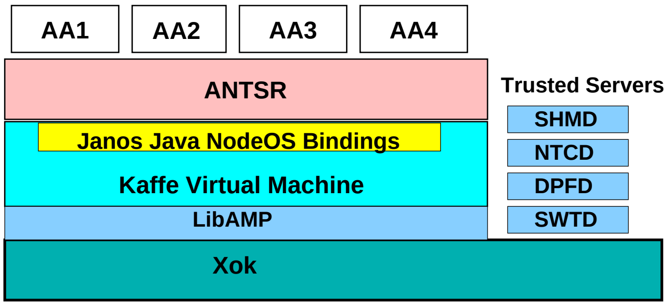

Fig. 10. The AMP Architecture.

A. Design Issues

The exokernel provides a minimal set of abstractions above the raw hardware. Ideally, only those mechanisms required to control access to physical resources and kernel abstractions are provided. All other OS abstractions are implemented in user space. Exokernel implementations utilize library operating systems co-located in the address space of each application, as opposed to protected OS servers executing in their own address spaces. This implementation choice reduces one of the well known problems of microkernel architectures, namely, the high direct and indirect costs of invoking services via RPC [18]. However, the exokernel also provides efficient support for RPC, which we have used extensively in our design and implementation.

AMP's influence on the NodeOS interface is reflected in the simplicity of the interface API with respect to security arguments. In fact, there is precisely one point, at domain creation, where credentials are passed to the NodeOS. These credentials represent the principal authorizing the domain's creation, and are used to determine limits on both resources and operations. By determining all rights for a domain exactly once, and at exactly one entry point in the NodeOS interface, this design facilitates the enforcement of security policies. AMP maps a domain's authorization to primitive protection mechanisms in the xok as described below.

The key to exokernel protection is the uniform support for hierarchically-named capabilities (CAPS). CAPS are more akin to an extensible Posix UID/GID mechanism than to capabilities, in that CAPS are checked against access control lists rather than naming and granting rights directly. Two properties of CAPS are essential to building a secure AMP system above xok:

- Kernel control over creation and use—xok maintains all CAPS in the system, controlling when new CAPS are created, associated with a process (environment abstraction in xok), and passed from one environment to another.
- Ubiquitous and exclusive use throughout the system call interface—every system call takes exactly one CAP as an argument to determine if the requesting entity has sufficient rights to perform the operation.

Together, these two properties provide the initial basis for assuring that the AMP system and its security mechanisms are tamperproof, non-bypassible, and intercept or enforce decisions on all requests for resources or services. By implementing the NodeOS abstractions and AMP security mechanisms above xok, development time is reduced, modularity is enhanced, and security requirements can be addressed in a straightforward manner.

B. NodeOS Interface and Trusted Servers

Each NodeOS resource abstraction must be controlled in order to ensure separation between the various domains instantiated in the system. The domain abstraction is the container that holds other resources, along with the credentials that authorize the domain's actions. Trusted servers provide control over the NodeOS resources by enforcing the current policy. These servers are rightly viewed as extended parts of the operating system implemented in user space. As such, they have powers granted to them at boot time as trusted software in order to carry out their function. They enforce access to their resources in essentially the same manner as the xok system call enforcement mechanism. An xok CAP is passed as an argument of each RPC to a trusted server, and used to check that the request is allowed. In explaining the implementation details of each node OS abstraction in AMP, we focus on the consequences of layering *above* xok.

Packet Forwarding: Unlike a typical monolithic kernel, xok does not include IP routing functionality in the kernel. Instead, a dedicated, non-active IP forwarder running in user space mimics the functionality provided within the kernel on other systems. One consequence is that IP forwarding is essentially identical to any other Active Network EE implementation running on the system. Our choice of a C implementation with support for a static IP forwarding function could easily be replaced by any EE ported to AMP, configured to run an AA that implements an appropriate IP forwarding function. However, this means that AMP can be expected to be somewhat slower forwarding IP packets than a corresponding Unix system, because of the additional costs of copying all packets up to user space and then back down for transmission.

Domains: The trusted server directly involved in managing domains, SWTD, interprets high-level policy, tracks domains and their associated credentials, and informs the other servers regarding what system resources are authorized to domains. Domains are established via an RPC to SWTD, which relies on a Credential Service (not described) to retrieve, validate and cache credentials. SWTD creates one xok CAP for each domain in the system. Since a CAP is passed on every system RPC to a trusted server, the CAP is actually used as an authenticated name for the domain. CAPs are never manipulated in user space, since they are passed by reference and maintained in the kernel. The credentials supplied with the domain create operation are used to determine the specific resources or services that the domain is granted access to. Our design calls for a flexible policy language, but our initial implementation hands out static policy mediation directives to each of the other trusted servers. For example, if a domain is created with the right to open certain input channels, then the policy mediation directives passed from SWTD to DPFD, which mediates inChan creation operations, would contain a canonical representation of the packet filter fields that must be specified by any inChan created by the domain.

There are several consequences to layering the domain abstraction above xok in this way. First and foremost, the kernel knows nothing about domains, but rather tracks the CAPS associated with domains. This means that other abstractions can be added to the domain container by implementing additional trusted servers, and informing them of the CAP and policy mediation directives associated with a domain. Since trusted servers are only special by dint of their possession of CAPS, and because they receive privileged communications from SWTD, it would not be difficult to extend the system. In fact, an EE or AA could play the role of trusted server with respect to control of an additional abstraction. Second, even though the domain hierarchy and CAP hierarchy were designed to be isomorphic, this property is not exploited at the trusted server level. The NodeOS interface allows parent domains to control their children domains; in theory, this could be implemented in AMP by having the parent use a CAP with the power of all its children's CAPs. Xok provides this exact functionality, but AMP can directly add the additional rights granted to each child domain to the parent's set of allowed operations at creation time.

Channels: The channel implementation in AMP is split across three address spaces: DPFD, the libAMP code colocated with the EE, and NTCD. DPFD enforces the policy regarding what packet filtering rules may be installed into the xok DPF mechanism by a domain, thereby guaranteeing strong separation of domains with respect to packets received over the network. NTCD plays a similiar, but not quite symmetric role for transmission of packets over output channels. The libAMP channel code does all the processing for ANEP, UDP and IP. (TCP is currently unsupported.) Output channels use a set of buffers mapped into the address space of NTCD to pass packets along for transmission, and NTCD selects the correct physical interface based on the destination IP address, and constructs the Ethernet header. NTCD can optionally enforce both transmission limits and header content controls on packets. Transmission limits on a domain can be enforced by directly controlling how many packets or bytes are sent per second.

The DPF mechanism is reused in NTCD to limit the values of header fields in transmitted packets. NTCD clones both the DPF implemented in xok, and the packet filter mediation function (from DPFD) used to control which filters are inserted into the DPF set. We observe that there is no need for a tight-coupling between the EE/libAMP channel implementation that establishes an outChan and the DPF rules used to control packet transmission. A small change to the NodeOS interface would allow other implementations, including ones in which trusted AAs separately supplied the filtering rules.

Threads: Our prototype uses the Kaffe implementation, for which we have developed a thread package called xok-jthreads. Xok only supports processes, while providing primitive mechanisms useful for implementing threads in user space. The xok-jthreads package is used by Kaffe for creating its own threads, as well as by the channel stack. Our prototype does not migrate a thread from the inChan into the EE, but rather delivers the packet and allows Kaffe to schedule one of its own threads to process the packet through the EE and AA. As threads are user space entities, we have designed (but not yet implemented) the machinery needed to limit CPU consumption by domains. In this design, a scheduler daemon acts as a first-level hierarchical scheduler, as well as controls all free scheduler timeslices (quantums). Xok provides an interface to control, allocate, and preempt quantums scheduled using a simple round-robin policy. The second-level scheduler inside each xok-jthreads implementation determines which thread inside the EE is run.

Files: CFFS is the native filesystem in AMP. Its operations are implemeted via a trusted server that is part of the original exokernel distribution. Our only design change is to control which portions of the file namespace are visible to the individual EEs. However, this is not adequate to assure the strong separation of different portions of the global filespace, since symlinks and shared inodes may obscure when sharing is taking place. The NodeOS sharing abstraction implemented via shm_open() and mmap() is intentionally restrictive in order to simplify the security aspects of controlling shared memory. The key restriction is that a shared region may only have a single writer. This obviates the need for controlling write access, implementing write locks, and reclaiming orphaned locks at domain termination. Moreover, it eliminates entirely the security policy and configuration that would be required to specify which entities had access to these ancillary operations. Instead, the shared memory daemon (SHMD) needs only check that read or write access for the shared region is permitted. Memory is provided by the writer, out of their mempool. Mempools currently exist only at the level of the entire EE—resource limits control the entire amount of memory used by the EE and all sub-domains.

C. Performance

We report on the forwarding rates for AMP at various layers in the architecture for minimum sized ethernet packets. Channelized IP corresponds to a C implementation of a forwarding process layered above the NodeOS channel abstraction. This process necessarily runs in user space, as the xok kernel does not directly implement forwarding. The limited performance reflects the costs of two copies and four CPU context switches required per each packet. The next two entries correspond to implementations that process ANEP packets encapsulated within UDP/IP. The minimal ANEP header does not carry any options, such as those requiring CPU intensive crytographic operations. The rough doubling of performance between the two cases reflects the benefit of a cutChan over a separate inChan/outChan pair anchored in a Java EE. The final entry measures the rate at which a minimal ANTSR capsule is forwarded. All numbers were measured on a testbed consisting of three 750MHz Pentium-III PCs with Intel EtherExpress Pro/100+ Mbit Ethernet cards connected to a Netgear FS516 switch.

Forwarding Path	Rate (Kpps)
Channelized IP	22.1
C channelized ANEP	17.5
Java channelized ANEP	6.9
ANTSR forwarder	1.2

TABLE III

Packet forwarding rates for various software layers, measured in thousands of packets-per-second (Kpps).

VII. Discussion

It is interesting to note how the differences between the three base systems impacted the way in which the domain abstraction was implemented. In Scout, the principal abstraction is the path, which essentially bundles a domain plus an inChan/outChan pair. In Moab, domains are closely associated with the JanosVM's abstractions for separate memory heaps and namespaces. Moab's support of threads as a first-class abstraction, coupled with the advantage of a single address space for memory, provide the right degree of support to allow the virtual machine to isolate and separately account for the resources used by active applications. In contrast to both of these, the exokernel allows AMP to map domains one-to-one with the fundamental protection mechanism of the system: hierarchically-named capabilities. This translates into the exokernel's notion of a domain as an owner of more primitive resources. A domain, in the eyes of exokernel, is roughly the resources it is permitted to allocate, and operations it is permitted to invoke.

Turning to the channel abstraction, the differences between the Scout and AMP implementations illustrate how underlying system structure impacts the design choices, and permeates the system in subtle ways. Scout's channel implementation consists of a number of system forwarding modules strung together into a protocol stack. AMP adopts a similar architecture. However, Scout implements inChan demultiplexing rules by distributing the pattern matching functions across the layers of the protocol stack, while AMP centralizes the demultiplexing function by constructing and downloading the pattern into the kernel. The Scout implementation allows individual system modules a great deal of flexibility, while the AMP implementation facilitates the imposition of higher-level security policy by checking inChan demultiplexing filters for conformance with the policy before they are downloaded into the kernel.

With the memory pool abstraction, the central implementation issue hinges on how to treat an EE's internal use of memory, versus the memory used by the NodeOS while performing an operation on behalf on a domain. Here, Moab assigns a single mempool to the entire JanosVM, and relies on the specific properties of that closely-coupled virtual machine to limit the memory used by individual domains within the EE. Most of the remaining work involves restructuring the memory allocation mechanisms below the Moab kernel interfaces to properly associate memory use with domains. AMP, on the other hand, has the goal of supporting EEs using different language technologies. Associating memory usage by specific domain is difficult, and requires modification of the EE implementations to create and manage mempools corresponding to separate virtual address spaces with both shared and private page ranges. As a simple step toward this goal, AMP includes a shared-memory abstraction that supports the inclusion of a set of physical pages into multiple virtual address spaces. Below the NodeOS interface, AMP has a relatively easy way to track memory. Because each domain has an assigned CAP, and every memory allocation operation requires that a CAP be provided, individual domain usage can be directly tracked.

VIII. Related Work

The first known active network, Softnet[33], implemented a programmable packet radio network in 1983. It built upon what one could call the first NodeOS/EE, a Forth environment called MFORTH [32]. This environment is consistent with the contemporary pattern of using special languages to program the network.

A more recent system, RCANE [19], defines a resource controlled framework for active network services. It supports the OCaml programming language [23], is implemented on the Nemesis operating system [17], and is interoperable with PLAN [11]. RCANE supports resource reservations for network bandwidth, memory use and computation, much like the NodeOS. The primary difference between RCANE and the NodeOS is the NodeOS's flexible communication abstraction. RCANE uses Nemesis's network, and allows only link layer communication, while the NodeOS allows any supported protocol to be used. (RCANE's link layer may be a virtual network implemented on top of UDP; nevertheless, RCANE does not allow the flexibility that the NodeOS provides.) Other differences include RCANE's reliance on a safe language to guarantee security.

Three recent router implementations—SuezOS [22], Click [15], and Router Plugins [7]—allow some degree of extensibility. In each system, router functionality can be extended by configuring safe extensions into the kernel. This is similar to the use of system modules to extend the forwarding paths in the Scout kernel. In contrast, the NodeOS separates the core OS from the EE, thereby allowing different EEs to safely implement different programming environments on the same router.

Bowman[20], which runs on top of Solaris, was the first NodeOS that targeted the same community-developed active network and NodeOS architectures that we target. Bowman was developed in the early days of the specification, and therefore implements a subset of the interface. It also does not provide resource controls since it runs on a generic Unix substrate.

IX. Conclusion

We have described an interface that allows active applications to access the resources available on an active router, and reported our experiences implementing the interface using three different operating systems. The interface is novel in how it is optimized to support packet forwarding, allows for fine-grain resource managment, and supports secure extensions. The three implementations not only demonstrate the feasibility of the interface, but perhaps more importantly, they also strongly influenced the design of the interface in the first place.

Acknowlegements

We are indebted to the many members of the active network community who contributed to the collaborative design effort that resulted in the DARPA active network architectural documents. We are grateful to the anonymous reviewers and our shepherd, David Wetherall, for their many helpful comments.

References

[1] Active Network NodeOS Working Group. NodeOS interface specification. Available as http://www.cs.princeton.edu/nsg/papers/nodeos.ps, January 2000.

[2] Active Network Working Group. Architectural framework for active networks, version 1.0. Available from http://www.darpa.mil/ito/research/anets/Arcdocs.html, July 1999.

[3] D. Scott Alexander, Marianne Shaw, Scott M. Nettles, and Jonathan M. Smith. Active bridging. In *Proceedings of the ACM SIGCOMM '97 Conference*, pages 101–111, September 1997.

[4] Gaurav Banga, Peter Druschel, and Jeffrey Mogul. Resource containers: A new facility for resource management in server systems. In *Proceedings of the 3rd Symp. on Operating System Design and Impl.*, pages 45–58, February 1999.

[5] Samrat Bhattacharjee, Ken Calvert, and Ellen Zegura. Congestion control and caching in CANES. In *ICC '98*, 1998.

[6] David Clark. The design philosophy of the DARPA Internet protocols. In *Proceedings of the SIGCOMM '88 Symposium*, pages 106–114, August 1988.

[7] Dan Decasper, Zubin Dittia, Guru Parulkar, and Bernhard Plattner. Router plugins: A software architecture for next generation routers. In *Proceedings of the ACM SIGCOMM '98 Conference*, pages 229–240, September 1998.

[8] Flux Research Group, University of Utah. University of Utah Network Testbed and Emulation Facility Web site. http://www.emulab.net/ and http://www.cs.utah.edu/flux/testbed/.

[9] Bryan Ford, Godmar Back, Greg Benson, Jay Lepreau, Albert Lin, and Olin Shivers. The Flux OSKit: A substrate for OS and language research. In *Proceedings of the 16th ACM Symp. on Operating Systems Principles*, pages 38–51, St. Malo, France, October 1997.

[10] John Hartman, Larry Peterson, Andy Bavier, Peter Bigot, Patrick Bridges, Brady Montz, Rob Piltz, Todd Proebsting, and Oliver Spatscheck. Experiences building a communication-oriented JavaOS. *Software—Practice & Experience*, 2000.

[11] Michael Hicks, Pankaj Kakkar, Jonathan T. Moore, Carl A. Gunter, and Scott Nettles. PLAN: A packet language for active networks. In *ICFP 98*, pages 86–93, September 1998.

[12] IEEE P1520 Working Group. IEEE P1520: Proposed IEEE standard for application programming interfaces for networks – web site. http://www.ieee-pin.org/.

[13] M. Frans Kaashoek, Dawson R. Engler, Gregory R. Ganger, Hector Briceno, Russell Hunt, David Mazieres, Thomas Pinckney, Robert Grimm, John Jannotti, and Kenneth Mackenzie. Application performance and flexibility on exokernel systems. In *Proceedings of the 16th ACM Symp. on Operating Systems Principles*, pages 52–65, St. Malo, France, October 1997.

[14] Scott Karlin and Larry Peterson. VERA: An extensible router architecture. In *IEEE OPENARCH 01*, Anchorage, AK, April 2001.

[15] Eddie Kohler, Robert Morris, Benjie Chen, John Jannotti, and M. Frans Kaashoek. The Click modular router. *ACM Transactions on Computer Systems*, 18(4), November 2000.

[16] Tal Lavian, Robert Jaegeer, and Jeffrey Hollingsworth. Open programmable architecture for java-enabled network devices. In *Proc. of the Seventh IEEE Workshop on Hot Interconnects*, Stanford University, CA, August 1999.

[17] I. M. Leslie, D. McAuley, R. J. Black, T. Roscoe, P. R. Barham, D. M. Evers, R. Fairbairns, and E. A. Hyden. The design and implementation of an operating system to support distributed multimedia applications. *IEEE Journal on Selected Areas in Communications*, 14(7):1280–1297, September 1996.

[18] Jochen Liedtke. Improving IPC by kernel design. In *Proceedings of the 14th ACM Symp. on Operating Systems Principles*, pages 175–187, Asheville, NC, December 1993.

[19] Paul Menage. RCANE: A Resource Controlled Framework for Active Network Services. In *Proceedings of the First Int. Working Conf. on Active Networks*, volume 1653 of *Lect. Notes in Comp. Sci.*, pages 25–36. Springer-Verlag, July 1999.

[20] S. Merugu, S. Bhattacharjee, E. Zegura, and K. Calvert. Bowman: A node OS for active networks. In *Proceedings of the 2000 IEEE INFOCOM*, Tel-Aviv, Israel, March 2000.

[21] David Mosberger and Larry L. Peterson. Making paths explicit in the Scout operating system. In *Proceedings of the 2nd Symp. on Operating System Design and Impl.*, pages 153–167, October 1996.

[22] Prashant Pradhan and Tzi-Cker Chiueh. Computation framework for an extensible network router: Design, implementation and evaluation. SUNY Stony Brook ECSL TR, 2000.

[23] Projet Cristal. The Caml language. URL: `http://pauillac.inria.fr/caml/index-eng.html`, 2000.

[24] Jonathan M. Smith, Kenneth L. Calvert, Sandra L. Murphy, Hilarie K. Orman, and Larry L. Peterson. Activating networks: A progress report. *IEEE Computer*, 32(4):32–41, April 1999.

[25] Oliver Spatscheck and Larry Peterson. Defending against denial of service attacks in Scout. In *Proceedings of the 3rd Symp. on Operating System Design and Impl.*, pages 59–72, February 1999.

[26] David Tennenhouse and David Wetherall. Towards an active network architecture. In *Multimedia Computing and Networking 96*, January 1996.

[27] Patrick Tullmann, Mike Hibler, and Jay Lepreau. Janos: A Java-oriented OS for active network nodes. In *IEEE Journal on Selected Areas in Communications, Active and Programmable Networks*, 2001.

[28] Carl A. Waldspurger. *Lottery and Stride Scheduling: Flexible Proportional-Share Resource Management*. PhD thesis, Massachusetts Institute of Technology, September 1995.

[29] David Wetherall. Active network vision and reality: lessons from a capsule-based system. In *Proceedings of the 17th Symp. on Operating Systems Principles*, pages 64–79, December 1999.

[30] David Wetherall, John Guttag, and David Tennenhouse. ANTS: A toolkit for building and dynamically deploying network protocols. In *IEEE OPENARCH 98*, San Francisco, CA, April 1998.

[31] Tim Wilkinson. Kaffe—a virtual machine to compile and interpret Java bytecodes. http://www.transvirtual.com/kaffe.html.

[32] J. Zander. MFORTH – programmer's manual. Technical Report LiTH-ISY-I-0660, Linkvping University, Dept of EE, April 1984.

[33] J. Zander and R. Forchheimer. Softnet – An approach to high level packet communication. In *Proc. Second ARRL Amateur Radio Computer Networking Conference (AMRAD)*, San Francisco, CA, March 1983.

[34] Lixia Zhang, Steve Deering, Debra Estrin, Scott Schenker, and D. Zappala. RSVP: A new resource reservation protocol. *IEEE Network*, 7(9):8–18, September 1993.

Larry Peterson (SM'95) received the B.S. degree in computer science from Kearney State College in 1979, and the M.S. and Ph.D. degrees in computer science from Purdue University in 1982 and 1985, respectively. He is a Professor of Computer Science at Princeton University. His research focuses on end-to-end issues related to computer networks, he has been involved in the design and implementation of x-kernel and Scout operating systems, and he is a co-author of the textbook *Computer Networks: A Systems Approach*. Dr. Peterson is the Editor-in-Chief of the *ACM Transactions on Computer Systems*, has been on the editorial boards for *IEEE/ACM Transactions on Networking* and the *IEEE Journal on Selected Areas in Communication*, and has served on program committees for SOSP, SIGCOMM, OSDI, and ASPLOS. He is also a member of the Internet's End-to-End research group, and a fellow of the ACM.

Yitzchak Gottlieb received a Sc.B. in Applied Mathematics–Computer Science from Brown University in 1998. He is currently a graduate student in Computer Science at Princeton University.

Mike Hibler received B.S. (1980) and M.S. (1983) degrees in computer science from New Mexico Tech. He is a research staff member with the Flux research group in the School of Computing at the University of Utah. His research interest is operating system design and implementation including virtual memory systems, network support and security. He was a major contributor to the original BSD project and has been involved with the design and implementation of the Mach4, Fluke and Moab research operating systems. He is arguably the best known computer science researcher from Truth or Consequences, New Mexico, lives in Utah despite the snow, and is an avid mountain biker.

Patrick Tullmann received the B.S. degree in computer science from the University of Vermont in 1995 and the M.S. degree in computer science from the University of Utah in 1999. He is a research associate in the Flux research group in the School of Computing at the University of Utah. His research interests lie at the intersection of operating systems and high-level languages, and he has worked on the Fluke, Alta, and Moab operating systems in the Flux research group. He is a member of the Usenix Association.

Jay Lepreau heads the Flux Research Group at the University of Utah's School of Computing. He has interests in many areas of software systems, most of them originating in operating systems issues, although many go far afield. Those disparate areas include information and resource security, networking, programming and non-traditional languages, compilers, component-based systems, and even a pinch of software engineering and formal methods. In 1994 he founded the highly successful and prestigious OSDI conference series, one of the two premier OS conferences. His current service efforts are focused on developing a large-scale, reconfigurable, network emulation testbed that is universally available to remote researchers.

Stephen Schwab received a B.S. degree in EECS from U.C. Berkeley in 1987, and a M.S. degree in computer science from Carnegie Mellon University in 1990. He is a senior research scientist at NAI Labs, the security research division of Network Associates, Inc., where he manages projects investigating high-speed firewall technology and active networking. He has been involved in the development and application of technology in the areas of operating systems, security, high-performance networking and parallel computing, and is a member of the ACM.

Hrishikesh Dandekar received the B.S degree in computer engineering from the University of Pune, India in 1996 and the M.S. in computer science from the University of Southern California in 1998. He is currently a research scientist at NAI Labs, the security research division of Network Associates, Inc., where he has been involved in implementing the NodeOS interface for active routers using the exokernel system. His research interests are in the areas of computer networking and operating systems.

Andrew Purtell is a research engineer at NAI Labs, the security research division of Network Associates, Inc., where he has developed high-speed firewall and active network prototypes. His research interests lie at the intersection of programming language technology and embedded operating systems. He is a member of the ACM.

John Hartman (M'95) received the Sc.B. degree in computer science from Brown University in 1987, and the M.S. and Ph.D. degrees in computer science from the University of California, Berkeley, in 1990 and 1994, respectively. He has been an Assistant Professor in the Department of Computer Science, University of Arizona since 1995. His research interests include distributed systems, operating systems, and file systems. Dr. Hartman is a member of the ACM.