Snow on Silk: A NodeOS in the Linux Kernel [*]

Nadia Shalaby Yitzchak Gottlieb Mike Wawrzoniak
Larry Peterson
{nadia,zuki,mhw,llp}@cs.princeton.edu

Department of Computer Science
Princeton University

June 17, 2002

Abstract

Transferring active networking technology from the research arena to everyday deployment on desktop and edge router nodes, requires a NodeOS design that simultaneously meets three goals: (1) be embedded within a wide-spread, open source operating system; (2) allow non-active applications and regular operating system operation to proceed in a regular manner, unhindered by the active networking component; (3) offer performance competitive with that of networking stacks of general purpose operating systems. Previous NodeOS systems, Bowman, Janos, AMP and Scout, only partially addressed these goals. Our contribution lies in the design and implementation of such a system, a NodeOS within the Linux kernel, and the demonstration of competitive performance for medium and larger packet sizes. We also illustrate how such a design easily renders to the deployment of other networking architectures, such as peer–to–peer networks and extensible routers.

1 Introduction

A general architecture for active networks has evolved over the last few years [7, 26]. This architecture stipulates a three-layer stack on each active node, depicted in Figure 1. At the lowest layer, 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 layer, one or more execution environments (EE) define a particular programming model for writing active applications. To date, several EEs have been defined, including ANTS [29,30], PLAN [2,11], SNAP [16], CANES [6] and ASP [5]. At the topmost layer are the active applications (AA) themselves. As evident from Figure 1, although more cumbersome for the user, an AA may access the

[*]This work supported in part by DARPA contract N66001–96–8518, NSF grant ANI-9906704, and Intel Corporation.

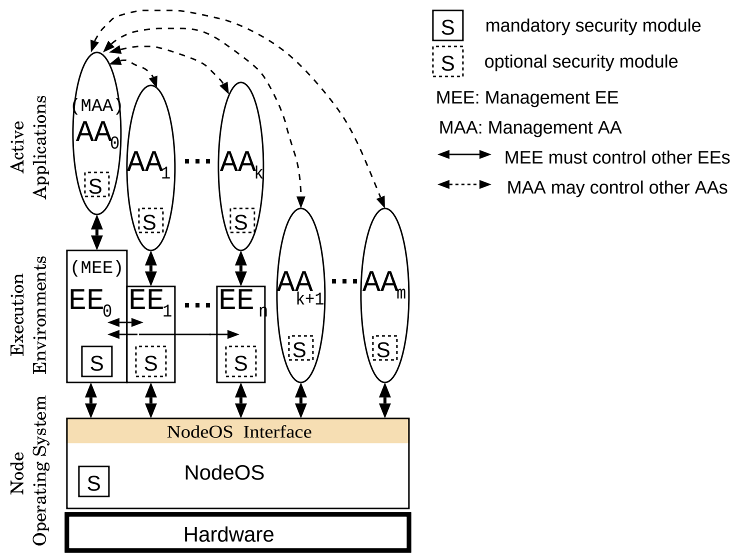


Figure 1: The architecture of an active node: a Node Operating System above the Hardware exports a NodeOS Interface; Execution Environments define a programming model for Active Applications. EEs and AAs access the operating systems via the NodeOS Interface. With respect to the network as a whole, a Management AA and a Management EE are the ones bootstrapped at systems initialization, and manage AAs and EEs respectively. The NodeOS and the Management EE necessarily maintain security on the node, while the Management AA as well as EEs and AAs may implement their own security modules as an option.

NodeOS directly, forgoing EEs. From the application level perspective, such AAs may also be regarded as single operation EEs.

In the realm of an active network, security risks are heightened at the level of end-to-end active users, the active router node itself, the EEs and the active code that traverses the network, thereby necessitating a security component at each of these layers [1, 18]. Within an active node, the NodeOS necessarily maintains security on the OS level, depicted as a mandatory security module. EEs and AAs may also choose to maintain their own security mechanisms, depicted as optional security modules.

Lastly, network control, monitoring and management is maintained by a Management EE (MEE), the first EE started at system initialization, which necessarily loads, manages and coordinates between subsequent EEs in the system, and maintains a mandatory security module. SENCOMM [12] is a comprehensive example of such a MEE. It is also possible for a Management AA (MAA) to exist, to similarly manage and coordinate between the AAs running on the node. If such a MAA acts with a role, or forgoes EEs to control AAs directly accessing the NodeOS, it would also maintain its own security module.

In order to transfer this powerful and flexible architecture from the research arena to wide scale everyday deployment, we need to design a NodeOS whose active networking functionality can be latent within a general purpose OS but, when necessary, implements this richer functionality without compromising performance. Therefore, our NodeOS design should simultaneously meet three goals: (1) be embedded within a wide-spread, open source operating system; (2) allow non-active applications and regular operating system operation to proceed in a regular manner, unhindered by the active networking component; (3) offer performance competitive with that of networking stacks of general purpose operating systems.

Previous NodeOS systems Bowman [15], Janos [27], AMP [8] and Scout [19], only partially addressed these goals. Our central contribution lies in the design and implementation of such a system, a NodeOS within the Linux kernel, and the demonstration of how we maintain performance comparable to Linux for routing on behalf of active applications. Besides contributing to decoupling the NodeOS specification from EEs/AAs from above, and the NodeOS implementation from below, we deliver a system with distinctly established protection boundaries between the AA/EE and NodeOS space, in terms of memory management, process scheduling and security. The benefits of our design are furthermore strengthened by the fact that it offers a straightforward mapping to other networking architectures, such as peer–to–peer networks and extensible routers, which would in turn lead to their deployment.

The remainder of this paper is organized as follows. The next section describes the design rationale and the recent key evolution of the NodeOS interface. Section 3 presents our architecture, which consists of Scout paths, their embedding into the Linux kernel as Silk, connecting to devices via the partner interface, and the functionality of two key device drivers communicating via the partner interface: Geni, a driver for Scout paths, and Sockets, a driver for user space. Section 4 describes the NodeOS implementation, how Snow and Sockets were used to interface to NodeOS API and user space respectively, and a brief description of the structure of the NodeOS module in Silk. We evaluate our design and implementation in Section 5, and demonstrate how it can be extended to conform to other networking paradigms in Section 6. Section 7 summarizes the related work to date and Section 8 concludes the paper.

2 NodeOS Interface

While each EE exports its own interface to the AAs, establishing a unified interface to the NodeOS is an essential element to the Active Networks architecture. We have contributed to the design and recent evolution of the NodeOS API specification [3], which has markedly influenced our NodeOS architecture and implementation. We briefly describe the primary abstractions of the NodeOS interface, outline the two kinds of data structures used, and highlight the recent evolution of the interface, which rendered our architectural design possible — in terms of the nature of data structure definitions, the arguments to the API function calls, and the two models for memory allocation and thread management.

2.1 Abstractions

We define four primary abstractions: thread pools, memory pools, channels and files, encapsulating the node's computation, memory, communication and persistent storage, respectively; and a fifth abstraction, the domain, aggregates control and scheduling for the other four abstractions. Semantically, however, a particular domain does not necessarily subsume the instantiations of each of the other abstractions, or all of their components. Rather, its relationship to the other abstractions is summarized in Table 1, where we observe many-to-one, one-to-one and one-to-many mappings.

Domain	$\xrightarrow{M:1}$	Memory Pool
Domain	$\xrightarrow{1:1}$	Thread Pool
Domain	$\xrightarrow{1:M}$	Channel
Domain	$\xrightarrow{M:1}$	File Name Space

Table 1: Relationship of Domains to the Memory, Thread, Channel and Persistent Storage Abstractions

Channels are further characterized as either *incoming*, *inChan*s, characterized by a demultiplexing key specifying the chain of protocols, a buffer pool for packet queueing, and a function to handle the packets; or *outgoing*, *outChan*s, characterized by a processing key specifying a chain of protocols and the link bandwidth it's allowed. Moreover, *cut-through* channels, *cutChan*s, both receive and transmit packets, and are characterized by the *inChan/outChan* pair since they can also be constructed by concatenating the two. Cut-through channels are primarily motivated by the desire to allow the NodeOS to forward packets without EE or AA involvement.

This abstraction set allows us 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 demultiplexing key, while the function that is to be applied to the flow is specified with a combination of module names and the handler function.

Other, non-primary abstractions include *events* which could be scheduled by a domain, a *heap* for memory management, *packets* to encapsulate the data that traverses a channel and the notion of *time* which NodeOS provides to EEs and AAs.

2.2 Objects and Specifications

The two major data structures of the API, are *specifications*, with visible, well-defined fields, and *objects*, which are opaque structures. The most important distinction being that, from the perspective of the NodeOS, the lifetime of a specification is a single API call, while the lifetime of an object is from its explicit creation via an API "create" call, until its explicit destruction, via an API "destroy" call.

For example, when creating a domain object via **an_domainCreate**, its an_Domain argument is an opaque object, whose memory cannot be reused until the domain is destroyed; whereas its an_ThreadPoolSpec argument is a specification, whose memory, can be reused immediately after the **an_domainCreate** function returns.

As a result of these semantics, objects can be subcomponents of a specification, but not vice versa. From the EE's perspective, it must ensure that the specification and any storage it references is not mutated or reclaimed until the NodeOS API function returns. This enables the EE to pass the same specification to multiple APIs calls simultaneously.

From the perspective of the NodeOS, when a reference to a specification is passed as a parameter to a NodeOS API function, the NodeOS may freely access the specification and any storage it references, however it cannot modify it in any way. This guarantees that when the NodeOS API function returns, the EE will be able to locate pointers to storage that it placed into the specification.

We have intentionally defined both specifications and objects as pointer types in a uniform way. The distinction lies not in *how* they are defined, but rather in *where* they are defined. Specification structures need to be defined in the EE, and are therefore explicitly defined in the NodeOS API document [3]. Object structures, on the other hand, are defined in the NodeOS, since they need to remain implementation dependent, and as such, opaque to the API.

2.3 Arguments to API Calls

Arguments to API call functions can be standard C types, such as void *, char *, pointers to functions, such as void *(*f)*(void * *arg*), objects, such as an_Domain, or specifications, such as an_ThreadPoolSpec.

Our decision to define both object and specification types as pointers allow us to pass them both by reference, as pointer type arguments. However, in the case of objects, the EE declares a pointer type of the form

```
typedef struct an_GenericObject * an_GenericTypeObject
```

in order to be able to make API calls with objects as arguments.

This recent change in the NodeOS API design provides three advantages. First, it allows for a set of portable header files, shared among various EEs. Second, this preserves the opaqueness of the objects from the EE's standpoint, thus allowing for different implementations of the NodeOS. And finally, as certain object types, such as the security credentials, become better understood, they could be migrated into specifications without changing the API [1], thus preserving backward compatibility with all existing EEs and AAs.

[1]It would be a simple matter of adding their structure definitions to the EEs' common header files.

2.4 Memory and Thread Management

Another key change was introducing *implicit* and *explicit* models for memory allocation and thread management. The former, and predictably more common case, delegates both to the NodeOS. The implicitly allocated objects will be automatically freed by the NodeOS when the corresponding destroy call is made, and all associated threads are also automatically destroyed.

The second, explicit model, serves the so called "trusted EEs", where the EEs and NodeOS are tightly coupled, sharing security and protection boundaries. In this case, the EE may wish to explicitly manage the memory used by the NodeOS for EE–created objects, and allocate its threads. It thus passes an appropriately–sized chunk of memory to all object create calls for the NodeOS to use. When such an explicitly allocated object is destroyed, the NodeOS tears down any internal state associated with the object and stored in the object memory, but does not free the memory.

This approach to the NodeOS specification allows for both trusted and untrusted EE implementations and gives the desired freedom in NodeOS design.

2.5 An API Example

The NodeOS API is specified via a portable set of header files, in C or in Java, consisting of type definitions for the objects, specifications and its API functions. For example, the header file includes the following definition for incoming channel creation

an_InChan **an_inchanCreate(**void * *mem*, an_Domain *d*, an_DemuxKey *dmxKey*, char *protspec*, char *addrspec*, an_NetSpec *netspec*, an_ChanRecvFunc *deliverfunc*, void * *deliverarg* **)**;

as well as for the an_NetSpec specification

```
typedef struct an_NetSpec {
        int             maxthreads;
        unsigned int    bandwidth;
        int     npbufs;
        an_PacketBuffer * pbufs;
} * an_NetSpec_t;
```

Additionally, EEs (or AAs directly) declare the opaque objects in a separate, non-portable header file, since they are only fully visible within the NodeOS, such as

```
typedef struct an_Domain * an_Domain_t;
```

3 Architecture

Our philosophy is that the provision of a modular networking tool-kit, coupled with good interface design to different system components, results in easy and efficient development of various networking architectures and services. The development essentially becomes a straightforward translation of one interface to another.

At the core of our architecture, is the observation that Scout paths [17] closely resemble the encapsulation of the primary abstractions of the NodeOS Interface. This observation has inspired the earlier work of implementing a NodeOS within the Scout stand–alone OS [19]. Moreover, architecting NodeOS around Scout paths results in employing the same mechanism for both the traditional and active forwarding services, thereby meeting our design goal of integrating the NodeOS interface in a manner that does not negatively impact our ability to forward non-active packets.

Unlike the Scout NodeOS implementation [19] however, and in line with our goal of employing a general purpose and wide spread OS, we embed the Scout path abstraction into Linux, known as "Scout in Linux Kernel" (Silk). This necessarily establishes user/kernel space boundaries, thus requiring a mechanism to communicate between them, the Scout paths, and other I/O devices.

In what follows, we explain the architecture of each these components and how they fit together to provide a NodeOS that meets our stated goals.

3.1 NodeOS Abstractions to Scout Paths

Designed to encapsulate I/O flows, a *Scout path* [17] is a structured system activity that provides an explicit abstraction for early demultiplexing, early packet dropping when flow queues are full, resource accounting per flow, explicit flow scheduling, as well as extensibility in adding new protocols and constructing new network services. Each Scout path consists of a string of protocol modules, and encapsulates a particular flow of data, such as a single TCP connection.

We can therefore characterize a path by its sequence of modules, a demultiplexing key, and the resource limits placed on the path (such as queue lengths, CPU cycle share, bandwidth). As such, this closely maps to the information 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, we are able to design a NodeOS module to perform a simple mapping between Scout path operations to domain, channel, thread, and memory operations.

To demonstrate this concept, consider Figure 2, which portrays four examples of Scout paths. Both ends of a path are delineated by the Geni (Generic Interface) module, which abstracts the device and communication interfaces to the path framework, performs the demultiplexing, and maintains input and output queues at each end.

The first, non-NodeOS path, corresponds to a single TCP connection going through Ethernet, IP and TCP processing, respectively. From the bottom, packets arrive via the network interface, while standard user space applications connect to this path via the socket interface at the top. Transforming this scenario to one where, say UDP, packets are controlled via the NodeOS Interface,

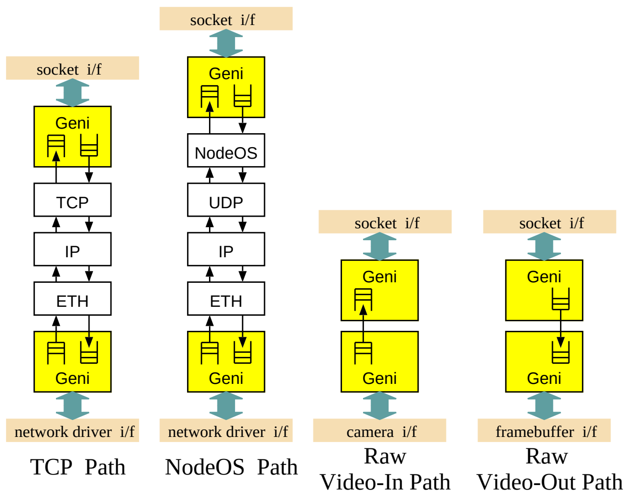


Figure 2: Four Scout paths interfacing with user space via Sockets: a TCP path, a NodeOS path via UDP and two raw video paths, from a camera and onto a framebuffer

we simply chain a NodeOS module that performs the path–to–NodeOS–Interface functionality mapping above the UDP module. The result is the NodeOS path shown.

We can also construct *raw paths*, as depicted in Figure 2, where no protocol modules are chained to the path, resulting in very fast data forwarding. For example, in one case, packets are generated via a video camera, arrive on the camera interface, and are sent to the application via the socket interface. Geni abstracts the interfaces on each end, and maintains queues if buffering is needed. Likewise, the other raw path portrays an application that sends packets via the socket interface to be displayed via the framebuffer interface on video screen.

3.2 Silk

Conforming with our first design goal of using an established, open source OS, we chose to design our NodeOS within the Linux kernel, via Silk [4], which encapsulates the Scout path architecture into a loadable Linux module. In order to meet our second design goal, requiring that we preserve the ability to download and run standard non-active applications, Silk provides its own path scheduler and thread package that coexists with the Linux CPU scheduler.

The major advantage of permitting the coexistence of two OS schedulers, is that it enables the scheduler that is virtually in control, in our case, Silk, to not only decide what share of the CPU to assign to Linux, but to also maintain its own suite of schedulers for scheduling Scout paths, such as fixed priority, Earliest Deadline First (EDF), Weighted Fair Queueing (WFQ), and Best Effort Real Time (BERT). By choosing an appropriate class of schedulers, such as fixed priority, we can implement elaborate quality of service (QoS) processing via the NodeOS interface, (such as providing packet streams with guarantees on the system's resources, for example CPU, bandwidth, memory pools, threads and queues), while still servicing best effort packets at line speeds [20].

3.3 Partner Interface

Stand-alone Scout must include a separate driver for every device it uses. Silk on the other hand, in theory, has access to a plethora of Linux drivers. In practice, however, since Silk is really an embedded OS in its own right, a separate Scout module would be necessary to interface to every such driver. As the number of these devices increases, such a design becomes neither efficient nor scalable.

Instead, we designed a uniform and symmetric interface for intelligent devices, called the *partner interface* [28], applicable to all the heterogeneous devices within an OS. The key idea is that rather than rewrite the Silk drivers for all the Linux interfaces to file systems, devices (such as network cards), and various user level APIs (such as sockets), thus effectively redoing all the work done within the general OS for this specific embedded module, we unify all the drivers into the *partner interface*, where they all exchange a common data structure, the *partner vector*, and assume a symmetric *partner modus operandis* after OS initialization, as portrayed in Figure 3.

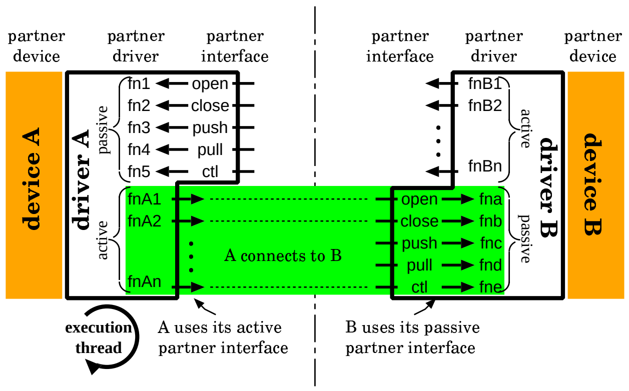


Figure 3: The execution thread is at partner driver A. Therefore, partner driver A executes its active partner interface, threreby invoking the passive partner operations of device B.

The partner interface consists of five operations: *open*, *close*, *push*, *pull* and *control*; for opening and closing a device driver, sending and receiving data to and from the driver, and controlling the data flow, respectively. Partner operations exchange an *aggregate* internal data structure, the partner vector, or *pvec*, which is designed to simultaneously be an aggregate of one or more of aggregate internal representations of any device driver (such as *iovec*s, *sk_buff*s, *frame_buffers*, etc.), and enable data manipulation among different device drivers without memory copying.

Each device driver must provide its side of the partner interface, thus translating its internal functionality to the partner operations. This allows communication flows established within a system to uniformly access the networking stack (e.g. via the Geni driver in the case of Scout paths), where data is forwarded or modified. Alternatively, if neither buffering nor additional processing is required, raw data can flow directly between a network interface card and an application via sockets due to the common partner interface.

Two particular partner drivers in Silk, Geni for Scout paths and Sockets for user space, are of special interest to the design of our NodeOS, and are therefore discussed in more detail below.

3.4 Geni: Scout Parnter Driver

From the partner interface standpoint within Silk, Scout paths are viewed as a device, and Geni, the driver to that device. Geni is conceptually architected to perform three major functions: (1) implement the partner operations, thereby translating partner operations to operations on Scout paths, such as pathCreate, pathDelete, demux, deliver and *drop*, and vice versa, respectively; (2) demultiplex the incoming packets, thus mapping network flows onto Scout paths (which the NodeOS modules maps to channels), and abstracting their forwarding functionality and (3) multiplex packets from the end of a particular path onto their target output devices, such as a network card or a socket.

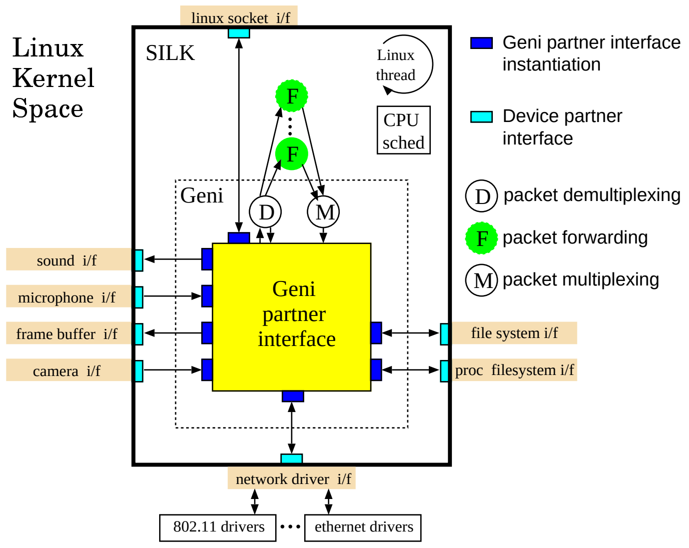


Figure 4: Scout's partner interface, Geni, connects Scout paths to network and media devices, file systems and socket interfaces

A schematic picture of Geni within Silk is given in Figure 4, illustrating how Geni interfaces Scout paths to interact to the other drivers via the partner interface. Portrayed are the partner drivers of each interface from Geni's side and the other device's side, packet multiplexing and demultiplexing. The examples shown are network devices, file systems, video and audio devices, as well as the drivers that bridge the Linux kernel and user space, sockets.

Within Geni, the partner operations *open, close, push* and *pull* are roughly mapped to the Scout operations *pathCreate, pathDelete, deliver* and *dequeue*, respectively. As for the partner *ctl* operation, Geni translates it to the corresponding control operation on either a path, a particular Scout module, or on the Silk OS itself. Consequently, for active packets, the *ctl* translates to an operation on the NodeOS module.

Device partner interfaces are depicted as the small rectangles in Figure 4. Each such dark rectangle conceptually represents an instantiation of Geni partnered with a particular device driver. To illustrate this point, consider again the Scout paths depicted in Figure 2. For the TCP and NodeOS paths, the bottom and top Geni modules are instantiations of Geni partnering with the network and socket partners, respectively. In the case of raw paths, the bottom and top modules are instantiations of Geni's partnering with the video and socket partners, respectively.

3.5 Sockets: User Space Partner Driver

The standard UNIX socket interface bridges user space applications with kernel networking stacks. In our architecture, Silk hijacks the kernel networking stack from Linux, while still employing the socket driver (via the partner interface) to communicate with user space.

Like any partner driver, the socket partner driver translates the socket API functions, such as *socket, bind, listen, connect, read, write, select* and *ioctl*, to the five partner interface operations, and vice versa, also transforming the data from *iovec*s to *pvec*s and back. In compliance with our design goal of allowing non-active applications to run unhindered by the NodeOS functionality, the socket partner intercepts all Linux application calls that use standard networking protocols via the socket API, such as calls on the *PF_INET* family, and passes their processing on to Silk. This allows unmodified legacy applications to access TCP and UDP paths transparently. For a detailed description of the socket partner driver, the reader is referred to [28].

To handle active applications on the other hand, Silk provides a new, *PF_SCOUT* protocol family, which is registered with Linux at system initialization. The socket partner interfaces to Scout paths via Geni, as illustrated in Figure 2. Like the *PF_INET* family, the *PF_SCOUT* protocol family has to be exposed to user space, by the socket driver. It is noteworthy that via this mechanism, our design extends to experimental and non-standard protocols beyond active networking. All that is needed is implementing the corresponding protocol as a module chained into a Scout path.

4 Implementation

We assembled all the components of our architecture to implement the NodeOS functionality within Silk, and expose the NodeOS API to the EEs and AAs in Linux user space. The overall implementation in its entirety is depicted in Figure 5. In line with our design philosophy, the underlying mechanism of our NodeOS design is to provide a mapping from the existing system components and interfaces, to ones we architected and implemented so as to provide the active networking functionality. Specifically, a mapping from a user space AA or EE to an active path and vice versa occurs as follows:

EE/AA		Snow		Linux kernel		Socket/Geni partner drivers		Silk
NodeOS API	⟷	Socket API	⟷	Linux Socket i/f	⟷	Partner Interface	⟷	Scout Path Operations

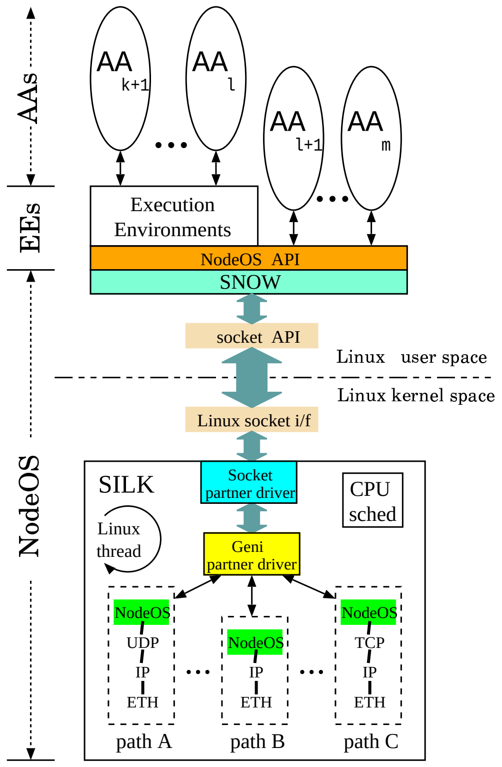


Figure 5: Overall NodeOS implementation in Silk: NodeOS API is provided for user space; Snow translates it to the socket API calls; the Socket partner driver translates that to the partner operations, which communicate with Geni; Geni sets up and drives the active path in Silk, which includes a NodeOS module.

To further elaborate on the implementation of the various components of our functional mapping in user space, Linux kernel to partner devices, and within a Scout path, we walk through an example of the creation of an incoming channel.

To create an incoming channel, an EE or AA calls the function

an_inchanCreate(*mem, domain, dmxKey, protspec, addrspec, netspec, deliverfunc, deliverarg* **)**;

Since our architecture stipulates using the UNIX Socket API to cross the user/kernel boundary in Linux (making it consistent with non-NodeOS application calls), we implement a set of Silk NodeOS Wrappers (Snow) to convert the AA/EE calls from the NodeOS to the Socket APIs, depicted in Figure 5. Snow translates all NodeOS calls into *ioctl* calls on a socket file descriptor, of type *PF_SCOUT* family, with the corresponding NodeOS API function and its arguments. Continuing with our example, to create a channel, Snow would invoke

fd = socket (*PF_SCOUT, NOT_USED, pathTemplateNo*);

ioctl(*fd, NODEOS_INCHAN_CREATE, argListPtr* **)**;

Linux then converts these *socket/ioctl* calls at the user level to a sequence of kernel level calls. Namely, the *socket* call becomes a *sys_socket* call, which invokes *sock_create* to allocate memory for the socket structure, and subsequently invoking the *create* call, as follows:

sys_socket(*PF_SCOUT, NOT_USED, pathTemplateNo* **)**;

sock_create(*PF_SCOUT, NOT_USED, pathTemplateNo, sock_str* **)**;

create(*sock_str, pathTemplateNo* **)**;

The corresponding user space *ioctl* call, on the other hand, translates to the invocation of the following two kernel function calls in sequence

sys_ioctl(*fd, NODEOS_INCHAN_CREATE, argListPtr* **)**;

ioctl(*inode, file, NODEOS_INCHAN_CREATE, argListPtr* **)**;

After the *socket/ioctl* calls cross the user/kernel boundary in Linux, our socket partner driver needs to convert the respective Linux kernel function calls to the partner interface, so that it can then communicate with the corresponding Scout path via Geni. The socket *create* call is implemented by setting up a Silk socket structure, with corresponding operations and state. On the other hand, all *ioctl* calls are translated to the partner *ctl* operation. In our example, the socket partner driver invokes the following control operation on Geni as its partner

ctl(*Geni, NULL, NODEOS_INCHAN_CREATE, argListPtr, sizeof(argList)* **)**;

For every control operation invoked on the Geni partner driver, Geni decides whether it is a control operation on a particular path, a module or on the Silk OS itself. In our case, it always translates to control operations on the NodeOS module. Specifically, Geni converts the above call to

ctl(*NodeOS, INCHAN_CREATE, argListPtr, sizeof(argList)* **)**;

Once a call reaches the NodeOS module, it translates the NodeOS API functionality of setting up channels, maintaining domains, thread and memory pools into their corresponding operations on Scout paths. Thus, in our example, finally, the NodeOS module invokes *pathCreate* with the appropriate arguments to setup the Scout path.

5 Evaluation

Our experiments were conducted on 1.5GHz Pentium-4 machines, with a 256K cache, and 512MB RAM, connected to a 100Mbps network link via one to four 100Mbps Tulip cards. Each data point represents the average obtained for 10,000 packets, on three distinct runs.

To evaluate the robustness of our NodeOS router, we generated an increasing amount of packets on the two 100Mbps input ports, causing the router to forward to the other two 100Mbps output ports, to avoid input/output port contention. We were able to sustain line speed throughputs for packet sizes from 1 to 1400 bytes of payload, for three types of forwarding paths; (1) IP cut-through paths, representing the minimal router resources consumed to forward IP packets; (2) UDP cut-through paths, similar to null proxies, which forward UDP packets through Silk within the kernel; and (3) UDP active paths, where UDP packets cross into user space, and are processed via the NodeOS module in Silk. In other words, our router implementation coupled with the prototype hardware cannot be saturated with 200Mbps link speeds. This illustrates the robustness of our router and gives us ample resources for the additional functionality that we seek.

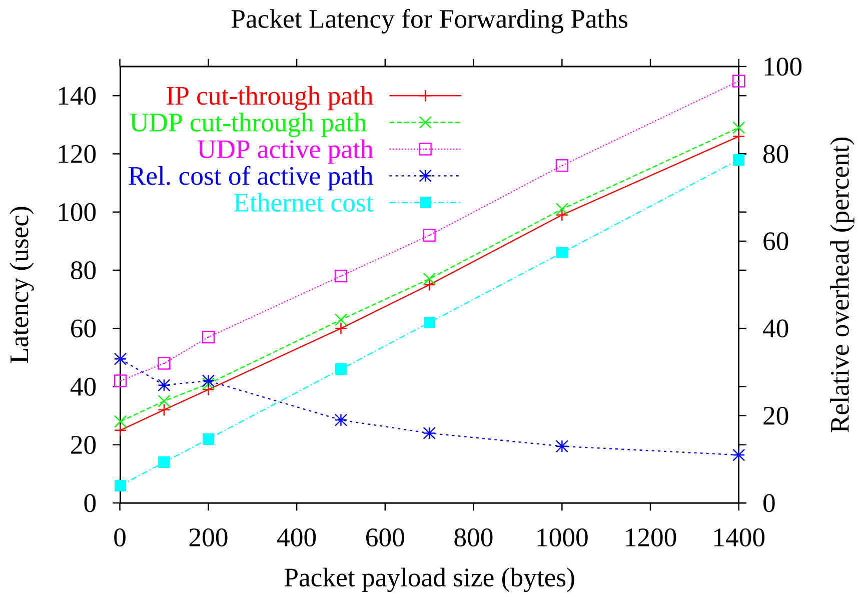


Figure 6: Packet latency for three different forwarding paths (IP cut-through, UDP cut-through and UDP active paths), with the latency cost of one 100Mbps Ethernet hop.

In our second experiment, we turn to latency measurements of the three aforementioned forwarding paths, which we depict for varying packet sizes in Figure 6. To put this evaluation in perspective, we include the latency component for a 100Mbps Ethernet hop per packet (half the latency cost is attributed to the sender and half to the receiver). We observe that latency linearly increases with packet size. The UDP cut-through path is only marginally more costly than its minimal IP counterpart. The difference between the UDP cut-through and active paths is approximately 15 μsec, which quantifies the overhead of two components: crossing into user space and back from Silk (the bulk of this overhead), and the NodeOS Scout module (minimal portion of overhead). We also plot the relative cost of forwarding active packets through the router, which starts at 33% for minimal packet sizes, and gradually drops to 11% for maximum size packets.

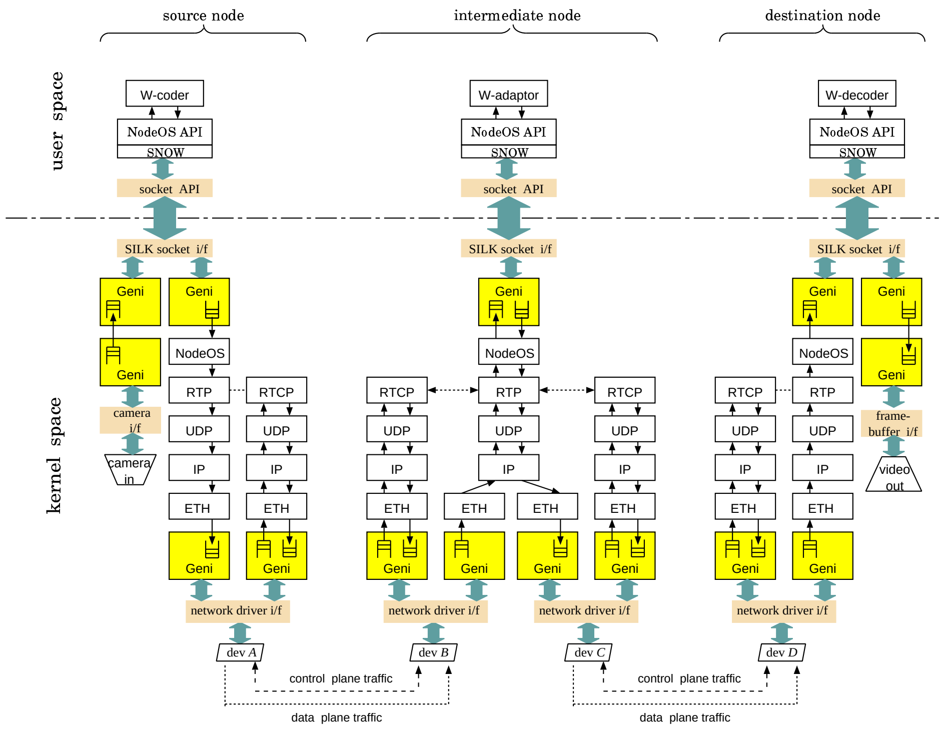


Figure 7: Wavelet dropper application invoking RTP and RTCP paths from user space. Source, intermediate and sync nodes execute the wavelet encoder, adaptor and decoder, respectively.

To demonstrate how an active application with distinct data and control plane components makes use of our NodeOS architecture, we setup a third experiment that implements the wavelet dropper [9] on three machines — the source, intermediate, and sync nodes, representing the wavelet encoder, transcoder and decoder, respectively, as illustrated in Figure 7. The wavelet transcoder (or dropper) divides a wavelet encoded video stream into multiple layers. Depending on the level of congestion experienced at a router, high-frequency packet layers are dropped. Scout's RTP [24] protocol module records the number of packets successfully forwarded per flow, while the RTCP [24] protocol module uses this information to determine the available forwarding rate, and from this, the cutoff layer for forwarding, which it then sends back to the data forwarder.

We measured the latency for a wavelet video stream, encoded into maximum size Ethernet packets, from source (camera) to destination (video) via an intermediate transcoder node, with no network congestion, to be 306μsec. To assure this latency is independent of our particular implementation of the wavelet algorithm (as well as the EE it would be running on, in practice), we factored out the time spent in the user space wavelet encoder, transcoder and decoder modules, thereby only reporting the remaining relevant part of this configuration, including the two network hops, as well as the three roundtrip crossings from kernel to user space. The network latency accounted for 236μsec, or 77% of this cost, demonstrating that our NodeOS implementation, along with its ability to process packets at line speeds, was efficient in providing the extra RTP/RTCP functionality within the kernel, and crossing into user space for setting up the active video streams.

6 Design Extensibility

An important feature of our base design is that apart from providing a NodeOS that meets our design goals, it enables us to deploy other systems with different networking architectures. One example is that we can embed a hierarchical extensible router into our framework, as shown in Figure 8.

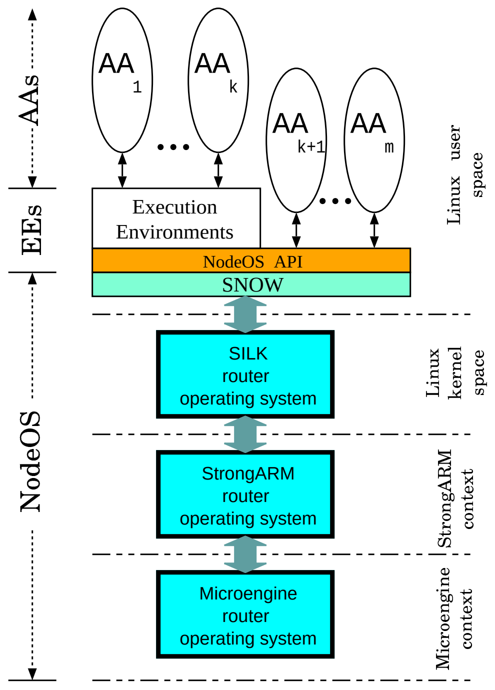
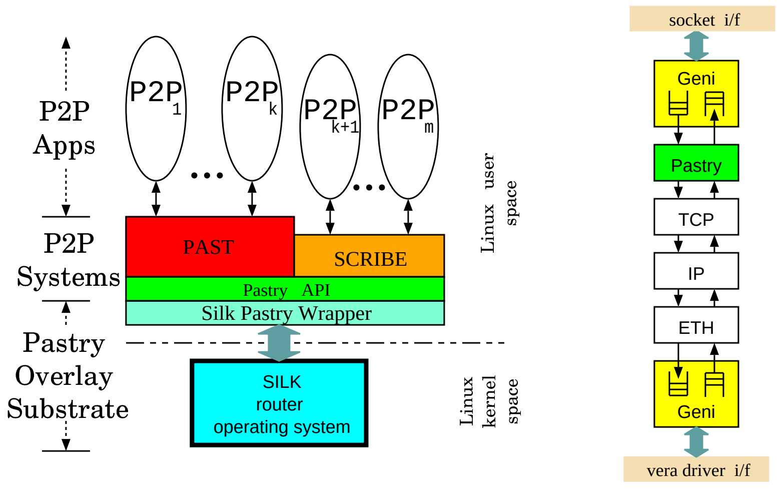


Figure 8: Mapping our NodeOS architecture to that of an Extensible Router — active packets can make use of the entire router hierarchy for data and control flows.

That particular extensible router [25] is comprised of connecting the Pentium, on which we run Silk, via the PCI bus to a network processor, namely an Intel IXP1200, which in turn consists of a StrongARM and a set of six MicroEngines executing in parallel. This results in four execution levels (shown in Figure 8), partitioning the router functionality among the hardware and software in concert, and corresponds to four distinct classes of paths through the system, from the fastest, at the lowest MicroEngine level, to the slowest, crossing into user space. From Silk's side, the PCI bus bridging to the IXP is accomplished via a special partner driver, `vera.o` [14].

Under such a setup, we view the NodeOS as the OS of the distributed extensible router, roughly spanning the lower three execution levels of Figure 8. Consequently, our design permits the NodeOS API abstractions to take advantage of the entire hierarchical OS of the prototype router, with its ability to process packets at line speeds, despite the added extensibility and functionality.

Another example is employing our base architecture to design a peer–to–peer (P2P) substrate node, such as Pastry [21], as depicted in Figure 9. Pastry has an asymmetric but simple API, where it exports two operations, *pastryInit* for node joins, and *route* for routing messages among the Pastry nodes; and to notify its applications about message *delivery* and *forward*ing, as well as a change in the node's leafSet status, *newLeafs*. Two such applications are layered on top of Pastry in Figure 9: PAST [22], a P2P storage management and caching systems, and SCRIBE [23], a P2P publishing and subscription system.



Architecture of the Pastry P2P Node in Silk   Pastry Path in Silk

Figure 9: Employing our base architecture to design a Pastry P2P Node by providing a layer of Silk Pastry Wrappers, and a Pastry module for Scout paths

To employ Pastry, only two components in our architecture need to be re-implemented for this purpose: Silk Pastry Wrappers would replace Snow, and a Pastry module for Scout paths instead of its NodeOS counterpart. All the other mapping mechanisms such as sockets, the partner interface and Geni remain intact. We currently have such a system under development.

We intend to further extend the P2P capability of our architecture by embedding a P2P application, such as PAST, into the kernel, for the sake of better performance. The architectural implications of such a project are illustrated in Figure 10. The PAST API consists of three operations, *insert, lookup* and *reclaim*, to store, retrieve and destroy a file at the node, respectively.

To embed the PAST functionality in Silk, two changes need to be made to the components

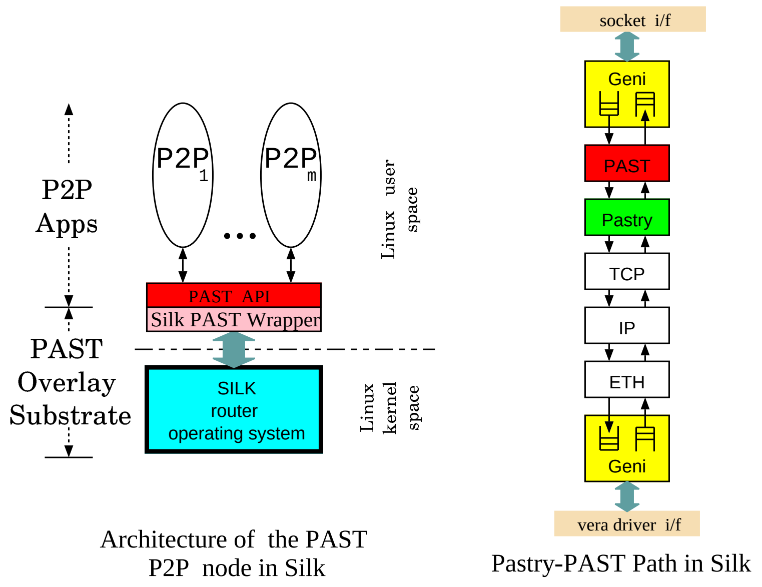


Figure 10: Employing our base architecture to design a Past P2P Node by providing a layer of Silk PAST Wrappers, and a PAST module on top of Pastry for Scout paths

of Figure 9: Silk PAST Wrappers would replace the Silk Pastry layer, and a PAST module for Scout paths would have to be additionally implemented to interface to its Pastry counterpart. We envisage this as future work.

7 Related Work

To date, several NodeOS implementations have been reported. Bowman [15], runs on generic UNIX substrate, and therefore cannot provide fine grained resource control. Additionally, being the first NodeOS, it was developed in the early days of the specification and provides only a subset of the interfaces.

Three other NodeOS implementations were discussed and compared in [19]. The first, implemented within stand–alone Scout [17], is the closest to our Silk implementation, where channel fucntionality is wrapped around Scout paths. However, no protection domains were maintained, necessitating EEs and AAs to implement their own security. Another shortcoming is that Scout is not a commonly used OS. Relative to that, the main shortcoming of our NodeOS in Silk is the overhead of context switching for small packets, as we have seen in Section 5.

Janos [27] is based on the OSKit component base [10], where the NodeOS and EEs shared a protection domain — running as a kernel on top of hardware or as one user space process on top of a Unix OS. Such a single address space system cannot guarantee separation between concurrent EEs. Finally, AMP [8] is layered on top of the MIT exokernel [13], and the NodeOS was implemented within a user space OS library, with its own memory and thread management. Although this design comparatively taxed system performance, its primary focus on security contributed several issues to the ongoing design of the NodeOS interface.

8 Conclusions

We set out to design and build a NodeOS which would be embedded within a widely used OS, while allowing non-active applications and regular OS operation to proceed in a regular manner, unhindered by the active networking component, and at the same time offer performance competitive with that of networking stacks of general purpose operating systems.

Our central contribution was to simultaneously satisfy these design goals with our underlying design mechanism: use established system components (namely Linux, Scout paths and Linux device drivers) with well known interfaces (such as sockets and the module interface to Scout paths), and construct mappings between those and the APIs and operations we designed for NodeOS (namely the NodeOS API, Snow and the Partner Interface), thereby delivering kernel–level NodeOS functionality from user space.

Additionally, we have contributed to decoupling the NodeOS specification from EEs/AAs from above, and the NodeOS implementation from below, thereby delivering a *portable* set of header files, allowing EE/AA/NodeOS modifications in any direction. this resulted in the additional contribution of delivering a system with distinctly established protection boundaries between the AA/EE and NodeOS space, in terms of memory management, process scheduling and security.

We demonstrated that our system is robust, sustaining line speed bandwidth for cut-through and active flows that cross into user space, and measured that the relative overhead cost of forwarding active paths within our NodeOS is around 11% for maximal size packets. Forwarding latencies were shown to scale linearly with packet sizes for cut-through and active paths, ranging from 40μsec to 145μsec for minimal and maximum sized active packets, respectively. A prototype application that uses the NodeOS API, the wavelet dropper, exhibited regular and expectable performance.

An additional contribution is the extensibility of our architecture. We demonstrate how to use our base architecture to deploy a P2P system or an extensible router by only changing the user space API from above, or extending the NodeOS to a hierarchical system from below, respectively.

References

[1] Active Networks Security Working Group. Security architecture for active nets. Available as `ftp://www.ftp.tislabs.com/pub/activenets/secarch2.ps`, July 1998.

[2] D. S. Alexander, M. Shaw, S. M. Nettles, and J. M. Smith. Active Bridging. In *Proceedings of the ACM SIGCOMM '97 Conference*, pages 101–111, September 1997.

[3] AN NodeOS Working Group. NodeOS interface specification. Available as `http://www.cs.princeton.edu/nsg/papers/nodeos.ps`, Jan. 2002.

[4] A. Bavier, T. Voigt, M. Wawrzoniak, and L. Peterson. SILK: Scout Paths in the Linux Kernel. Technical Report TR–2002–009,, Department of Information Technology, February 2002.

[5] S. Berson, B. Braden, T. Faber, and B. Lindell. The ASP EE: An Active Network Execution Environment. In *Proceedings of the DARPA Active Networks Conference and Exposition (DANCE)*, San Francisco, CA, May 2002.

[6] S. Bhattacharjee, K. Calvert, and E. Zegura. Congestion control and caching in CANES. In *ICC '98*, 1998.

[7] K. Calvert. Architectural framework for active networks. Available as `http://www.cs.gatech.edu/projects/canes/papers/arch1-0.ps.gz`, July 1999.

[8] H. Dandekar, A. Purtell, and S. Schwab. AMP: Experiences in building an exokernel–based platform for active networking. In *Proceedings of the DARPA Active Networks Conference and Exposition (DANCE)*, San Francisco, CA, May 2002.

[9] M. Dasen, G. Fankhauser, and B. Plattner. An Error Tolerant, Scalable Video Stream Encoding and Compression for Mobile Computing. In *Proceedings of ACTS Mobile Summit 96*, pages 762–771, November 1996.

[10] B. Ford, G. Back, G. Benson, J. Lepreau, A. Lin, and O. Shivers. The Flux OSKit: A substrate for OS and language research. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles*, pages 38–51, St. Malo, France, October 1997.

[11] M. Hicks, P. Kakkar, J. T. Moore, C. A. Gunter, and S. Nettles. PLAN: A packet language for active networks. In *Proceedings of the 3rd ACM SIGPLAN International Conference on Functional Programming Languages*, pages 86–93, September 1998.

[12] A. W. Jackson, J. P. Sterbenz, M. N. Condell, and R. R. Hain. Active Monitoring and Control: The SENCOMM Architecture and Implementation. In *Proceedings of the DARPA Active Networks Conference and Exposition (DANCE)*, San Francisco, CA, May 2002.

[13] M. F. Kaashoek, D. R. Engler, G. R. Ganger, H. Briceno, R. Hunt, D. Mazieres, T. Pinckney, R. Grimm, J. Jannotti, and K. Mackenzie. Application performance and flexibility on exokernel systems. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles*, pages 52–65, St. Malo, France, October 1997.

[14] S. Karlin and L. Peterson. VERA: An Extensible Router Architecture. *Computer Networks*, 38(3):277–293, 2002.

[15] S. Merugu, S. Bhattacharjee, E. Zegura, and K. Calvert. BOWMAN: A node OS for Active Networks. In *Infocom 2000*, March 2000.

[16] J. T. Moore, M. Hicks, and S. Nettles. Practical programmable packets. In *Proceedings of the Twentieth IEEE Computer and Communication Society INFOCOM Conference*, pages 41–50, April 2001.

[17] D. Mosberger and L. L. Peterson. Making Paths Explicit in the Scout Operating System. In *Proceedings of the Second USENIX Symposium on* , pages 153–167, Seattle, WA USA, October 1996.

[18] S. Murphy, E. Lewis, R. Puga, R. Watson, and R. Yee. Strong security for active networks. In *Proceedings of the Open Architectures 2001 Conference*, pages 1–8, Anchorage, AK USA, April 2001.

[19] L. Peterson, Y. Gottlieb, M. Hibler, P. Tullmann, J. Lepreau, S. Schwab, H. Dandelkar, A. Purtell, and J. Hartman. An OS Interface for Active Routers. *IEEE Journal on Selected Areas in Communications*, 19(3):473–487, March 2001.

[20] X. Qie, A. Bavier, L. Peterson, and S. C. Karlin. Scheduling Computations on a Software-Based Router. In *Proceedings of the ACM SIGMETRICS 2001 Conference*, pages 13–24, June 2001.

[21] A. Rowstron and P. Druschel. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In *IFIP/ACM International Conference on Distributed Systems Platforms (Middleware)*, pages 329–350, November 2001.

[22] A. Rowstron and P. Druschel. Storage management and caching in PAST, a large-scale, persistent peer-to-peer storage utility. In *18th ACM Symposium on Operating Systems Principles*, pages 188–201, October 2001.

[23] A. Rowstron, A.-M. Kermarrec, M. Castro, and P. Druschel. Scribe: The design of a large-scale event notification infrastructure. In J. Crowcroft and M. Hofmann, editors, *Networked Group Communication, Third International COST264 Workshop (NGC'2001)*, volume 2233 of *Lecture Notes in Computer Science*, pages 30–43, November 2001.

[24] H. Schulzrinne, S. Casner, R. Frederick, and V. Jacobson. RTP: A transport protocol for real-time applications; RFC 1812. *Internet Request for Comments*, January 1996.

[25] N. Shalaby, L. Peterson, A. Bavier, Y. Gottlieb, S. Karlin, A. Nakao, X. Qie, T. Spalink, and M. Wawrzoniak. Extensible Routers for Active Networks. In *Proceedings of the DARPA Active Networks Conference and Exposition (DANCE)*, San Francisco, CA, May 2002.

[26] J. M. Smith, K. L. Calvert, S. L. Murphy, H. K. Orman, and L. L. Peterson. Activating Networks: A Progress Report. *IEEE Computer*, 32(4):32–41, April 1999.

[27] P. Tullmann, M. Hibler, and J. Lepreau. Janos: a Java–oriented OS for Active Networks. *IEEE Journal on Selected Areas in Communications*, 19(3), March 2001.

[28] M. Wawrzoniak, N. Shalaby, and L. Peterson. Intelligent Devices as Symmetric Partners: Architecture and Implementation. Technical Report TR–642–02, Department of Computer Science, Princeton University, January 2002.

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

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