Crossroads
The ACM Student Magazine

An Introduction to Active Network Node Operating Systems

by ***Parveen Patel***

Introduction

An operating system performs at least two important tasks for user applications. First, it shields application programs from hardware details and provides a simple interface to work with. Second, it manages shared resources, such as CPU, memory, hard disk, network interface, or scanner. Based on the type of applications most commonly run, operating systems can be broadly classified into two categories: **general-purpose** operating systems and **special-purpose** operating systems. General-purpose operating systems are those that mostly run common end-user applications, like word processors, web browsers, and compilers, etc. Linux, Unix, Mac OS, and Microsoft Windows are all examples of general-purpose operating systems. Special-purpose operating systems, on the other hand, are designed and/or optimized to run one special type of application, such as database services, multimedia streaming, and internetwork services. Examples of special-purpose operating systems include Oracle's RDBMS server, Cisco's Internetwork Operating System (IOS), and Novell Netware. It is not uncommon for operating systems to support both general-purpose and some special-purpose applications. For example, Unix and Windows NT are also successful

network operating systems (A network operating system is one that manages shared network resources such as printers and data storage). This feature gives an introduction to a special-purpose operating system called **active network node operating system**, or a **NodeOS** [2][3], whose primary function is to support packet forwarding in an **active network**.

Data packets in a traditional computer internetwork, such as the Internet, are routed from one machine to another by special nodes inside the network called **routers**. A router in the Internet decides which subsequent router a packet should go to in order to reach its final destination. A packet does not perform any activity inside the network and gets forwarded to its destination by the routers. So a data packet in a traditional network is a passive entity. In contrast, data packets in an **active network** are *active* inside the network. On a router, an active packet selects an **active application** which will process this packet. An active application ideally can perform any arbitrary function on the active packet. For example, an active packet can invoke an application-specific routing function on itself, or it may encrypt itself before passing through a foreign network. Users of the network write the code for active applications. An active packet can either carry the code for its active application, e.g., Java byte-code, or active applications may be distributed off-line to the routers and active packets can carry an identifier of the active application, e.g., a class name or a method name. Contrast this with the Internet, where the code that acts on a packet is fixed in the router and is installed by the router vendors. Active networking gives users much more flexibility and control over what happens to their packets inside the network. But this increased flexibility and control comes with its own price. Because active applications are written by the users of the network, as opposed to router vendors, they are more likely to be buggy or malicious. A buggy active application might corrupt the state of a router or might harm other active applications. A malicious active application may try to take control of the router or the network, or may launch attacks on other machines in the network. Therefore, to prevent any kind of misbehavior, an active application must be run in a controlled execution environment, called **active network execution environment**.

A NodeOS is a special-purpose operating system that runs on the routers of an active network and supports active network execution environments (A router in an active network is called an **active node**, and hence the name NodeOS). In order to prevent active applications from misbehaving, active network execution environments enforce fine-grained control over the resources consumed by active applications. For example,

an execution environment may restrict the number of CPU cycles an active application can consume, or it may enforce a limit on the number and type of packets an active application can receive and send. The interface/API provided by traditional operating systems is inadequate for such needs of execution environments. For example, in traditional Unix, where all resources are associated with a **process**, it is very difficult to enforce an absolute limit on resources consumed by an active application if it is not a process, and active networks would be very slow if each active application is run in a separate Unix process. A NodeOS provides the exact interface needed by active network execution environments. A NodeOS is also different from a traditional OS in terms of the overhead it imposes to do its job. Further, a NodeOS should be capable of handling as many network packets per second as possible. Therefore, the NodeOS should impose minimum overhead to perform operating system functions. The above requirements raise interesting operating system design issues, primarily in the areas of API design and effificent resource control. We will look at these issues and how two existing NodeOSes solve them in the following sections. First, we present a brief background on active networks, define the necessary terminology, and identify the role of a NodeOS in an active network. Next, we discuss the primary services provided by a NodeOS and then the important operating system design issues for a NodeOS. We follow with a brief discussion of two examples of existing NodeOSes, and then present a conclusion to end the feature.

Active Networks

In traditional IP-based internetworks, such as the Internet, network middleboxes, called routers or switches, maintain the network topology and make decisions about the route a packet of data should take to reach its destination(s). Because of the nature of their primary function and their strategic position in a network, routers can be used to provide many value-added network services, such as support for real-time applications, multicast, or web caching. Commonly in today's networks, routers use proprietary hardware and software to do their job. Therefore, it is very difficult to add or experiment with new services that can be provided using routers. Active Networking is an approach to open up these middle boxes and make them programmable by end-users [**4**][**5**][**6**]. The main idea behind active networking is to build more efficient services by combining application-specific computation with network-related knowledge on a router. In an active network, routers of the network, called **active nodes**, perform customized computations on the packets flowing through them. Applications of an active network, called **active applications** (**AA**s), can download code into active nodes and customize the network to their needs. An example application of active

networking is **Fusion**. In Fusion, multiple packets destined towards the same router or end-host can be combined into a single packet. Which packets can be combined into a single packet, and how they are combined, is application-specific. For example, for a remote GUI application, multiple updates to the same window can be combined into a single update. Fusion is a useful technique to reduce network traffic by doing application-specific computation inside the network. For further details of active networking beyond the scope of this discussion, interested readers may refer to [**4**], [**5**], and [**6**].

An Active Node

As described above, an active node or an active router is a router that supports active applications. **Figure 1** shows the diagram of various software layers present on an active node [**2**][**5**]. As shown in the diagram, an active application runs within an **active network execution environment** (**EE**) and does not directly interact with the underlying operating system. In an active network, all end hosts and routers run an instance of the EE. An EE defines a particular programming interface for AAs and provides access to services and resources available in the network.

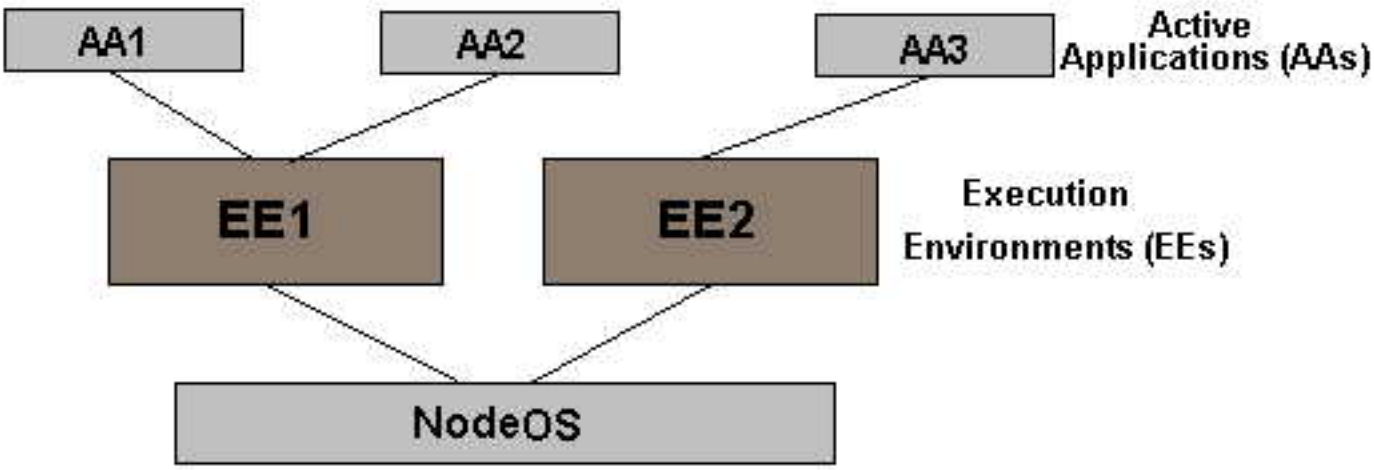

Figure 1: Software Structure of an Active Node.

An EE is also responsible for protecting itself from misbehaving AAs (an example being that a segmentation fault in an active application should not crash the EE). The lowest layer of software that runs on an active node is called a **node operating system** or a **NodeOS** . A NodeOS is responsible for multiplexing nodes' resources among multiple EEs and protecting itself and the node from misbehaving EEs. The NodeOS allocates

nodes' resources among competing EEs, which in turn redistribute their resource among competing AAs. Thus the three software layers on an active node are organized into a three-level hierarchy for resource management and protection. In this feature we focus on the lowest layer of this hierarchy: the NodeOS. The following section discusses the primary services provided by the NodeOS.

Services provided by the NodeOS

As discussed in the previous section, a NodeOS provides services needed by EEs which support active applications. A NodeOS, however, does not need to provide services needed by general-purpose applications. For example, a NodeOS need not support graphical user interface or printers. In this section we look at some of the important services that a NodeOS provides.

1. **Fast Packet Input and Output:** Packet input and output is one of the most basic services a NodeOS needs to provide. An EE should be able to receive and send packets of its own choosing. EEs should be able to conveneintly specify which packets they (and their AAs) are interested in and what output bandwidth they require. In a NodeOS packet, input and output should be highly optimized so that more packets from the network can be processed.
2. **Memory Allocation and Deallocation:** A NodeOS is responsible for managing the main memory available on a node. EEs allocate, deallocate, and share main memory by requesting the NodeOS. A significant part of main memory will be used by EEs to receiver input packets and send output packets. Thus, a NodeOS may provide a separate memory allocation/deallocation scheme for network packets, for example, by considering fixed-sized packets as unit of allocation, deallocation, and sharing rather than variably-sized byte chunks.
3. **CPU Scheduling:** A NodeOS multiplexes computation resources by scheduling EEs on available CPUs. EEs depend on the NodeOS for receiving a fair share of the available cycles.
4. **Resource Accounting:** Precise resource accounting is crucial for active networking. The resources that are important from an active networking point of view include CPU cycles, main memory, and outgoing network bandwidth. These resources need to be divided among all EEs and in turn among AAs. Because an EE runs applications downloaded by arbitrary end-users (possibly malicious), it needs to ensure that no active application consumes more resources than its allocated share. Because all resources are actually allocated by the NodeOS, only it can properly account for them. An EE relies on the NodeOS for proper resource

accounting. The traditional resource accounting in a Unix-like operating system is inappropriate for active networking. For example, the resources that are consumed by Unix when an interrupt is triggered, e.g., when a network packet arrives, are generally charged to the process that happens to be running at that time and not to the process that actually receives the packet. Similarly packets received by one process can block packets for other processes by not freeing the kernel packet buffers fast enough. These and many other problems with Unix resource management were identified in [12] and a solution called **resource containers** was proposed. A resource container is a logical resource bin in an operating system that a process can create and use to do an independent activity. If the NodeOS provides resource containers, then an EE can create a separate resource container for each active application and fill it with its share of resources. The NodeOS can then make sure that no active application consumes more resources than available in its associated resource container. As we will see in later sections, existing NodeOSes provide a resource container like abstraction.

5. **Disk Access:** A NodeOS needs to provide access to available persistent storage. A NodeOS can provide disk access with a filesystem-like interface.

OS design issues for the NodeOS

In this section, we discuss two important operating system design issues in which a NodeOS differs from a traditional Unix-like operating system: API design and resource management.

API Design

> *"Designing interfaces is the most important part of a system design and it is also usually the most difficult* [9]."

An Operating System provides access to node resources in a particular way by implementing a particular **Application Programmable Interface** (**API**). For example, most Unix-like operating systems, such as Linux and FreeBSD, implement the POSIX API standard; similarly, Microsoft Windows operating systems implement the Win32 API. The operating system API keeps the applications shielded from implementation details and is really essential for application portability. Designing an API that is suitable for all applications that can be run on an OS is really hard and takes experience to get right. The API design is also important because usually the design of the operating system itself depends on the API it supports. The OSes are generally

structured the way they are (e.g., micro-kernel or monolithic-kernel) because of the API they support.

The NodeOS implements a particular API to which the EEs can be conveniently programmed. The API supported by the NodeOS differs from traditional operating system interfaces because the primary purpose of its API is to support EEs as opposed to running general-purpose applications. Thus, the API for a NodeOS reflects the needs of EEs. For example, the API should include functions which the EEs can use to register interest in incoming packets and reserve outgoing bandwidth. In the next section we will look at one standard API that some of the exisiting NodeOSes implement.

Resource Management

The NodeOS is responsible for multiplexing node resources among multiple EEs. Resource management is one of the hardest problems any OS solves. The problem gets harder for a NodeOS because the resource management has to be both efficient and fast. Resource management needs to be efficient so that resources are better utilized, and it needs to be fast so that more packets from the network can be processed. An important design issue for resource management is the unit of resource management. In a Unix operating system a process is the unit of resource management. All resources are allocated to and accounted for each process. As discussed in the previous section, a process is not an appropriate unit of resource management for a NodeOS but a resource container is. Therefore, throughout this discussion we consider a resource container as the unit of resource management. Because resource containers can be organised into a hierarchy[**12**], we will assume that the resource containers associated with AAs are child resource containers of their EE.

The resources that are most interesting from an active networking point of view are: communication (network bandwidth), memory (main memory), and computational resources (CPU cycles). In the following section, we discuss issues related to the management of these resources.

1. **Communication resources:** The NodeOS needs to multiplex its input and output link bandwidth among competing EEs. For incoming network bandwidth, the NodeOS ensures that no EE can snoop on the incoming packets of another EE. For the outgoing network bandwidth, the NodeOS needs to ensure that all resource containers get their allocated share. It is very difficult to tightly control

the input bandwidth consumed by a resource container because an EE (or an AA) generally has no control over the rate of incoming packets. Output link bandwidth, on the other hand, can be tightly controlled. The NodeOS controls the output link bandwidth by scheduling the output packet requests associated with a resource container based on its allocated network bandwidth.

2. **Memory Management:** The NodeOS needs to provide support for fair allocation, deallocation, and sharing of memory among resource containers. As discussed in the previous section, in a NodeOS a significant part of the main memory is used for input and output network packets. Therefore, the NodeOS may directly support allocation, deallocation, and sharing of memory in terms of network packets in addition to in terms of raw bytes. One important way in which memory management in a NodeOS differs from a traditional operating system has to do with the memory used inside the operating system kernel. In Unix-like operaing systems, kernel memory is shared by all processes and one user process may consume excessive kernel memory and prevent other processes from getting their fair share. A NodeOS should properly allocate the kernel memory among resource containers and make sure that no resource container can exceed its limit. Another important aspect of memory management is protection. The NodeOS may use virtual memory techniques to create protected address spaces among EEs or it may rely on EEs being well behaved (e.g., because they are written in a Java-like language that doesn't allow raw pointers to memory).
3. **CPU scheduling:** A NodeOS needs to fairly allocate CPU cycles among resource containers. A NodeOS that supports multi-threading can allocate a fixed number of threads for each resource container and schedule threads based on resource containers' CPU limits. Similar to memory and communication resources, CPU cycles consumed inside the kernel should also be properly charged to responsible resource containers.

The NodeOS API

The NodeOS interface or **the NodeOS API** [3] is a standard specification that defines a set of primary abstractions and interfaces the NodeOS should implement. To get a feel for how a NodeOS is different from a Unix, we look at the five important abstractions that are defined by the NodeOS API. Further details of the NodeOS API are out of the scope of this introductory discussion.

1. **Domains:** A domain is a resource container in the NodeOS. The need for separating resource containers from processes was first felt for web servers [12]. Domains can contain child domains hence forming a hierarchy of domains. On the NodeOS interface, an EE can create a domain for each AA and attach a share of resources with the domain. The NodeOS ensures that no domain consumes more resources than its allocated share.
2. **Channels:** Channels are the primary abstraction used to send, receive, and forward packets. The NodeOS API defines three types of channels: input channels, output channels, and cut-through channels. Input channels are used to receive packets, output channels to send packets, and cut-through channels to forward packets without any EE or AA intervention.
3. **Thread Pool:** Thread pools represent the CPU cycles and are associated with domains. An EE can create separate thread pools for each AA and hence control the CPU cycles consumed by an AA.
4. **Memory Pool:** Memory pools represent the main memory and are associated with domains. A memory pool may be associated with more than one domain to enable sharing of memory. In order to do proper memory accounting, packets that are used to buffer input and output packets for a channel are allocated from the memory pool associated with the channel's domain.
5. **Files** Files represent persistent storage and enable offline sharing of data. File interfaces in the NodeOS API are similar to those in the POSIX standard.

The NodeOS API has been implemented by at least three NodeOSes [2]. In the next section we briefly discuss two NodeOSes that implement the NodeOS API.

Example NodeOSes

Moab

Janos [1], a Java oriented Active Network Operating System, was developed at the University of Utah and is currently in use by many research projects.

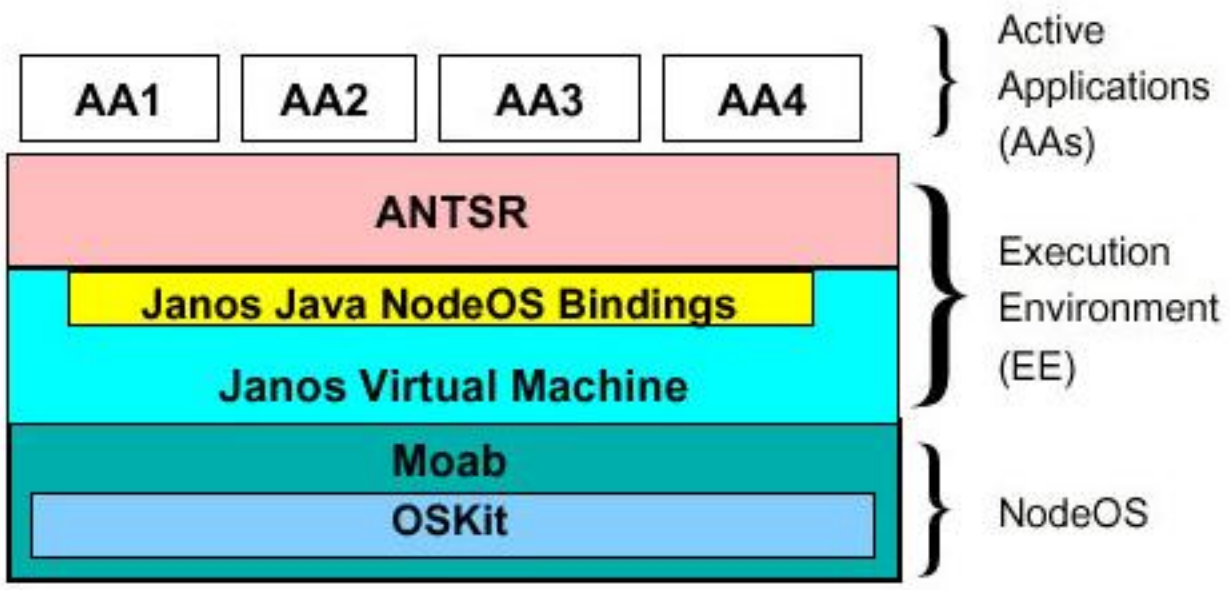

Figure 2: Janos Architecture.

As shown in **Figure 2**, Janos implements both infrastructural layers that run on an active node: the NodeOS, and a resource-aware Java Virtual Machine based execution environment. The Janos NodeOS layer is implemented by Moab. Moab is a multi-threaded, fully-preemptible, single-address-space operating system implementing the NodeOS interface. Moab is built using the OSKit[**9**], an OS component library for building systems software. The OSKit includes suites of device drivers, numerous file systems, 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 Moab implementation is generally compliant with the NodeOS interface but it differs from it in some minor ways [**2**]. In Moab the NodeOS API calls are implemented as direct function calls rather than system calls hence improving the performance of the system. Another important feature of Moab is that it exposes Java bindings for the NodeOS API, hence enabling implementation of other Java-based EEs.

Scout

The Scout operating system [**7**] also implements the NodeOS API [**2**]. Scout is a configurable system specifically designed for network packet processing. An instance of Scout is created by constructing a graph of required modules from a set of existing modules. The Scout operating system has an explicit notion of a **path**, which

encapsulates the flow of I/O data from an input device to an output device. Similarity of the path abstraction to the active node model lets Scout implement both active and non-active routers using the same model [7]. **Figure 3** shows a Scout module graph that constructs infrastructural layers of an active node.

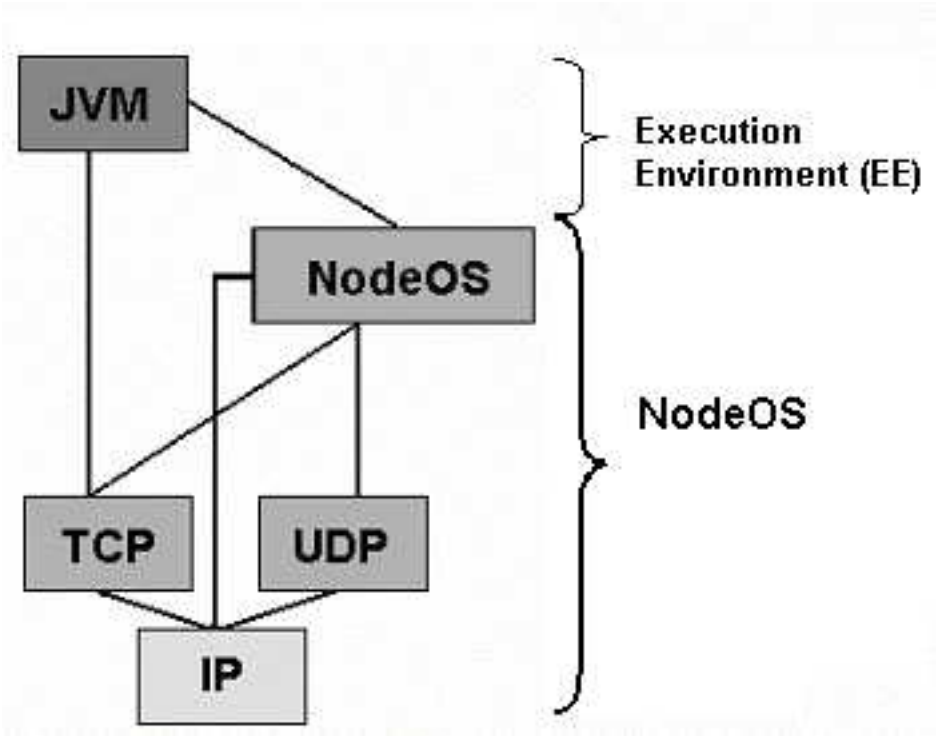

Figure 3: Scout Active Node Module Graph.

As shown in **Figure 3**, the NodeOS module, and all the modules below it, implement the NodeOS layer of an active node. The JVM module is an EE that runs active applications. From Scout's perspective, the NodeOS and the EE layers are constructed from a combination of system modules, while AAs are constructed from a combination of user modules. The three layers of an active node combine together to form an *active forwarding path*.

Conclusion

An active network node operating system or a NodeOS is a special-purpose operating system whose primary purpose is to support packet forwarding in an active network. The NodeOS runs at the lowest level in an active node and multiplexes node resources among active network execution environments. The NodeOS provides a number of imporant services to active network execution environments including resource scheduling and accounting and fast packet input-output. The two important operating system design issues for a NodeOS are API design and resource management. The

NodeOS interface/API is a proposed API standard which specifies important abstractions and interfaces that a NodeOS should implement. A resource container is a better unit of resource control in a NodeOS than a process. A NodeOS manages its resources among various resource containers and does proper accounting of resources consumed even inside the OS kernel. Moab and Scout are two exisitng NodeOSes that implement the NodeOS API. Moab is implemented as a multi-threaded, fully-preemptible, single-space operating system, using the OSKit. Scout is a configurable network-centric operating system that implements the active node infrastructure with a set of Scout modules.

References

1

P. Tullmann, M. Hibler, and J. Lepreau. Janos: A Java-oriented OS for Active Networks. Appears in *IEEE Journal on Selected Areas of Communication*. Volume 19, Number 3, March 2001.

2

L. Peterson, Y. Gottlieb, S. Schwab, S. Rho, M. Hibler, P. Tullmann, J. Lepreau, and J. Hartman. An OS Interface for Active Routers. IEEE Journal on Selected Areas in Communications, 2001.

3

AN Node OS Working Group. *NodeOS Interface Specification*, Edited by Larry Peterson (January 2000).

4

Wetherall, D. "Active network vision and reality: lessons from a capsule-based system", Operating Systems Review, vol.33, (no.5), ACM, Dec. 1999. p.64-79.

5

J. Smith, K. Calvert, S. Murphy, H. Orman, and L. Peterson. Activating Networks: A Progress Report. *IEEE Computer* (April 1999).

6

D. Tennenhouse, J. Smith, W. Sincoskie, D. Wetherall, and G. Minden. A Survey of Active Network Research. IEEE Communications Magazine, pages 80--86, January 1997.

7

A. B. Montz, D. Mosberger, S. W. O'Malley, L. L. Peterson, T. A. Proebsting. Scout: A Communications-Oriented Operating System. *Hot OS* (May 1995).

8

Lampson, B. W. Hints for Computer System Design. IEEE Software, 1(1):11--28, January 1984.

9

B. Ford, G. Back, G. Benson, J. Lepreau, A. Lin, and O. Shivers. The Flux OSKit:

A Substrate for Kernel and Language Research. In Proceedings of the Sixteenth ACM Symposium on Operating System Principles, pages 38--51, Saint-Malo, France, 1997.

10

R. Wahbe, S. Lucco, T. Anderson and S. Graham. Efficient Software-Based Fault Isolation. *Proc. Fourteenth ACM Symposium on Operating System Principles* (December 1993), pp. 203--216.

11

G. Necula. Proof-carrying code. In Proceedings of the 24th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Langauges (POPL '97), January 1997.

12

G. Banga, P. Druschel, and J. C. Mogul. Resource containers: A new facility for resource management in server systems. In Proceedings of the Third Symposium on Operating Systems Design and Implementation (OSDI '99), pages 45--58, 1999.

Biography

Parveen Patel (**ppatel@cs.utah.edu**) is a graduate student at the School of Computing at the Univeristy of Utah. He works with the **flux** group.