Janos: A Java-oriented OS for Active Network Nodes

Patrick Tullmann, Mike Hibler, Jay Lepreau
Flux Research Group
University of Utah
http://www.cs.utah.edu/flux/janos/

March 2001

Abstract

Janos is an operating system for active network nodes whose primary focus is strong resource management and control of untrusted active applications written in Java. Janos includes the three major components of a Java-based active network operating system: the low-level NodeOS, a resource-aware Java Virtual Machine, and an active network protocol execution environment. Each of these components is separately usable. This article lays out the Janos design and its rationale.

1 Introduction

An active network is akin to a regular network: packets of data are routed through the network from a source node to one or more destination nodes. In an active network, the "routers" are programmable devices. The logic for manipulating a packet while at intermediate nodes can be much more flexible than simple, hard-coded destination routing. Packets in an active network can be manipulated, copied, and modified by dynamically installed code.

The software at an endpoint (host) or intermediate (router) node in an active network can be described in terms of a model that divides the system into three logical layers [2]: the *NodeOS*, the *execution environment*, and *active application* layers. The NodeOS layer [1] abstracts the hardware and provides low-level resource management facilities. An execution environment (**EE**) sits atop the NodeOS and provides the basic application programming interface (API) available to the active network programmer. For example, an EE may be a virtual machine or simply a set of rules for interpreting packet headers. Conceptually, there may be several EEs running above the NodeOS, each providing a separate programming environment. The third and topmost layer of the architecture comprises the active applications (**AA**s), each of which contains code injected into the network to support a network service. In this model, a *domain*—similar to a process in a traditional operating system—is the unit of resource control and termination. An active application runs in the context of a domain, analogous to the way an executable image runs in the context of a process on a traditional operating system. The NodeOS and EE software are generally installed by a node administrator. An EE supports dynamic installation and removal of AAs in response to administrator or user demands. The active network node architecture and nomenclature are part of the DARPA active network community's architectural framework and not a direct development of our research, although we are participants in the community design process.

Janos is our operating system for active nodes, implementing both the NodeOS and EE layers of the active node model, described above. Active applications for Janos are written to our ANTS-based [43] ANTSR runtime [39], which runs atop our modified, resource-conscious Java virtual machine called the JANOSVM. Together, these two components constitute the EE layer and run on Moab, our implementation of the NodeOS. The JANOSVM and ANTSR runtime support AAs with an environment quite similar to a Java runtime [20]—in fact, the language syntax is identical—though ANTSR provides a narrower set of standard libraries. The ANTSR API provides the active application programmer with simple mechanisms for dynamic, on-demand code loading, implicit registration of packet-matching keys, and dispatch of packets. Additionally, simple facilities for maintaining soft state and logging are also provided. Figure 1 shows the relationship between the components of Janos and how they align with the DARPA active node model.

Janos is designed to prevent separate active applications from interfering with one another and to provide node administrators with strong controls over active ap-

This research was largely supported by the Defense Advanced Research Projects Agency, monitored by the Air Force Research Laboratory, Rome Research Site, USAF, under agreements F30602-96-2-0269 and F30602-99-1-0503. The United States Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright annotation hereon.

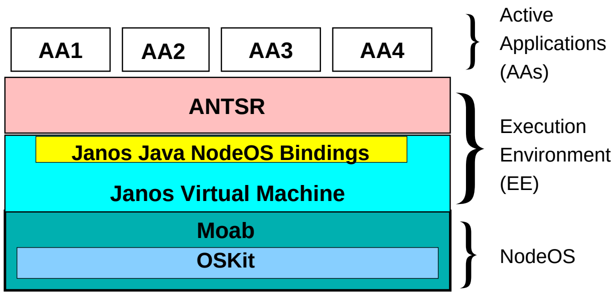

Figure 1: The Janos Architecture and the corresponding DARPA Active Network Node Architecture. Moab is our NodeOS layer. It is built on our OS component library, the OSKit. The EE layer is comprised of the JANOSVM, the Janos Java NodeOS, and ANTSR. AAs are hosted on this foundation.

plications' resource usage. To this end, the ANTSR runtime is carefully designed to constrain unauthorized, malicious, or buggy code by hiding interfaces for security- and resource-critical operations such as thread termination, packet spoofing, incoming packet matchers, resource limit changes, or shutting down a node. Operations along the lines of those just mentioned are made available only to authorized users, i.e., a node administrator. The administrator of a Janos node, in addition to initiating the bootstrap process, must be able to control access to the node, assign resources, and query the node about its state. In Janos, an administrator is given a control interface to the ANTSR runtime. From there, the administrator can start and terminate domains, gather statistics about the node, change resource limits, and modify access privileges.

A critical challenge in building Janos from separate components (Moab, the JANOSVM and ANTSR) is to ensure that features are provided at the appropriate level and that there is no redundancy among the components. For example, our design provides for per-domain memory accounting in the JANOSVM—where the garbage collector already tracks memory usage—while Moab provides only coarse accounting services. In addition to supporting active network users and administrators, Janos was designed to support research and development of other active network systems. The three components of Janos and their subcomponents are designed to be separately reusable. The utility of these separate components is enhanced by the existence and adherence to community standards such as the NodeOS API.

The rest of this article describes the components of Janos in greater detail, with particular emphasis on those aspects of the design that allow Janos to provide strong resource control over active applications within an active network node. Section 2 lays out the goals of the Janos architecture while Section 3 presents the system's design. Section 4 follows with a list of open issues and areas unaddressed by Janos. The current status of the Janos implementation is described in Section 5. Finally, Section 6 summarizes work related to Janos.

2 Goals

The major research focus of Janos is to provide comprehensive, precise resource control over the execution of untrusted Java bytecode. To adequately meet this demand Janos is obliged to provide sufficient security—or the resource limits will just be circumvented—and sufficient performance—or there will be little that can be accomplished under a limit. This section presents the goals for Janos; the details of how these goals are met are presented in the Section 3.

We want Janos to track the existing, relevant community standards for active networks to improve its usefulness to other researchers. In addition to providing concrete targets and leveraging the designs of previous systems, adherence to standards makes comparisons between different implementations easier to perform [31].

2.1 Untrusted Code Support

Janos, as an active network node operating system, must support the execution of untrusted code near the lowest levels of packet receipt and dispatch. Such support implies several safety and security goals for Janos. First, user code (i.e., an active application) must see only those packets that it is allowed to see. For example, an unprivileged AA cannot install a filter that matches all incoming packets. Conversely, user code must send only packets that it is allowed to send: it may not "spoof" packets. Second, to preserve resource constraints and timely dispatch of incoming packets, all user code must execute "quickly" or be preemptible. Third, as in any operating system, user code must not interfere with other user programs, access data outside its scope, or interfere with the proper functioning of the operating system itself. Finally, Janos must always have the ability to terminate user code.

2.2 Resource Management Goals

Janos must be able to limit the use of resources by running AAs, and be able to reclaim resources from terminated AAs. Janos does not attempt to address the problem of resource management on a network-wide basis; Janos provides resource management only on a single node. Janos' goal, however, is to provide a sufficiently strong foundation for others to implement network-wide resource management schemes. Specifically, resource control in Janos encompasses three independent resources: memory, CPU usage, and outgoing network bandwidth.

For memory, Janos has three management goals: first, to be able to guarantee and enforce a limit on each domain's memory usage; second, to be able to reclaim memory from terminated domains; and third, to provide active applications with sufficient infrastructure to allow them to manage their memory internally. The first two goals are important in maintaining the integrity and availability of a Janos-based node as a whole. The third goal is important to the authors of active applications. Specifically, for each domain, an active application should be able to specify how much memory is dedicated to buffering of its incoming packet streams, versus storing code and program state.

Janos must also control the amount of CPU time used by individual domains. This goal has two distinct subgoals: first, for each domain, to guarantee and limit access to CPU time; and second, to reduce, amortize, or eliminate CPU time that cannot properly be charged to any domain. In regulating CPU resources of individual domains, our initial goal for Janos is to support simple relative-share CPU reservations. Support for scheduling according to more complex service guarantees—such as latency guarantees or real-time deadlines—is beyond the scope of the current Janos implementation effort. Additionally, the Java components of Janos need to be able to constrain time spent in the garbage collector, or charge that time to the appropriate domain.

The third explicitly controlled resource is outgoing network bandwidth. Clearly, as a network router, Janos must manage and constrain each domain's access to the network. Similar to CPU scheduling, our initial goal for Janos is to provide simple relative-share outgoing rate guarantees to each domain. More complex scheduling parameters, such as latency requirements, are possible in the design, but are beyond the scope of our current implementation. As with the other resources, domains should be able to manage independent outgoing streams separately. Specifically, information about the packet queues on links must be provided so that intelligent packet drop decisions can be made.

In contrast to outgoing network bandwidth, for incoming bandwidth there is no direct way for a receiver to influence the schedule of incoming packets. Janos does not artificially limit a domain's incoming network usage. A domain should, however, be able to control the buffering for each input channel independently, and should be able to specify how packets destined to a "full" input channel are handled (by dropping them or by replacing older packets).

Given that Janos enforces limits on memory, CPU, and network bandwidth, active applications must be able to specify their resource requirements to Janos. Janos, in turn, must be able to describe resource availability and allocations to users. The goal of presenting an appropriate representation or "language" for resource descriptions is challenging. In an active network, resource specifications must be general enough to apply to a heterogenous set of nodes with unique hardware in multiple administrative domains, yet these descriptions must be precise enough to allow accurate accounting and admission control on an individual node. Our goal with Janos is to provide a simple and coarse system for reconciling high-level, abstract resource specifications with the low-level physical properties of nodes.

2.3 Performance Goals

Janos must perform at the level of similar solutions on a traditional operating system. While the prototype nature of Janos, the immaturity of Java optimization tools, and the inherent inefficiencies of stock PC hardware all present significant performance challenges, Janos should at least demonstrate that flexible active nodes can be fast and efficient for appropriate uses.

In fact, there is considerable reason to believe that Janos will offer good performance to active applications. Because Janos is designed specifically to service active network applications, it can expose domain-specific optimizations to applications—optimizations that would not generally be possible in a conventional operating system. For example, the nature of a network interface in a router is much more constrained than in a general purpose operating system. We expect that the difference between control and data streams can be exploited for performance gains. Additionally, in an active network router network traffic will generally be manipulated as discrete, physical-layer-sized objects. Abstracting network access behind the stream-oriented, physical-layer-independent network interfaces found in traditional operating systems prevents active applications from making informed decisions about the network around them.

2.4 Separable Components

A final goal of our Janos effort is that, although the major components are optimized to work together, each should also be useful independently. This maximizes the impact of our research by providing infrastructure to other designers and active network researchers. Java-based EEs should be sufficiently portable to run on the JanosVM, while other EEs written to the NodeOS API should be easy to port to Moab.

To this end, three major components of Janos provide or implement well-defined programming interfaces oriented around the NodeOS API. The ANTSR runtime relies on

Java bindings to the NodeOS API, while the JANOSVM relies on the C implementation and provides the Java bindings. Moab provides an implementation in C. Figure 1 shows the relationships between the components of Janos.

3 Design

Janos takes advantage of a wealth of existing infrastructure and simultaneously provides new facilities and features. Most of the existing infrastructure in Janos is obtained from two software projects: the OSKit [19] component libraries and Kaffe [45], a freely available Java virtual machine. The OSKit is a collection of components such as device drivers, POSIX APIs, filesystems, and network protocols, culled from existing systems and made interoperable. Kaffe is a complete, open source Java virtual machine. Kaffe running on the OSKit provides a basic Java implementation on raw hardware: the infrastructure upon which we architect Janos.

Janos is one of a new wave [6, 7, 21, 40] of hybrid language/OS systems. As with these systems, Janos leverages the type-safety of Java to provide memory safety and to allow safe, fine-grained sharing across the user/kernel boundary. Unlike traditional operating systems, Java operating systems do not use separate virtual address spaces or system call traps to separate applications and the kernel, instead using type-safety to enforce protection and separation [8]. All untrusted code to be run on Janos must be written in Java; C is used only for implementing trusted portions of the system. Janos differs from previous Java operating systems in its customization for the active network domain, and in its emphasis on resource management.

We divide the description of Janos's design among the three major components: Moab, the JANOSVM, and ANTSR. For each component, we describe the architecture and explain its contribution to Janos.

3.1 Moab

Moab is compliant with the DARPA Active Network Program's NodeOS specification [1]. As such, Moab's implementation is somewhat determined by the major features defined in the specification. Our initial work on Moab helped to inform and shape the NodeOS API specification.

Moab is built upon the OSKit, from which Moab gets a full complement of device drivers (originally from Linux), several complete filesystems, a threading implementation, the FreeBSD networking stack, and many support modules such as boot loaders and remote debugging support. One benefit of building on the OSKit is that POSIX-reliant systems will work almost immediately on the OSKit, and can be incrementally migrated to Moab. The NodeOS API is implemented as a library layered on top of the OSKit's libraries. Moab is not implemented as a traditional operating system; invocations of NodeOS functions are direct function calls and do not "trap" into the OS. (Janos protects itself from untrusted code at the JANOSVM layer, not in the NodeOS.) Moab, like the OSKit, is written in C.

We do not fully address the motivation or design of the NodeOS API in this article, and will only cover those areas where Moab departs from the specification, or provides services beyond the scope of the specification. A treatment of the issues involved where Moab follows the NodeOS specification are covered by another article in this journal [31].

Before explaining how Moab differs from the specification, we present a (very) short review of the NodeOS API. The *domain* is the unit of resource control. Each domain is associated with a *memory pool* of physical memory pages. A domain contains *thread pools* from which *thread* objects are taken to handle *packets* dispatched out of *input channels*, based on the *demultiplex key* specified with the channel. Packets are sent on *output channels*. A *cut-through channel* allows the NodeOS to optimize directly connected input and output channels.

Currently, Moab deviates from the NodeOS specification (but, ideally, not from future specifications) in several minor ways, including two incompletely implemented NodeOS interfaces (memory pools and events). We plan to fill out our implementation to satisfy the NodeOS API in these areas. There are several areas, however, where we have deliberately diverged from the specification for reasons described in the following sections. Specifically, our changes concern: our single trusted-EE assumption, hardware-specific resource specifications, a constrained packet buffer interface, and threadpool associations. Further, Moab compatibly extends the NodeOS API with a new type of cut-through channel based on Click [28].

3.1.1 Single, Trusted Execution Environment

Where the NodeOS specification leaves open the ability of a NodeOS to host multiple execution environments, Moab assumes it will play host to a single EE. This design decision is based on the assumption that a "production" active network node will contain a single EE tailored to its environment. Our single-EE focus means that Moab does not need to mediate between EEs (for example, with respect to demultiplexing incoming packets).

Additionally, the NodeOS specification leaves open the trust relationship between the NodeOS and an EE. In Moab we chose to implicitly trust the EE to perform many services correctly. We believe the boundary between trusted and untrusted code belongs at the AA/EE boundary and that duplicating safety and separation at the EE/NodeOS boundary is wasteful. Specifically, many security and correctness checks are better accomplished by the type-safety of the Java runtime than by explicit checks in Moab.

3.1.2 Resource Specification

The current NodeOS specification calls for fairly imprecise, hardware-independent resource specifications. In contrast, Moab resources are specified in a very precise manner in terms of the local node's hardware. For example, physical memory limits are specified in terms of memory pages, CPU rates are specified in units of processor cycles per second, and outgoing network bandwidth is specified in bytes per millisecond. It is the job of the EE (in our case, ANTSR and the JANOSVM) to present these hardware-specific controls to application programmers and node administrators in a meaningful fashion.

This design puts the burden of developing hardware-independent resource specifications at the more flexible EE level. Also, with our design, EEs can make informed decisions about the local node based on precise numbers that map directly and clearly to local resources. Reporting the actual resource usage on a local node as hardware independent values would reduce the confidence that execution environments have in their resource allocations. Thus, Moab is useful to a broader array of EEs, including EEs that want to precisely distinguish "EE" resources from "AA" resources.

3.1.3 Memory Accounting

To meet the goal of allowing applications on Janos to manage their own memory usage, we have designed Moab so that almost all memory management can be performed by the EE. That is, all memory used by Moab on behalf of the EE is provided explicitly by the EE. For example, creating a thread requires the EE to provide memory for the thread control block and memory for the thread's stack. This "zero-malloc" interface design, which has been incorporated into the most recent NodeOS specification, is also intended to improve performance by removing memory management operations from critical paths. Dynamic memory allocation (plus the implied deallocation) in a multi-threaded system can be costly. While there are many approaches to reduce the cost of allocation [24, 36], avoiding allocations removes this overhead and results in code that is simpler to analyze, and that by definition cannot throw out-of-memory exceptions.

Unfortunately, while all of the Moab code meets this goal, we are reusing a large amount of existing code in the OSKit that is not necessarily written in this style, such as the device drivers, filesystem code, and primitive thread library.

3.1.4 Packet Buffers

Packets are special in Moab. The design of packet buffers is oriented around a single problem: the NodeOS must receive an incoming packet into memory before deciding which domain owns the packet. Moab is designed to implement zero buffer copies in the common case of forwarding a packet, even if untrusted user code makes the forwarding decisions and manipulates the packet. Compare this with a traditional operating system model in which data must be copied in order to protect the kernel from untrusted applications.

To avoid copies, we require that a packet be received into a buffer that can be handed directly to the appropriate domain. An EE associates buffers with each of its input channels. To maintain resource limits, a buffer is only handed off to a domain that has a free buffer available on its input channel. Moab swaps the "full" buffer for the "empty" one. Thus Moab has a constant supply of buffers for incoming packets, and the domain neither gains nor loses buffers. Of course, this design requires that all buffers be interchangeable. For example, the system would quickly break down if an "empty" 10-byte buffer were equivalent to a "full" 4096-byte buffer. Buffers must be big enough to receive large packets, so all buffers must be the maximum transmission size of the local node's physical layers.

In a traditional operating system, this design would probably be overwhelming. In a router, however, we postulate that reassembly of disjoint packets is rare (i.e., that contiguous, physical-layer packets are the most common type of packet). This assumption provides an opportunity for Moab to impose a reasonable restriction that results in a simpler, zero-copy dispatch of packets to domains. Additionally, by fixing the size of packet buffers, management of packet buffer caches is greatly simplified.

We anticipate, however, that creating full-size buffers for all packets will waste a significant amount of memory on the many small packets that can be expected. Because packet distributions tend to be bimodal [12], and since copying small packets is cheap, an EE may choose to immediately copy the packet into a smaller buffer and immediately recycle the large buffer.

Moab provides no direct support for hierarchical packet representations made from multiple buffers (such as mbuf-chains [25] or x-kernel msgs [29]). The outgoing channel send interface, however, supports a gather interface similar to `writev()` that takes either a variable length argument list of (*buffer*, *offset*, *length*) tuples or a pointer to a list of such tuples. The exact structure and maintenance of complex buffer representations is left to the higher levels of Janos. We justify this design with three observations. First, the common case for packet handling in a NodeOS should be physical layer packets that, by definition, fit in a single buffer. Second, any complex packet structure at the NodeOS level is imposed on higher layers, regardless of their needs. Third, our design supports the major performance optimization that comes from more complicated hierarchical packet representations, namely the gather operation performed on packet transmit.

3.1.5 CPU Accounting

Moab deviates from the NodeOS thread pool specification to meet our goal of allowing users of the NodeOS to manage CPU usage within a domain. Instead of a domain-wide thread pool, in Moab thread pools can be bound to specific input channels. This allows the domain to intelligently subdivide its CPU resources among its input channels. This can be useful in prioritizing control and data packets, or for giving priority to one physical interface over another.

Another aspect of precise CPU accounting in a router is the difficulty in charging for the time spent processing incoming packets. Specifically, resources are required to calculate that an incoming packet belongs to no domain in the system. Moab does all it can to minimize this cost by implementing the simple packet dispatch mechanism described earlier and additional, well-known techniques [16, 37]. But in the end, Moab charges such CPU usage to the system.

3.1.6 Memory Pools

Memory management in the NodeOS is not performed at domain granularity, but instead over groups of domains. Each domain belongs to exactly one *memory pool*, from which all the memory for that domain is allocated. The limit associated with a memory pool is the sum of the limits of all the domains associated with that pool. Thus, when a domain is terminated, the overall memory pool loses one domain's worth of memory.

This design matches the memory management requirements of the JANOSVM and leverages its existing infrastructure. To enable the sharing of code and data between AAs within the JANOSVM, the VM needs to be able to charge memory to a set of domains. To that end, the VM creates a single NodeOS memory pool for itself and all Java-based domains. The JANOSVM already must do memory accounting within its garbage collector, so putting per-domain memory management at that level avoids redundancy. Memory management in the JANOSVM is further described in Section 3.2.

The current memory management interfaces (at all levels) deal with memory as a guaranteed, strictly accounted resource. Opportunistic and revocable uses of memory (e.g., a domain using available memory for caching) are not covered in the current APIs.

3.1.7 Click Channels

The Click Modular Router [28] is a software package from MIT that enables administrators to dynamically create Linux-based packet routers by wiring together small, simple, well-defined components that perform such functions as Ethernet receive, IP checksum, and so on. Click router graphs match the semantics of NodeOS cut-through channels.

Moab supports Click-channels, in addition to the standard NodeOS API channels. Click-channels use Click router graphs in place of the simplistic protocol specifications defined by the NodeOS specification. For example, a Click cut-through channel contains a Click router graph description in place of a protocol specification. The instantiated Click router elements run inside Moab. The Click channel specifications are more complex, but features such as reassembly, timeouts, and buffering are specifiable.

3.2 JanosVM

The JANOSVM is the most critical part of a Janos node. This is where we map the C-based Moab interfaces into Java and provide ANTSR with support for managing untrusted, potentially malicious, user applications.

The JANOSVM is a virtual machine that accepts Java bytecodes and executes them on Moab. Both the VM and the Java code running within it use the facilities provided by Moab. The JANOSVM provides access to the underlying NodeOS interfaces through the Janos Java NodeOS bindings, which wrap simple Java classes around the C-based API. In terms of resource controls, the CPU and network controls available in Moab are unchanged by the JANOSVM. Per-domain memory limits, however, are enforced by the garbage collection mechanism outlined below.

The design of the JANOSVM followed directly from our prior experience in Java operating systems with the Alta Operating System [40] and KaffeOS [6]. Foremost, the JANOSVM is based directly on the KaffeOS implementation. KaffeOS is a Java virtual machine that provides the ability to isolate applications from each other and to limit their resource consumption. The KaffeOS architecture supports the OS abstraction of a *process*—a domain—in a Java virtual machine. Each process executes as if it were run in its own virtual machine, including separate garbage collection of its own heap. KaffeOS uses *write-barriers*—compiler-inserted checks on certain object field writes—to prevent a process from writing outside its own heap. Through this architecture and careful engineering, CPU and memory resources, including indirect usage such as for JIT'ed code blocks, can be controlled on a per-process basis. We have demonstrated that KaffeOS achieves effective resource control with low overhead [6].

KaffeOS implements a general OS architecture. In Janos, we have simplified the KaffeOS design to leverage the constraints of our active network target. In particular, we introduce a more restrictive process model. The major difference is that the JANOSVM does not support KaffeOS shared heaps; thus the write barriers can be somewhat simpler and less frequent in the JANOSVM. Like KaffeOS, the JANOSVM supports multiple, separate heaps, separate garbage collection threads for each heap, per-heap memory limits, and all of the basic JVM features (JIT compilation, reflection, etc.). The JANOSVM is implemented in a mix of C and Java.

The JANOSVM by itself is not a complete EE; although it supports a type-safe environment through Java, it also exposes many interfaces that untrusted code cannot safely be allowed to invoke. A Java-level runtime (ANTSR in our case) is required to present AAs with restricted access to the NodeOS abstractions and provide services such as protocol loading. The importance of drawing a "red line" [8] between trusted system code and untrusted code in language-based operating systems was identified in our prior work with Java operating systems.

As we cannot describe all of the JANOSVM in this short article, we focus on four aspects: the strict separation of domains enforced by the JANOSVM, the special handling of packet buffers, the specification of resources, and the thin wrappers of the NodeOS API.

3.2.1 Strict Separation of Flows

To meet our goal of hosting untrusted, potentially malicious Java bytecode, the JANOSVM implements a strict separation of domains. Each domain runs in its own namespace and in its own heap. Namespace separation is achieved by a ClassLoader, the standard Java namespace control mechanism. The separate heaps are provided by our domain-aware garbage collector in the JANOSVM. The only shared objects permitted between domains are packet buffers.

The JANOSVM provides each domain with its own heap and a separate garbage collection thread for cleaning that heap. In addition to separating the memory usage of each domain, separate heaps implicitly constrain the garbage collection costs incurred by each domain. Internally, the JANOSVM's allocator groups similar-sized objects together on each "page" (4096 bytes, currently), which can cause fragmentation of memory. Thus, to eliminate inter-domain fragmentation attacks, the JANOSVM charges whole pages to a domain. Maintaining memory ownership on a per-page basis greatly simplifies memory reclamation upon domain termination as the JANOSVM can sweep whole pages into the free page list.

The strongly enforced separation between domains resolves the difficult problem of fine-grained sharing in a type-safe system that supports termination. By disallowing objects to be shared between domains (except for the special case of packet buffer objects), we can avoid all the overhead of write-barriers in user code (as in KaffeOS [7]) or complex compiler and run-time support (as in Luna [22]).

As noted in Section 3.1.6, the JANOSVM uses a single Moab-managed memory pool for all domains. The JANOSVM itself enforces per-domain memory limits. When a domain is terminated, the JANOSVM has to return the same number of pages to Moab as it acquired when the domain was created. The exact memory pages that were provided, however, need not be the ones that are returned.

While total separation of domains makes resource management and domain termination simpler, it also makes the system less flexible. So, the JANOSVM provides for limited inter-domain communication. We believe inter-domain communication will be rare, and have provided few concessions to it. To support inter-domain communication, the JANOSVM uses a mechanism similar to a tuple-space. All objects (except for packet buffers) are deeply copied through this interface, thus preserving the separation of heaps. Because access to packet buffer memory is already specially handled by the JANOSVM (see Section 3.2.2), sharing packet buffers between domains does not create additional complications in the garbage collector.

Although Janos supports only a limited form of inter-domain communication, active applications can efficiently communicate across the user/kernel boundary to the JANOSVM and Moab. For example, operations such as sending packets, manipulating a thread, and receiving a buffer are accomplished with a minimum of overhead through a fairly rich interface.

3.2.2 Packet Buffers

In the JANOSVM, packet buffers are wrappers around Moab's buffer abstraction. The JANOSVM does not fundamentally change the buffer abstraction provided by Moab. Buffers are presented as simple contiguous chunks of memory. No higher-level organization of buffer objects is supplied; that is left to the ANTSR runtime and its active applications.

Access to buffers from Java is indirect through the wrapper object, allowing the JANOSVM to revoke access to buffer objects from AAs and to separately manage and account for buffer memory. This design effectively moves the user/kernel boundary across which packets are traditionally copied, all the way up to the AA's buffer access interface. Thus, for domains that just accept and send their packets, there is no boundary crossing at all. Applications that simply read and manipulate fields in a header pay only for those operations. Sequential processing of large contiguous chunks of the buffer, as would be the other common case for active protocols, is amenable to compiler optimizations that amortize the indirection cost over each contiguous buffer chunk. Using such techniques, several Java projects [22, 41] have shown how to remove most of the overhead introduced by wrapper objects such as that introduced by the JANOSVM.

Buffers are sharable between domains, though the memory cost is always borne entirely by a single domain (the *owner* of the buffer). As described in [7], alternative shared object cost models are not viable. A domain may accept the cost of a buffer from another domain, effectively transferring ownership. If the owner is terminated or exits, all of its buffers are revoked and reclaimed. This cleanup is made possible by the same layer of indirection on buffers that protects the NodeOS.

3.2.3 Resource Specifications

As discussed in Section 3.1.2, Moab provides hardware-specific resource information and leaves the job of mapping portable resource specifications to the EE. In Janos, that task is performed by the JANOSVM which will provide a library for mapping platform independent resource specifications into Moab's hardware-specific specifications.

The actual API for the platform independent specifications is not yet complete, but will be expressed along the lines of a multiple of "IP packet forwarding cost" for CPU usage specifications and a multiple of "MTU packet size" for memory resource specifications.

3.2.4 Thin Wrappers

For most NodeOS API features and interfaces provided by Moab, the JANOSVM maps the NodeOS abstractions into Java with minimum overhead. For example, the NodeOS channel APIs are presented in Java as `OutChannel`, `InChannel`, and `CutThroughChannel` classes with the same operations that are available in Moab. In addition to mapping the API directly, the JANOSVM supports the same memory accounting design as Moab, making memory allocations as restricted as possible and keeping the critical path free of memory management.

3.3 ANTSR

The ANTSR Java runtime is based on ANTS 1.1 [42, 43] and provides the interfaces for untrusted, potentially malicious, AAs to interact with the system. This is the layer that is responsible for hiding critical JANOSVM interfaces and specifying per-domain resource limits. ANTSR is written entirely in Java.

ANTSR supports active packet streams, where code is dynamically loaded on demand when packets for a new stream arrive. Demultiplexing of incoming packets is implicitly defined by the signature (an MD5 hash) of the bytecode making up the protocol. This elegantly solves packet spoofing and snooping problems because the code implicitly defines the type of packets that can be sent and received.

ANTSR differs internally from ANTS, in that ANTSR is designed to take advantage of the NodeOS abstractions and the support provided by the JANOSVM. Under the hood, ANTSR adds many significant new features including domain-specific threads, separate namespaces, improved accounting over code loading, and a simple administrator's console. Despite these changes, ANTSR is featurewise equivalent to ANTS, and the recent standard ANTS 2.0 release is based on ANTSR.

Our experience with rewriting ANTS to use the NodeOS APIs was very encouraging. The NodeOS abstractions provide exactly the services that ANTS contained ad-hoc implementations of. In many cases, the NodeOS/EE distinction that we necessarily imposed on ANTS cleared up internal abstractions that were confusing. For example, the demultiplex key support in the NodeOS is a perfect fit with the ANTS model of prefixing all user packets with a 16-byte hash identifying the protocol.

More difficult than simply using the available APIs was the task of separating the resource usage of different domains in ANTS. For example, one of our goals with ANTSR was to correctly account for the cost of downloading code from a previous node. In ANTS this was effectively a system-provided service: a global table kept track of what code had been downloaded, and requests for code were sent out based on that. Once all the code for a new protocol had arrived, ANTS started the new protocol. To correctly account for the cost of code downloading in ANTSR, we create a new domain early and make download and installing its code the first task of the new domain. This design implicitly restricts code loading resources to the new domain's limits.

Overall, our problems in converting ANTS to ANTSR arose from retroactively adding resource control to an existing system.

3.4 Review

In summary, Janos is designed to provide services and features in the appropriate layer, without overlap. Memory accounting is done at the domain level within the JANOSVM, as is the mapping from abstract to concrete resource specifications. Moab performs the management of CPU and network resources. The containment of untrusted code is performed at the highest level, in ANTSR. Together, these separate pieces implement a coherent whole.

4 Open Issues

The current design and implementation of Janos leaves some important topics unaddressed, such as the implementation of additional OS services that would be of use to certain active applications. We briefly describe some of these issues below and point out areas that we feel should be explored in the future.

Janos provides a relatively restricted model for the composition of active applications and for data sharing between applications. While we expect that the model is suitable for a wide varitey of practical applications, it remains to be seen how the model suits the needs of third-party developers in practice. Our belief is that a protocol composed of many components will typically resemble a composition of libraries rather than a composition of server and client processes. Because we have provided few concessions to inter-domain communication, building a system based on communicating domains will be difficult. Assuming a clean system is devised for naming and referencing libraries, Janos will be able to dynamically compose protocols and reuse shared code.

Beyond the mechanisms for defining and distributing active applications, Janos presents interesting issues in the area of Java code optimizations. First, there are many general optimizations for Java Virtual Machines that would be applicable and useful to the JANOSVM—a list not limited to dynamic recompilation [5], lock eliding [9, 10, 14, 44], method inlining, stack allocation [34], and thread-local heaps. More specific to Janos, however, are issues involving compilation techniques for handling special data structures such as packet buffers. Since an active application may be untrusted, it cannot generally be given direct (i.e., unmediated and unrevocable) access to any unprotected system resource, including packet buffers (Section 3.1.4). The JANOSVM must therefore interpose on access to packets, though this adds overhead to a potentially critical path in the system. One solution is to incorporate new analyses into the JANOSVM JIT compiler to recognize when duplicate access checks can be eliminated, and to produce appropriate code that optimizes access checks. We intend to implement these JIT compiler enhancements in future work.

Because the JANOSVM imposes strictly enforced memory limits, and because memory cannot be cleanly revoked from running applications, excess physical memory in a system goes unused. Ideally, however, applications should be able to take advantage of unreserved memory in the system for caching or other opportunistic uses. We will be developing a new API for domains to maintain data in a weakly held store. The fundamental characteristic of this memory is that the system can trivially reclaim it without cooperation from the domain, similar to the semantics of weak references in garbage collected systems [24].

A final issue is in providing additional operating system services that may be needed by some active applications. ANTSR currently provides facilities for storing non-persistent state at a node, but some protocols may require persistent state. Moab currently supports access to persistent storage on a node, though the API presented is merely the existing POSIX-derived file system APIs. Persistent storage would be useful not only to active applications, but also to Janos itself. For instance, Moab should be able to store its configuration information and other state required for a quick, stand-alone reboot of the node, and the JANOSVM/ANTSR execution environment should be able to cache dynamic code, security policies, and other EE-level state. Providing active applications with access to persistent storage creates a new resource that needs to be closely managed, but opens up opportunities for dynamic, protocol-specific caching in the network.

5 Results

5.1 Janos Component Status

Each of the major components of Janos is currently available and usable. Initially developed as entirely separate software components, many subsets now work together, in multiple configurations, and the whole set will soon be distributed as a complete system. There are six separate software components in a complete Janos node: the ANTSR runtime, the Janos Java NodeOS bindings, the JANOSVM, Moab, the Click modular router, and the OSKit. All Janos components are fully open source and available at the Janos Web site [18].

Moab supports threads, domains, the filesystem API, events, proportional share scheduling of CPU and network bandwidth, and raw Ethernet, UDP, TCP, and Click-based channels. Moab can be configured to run on FreeBSD, Linux, Solaris, or—when linked against the OSKit—can run in kernel mode on bare hardware. Some Moab resource controls and features rely on OSKit features unavailable in stock Unix, so are only available in the OSKit configuration.

The JanosVM is running, and initial experience supports our design choices regarding packet buffers and the restricted heap model. The Janos Java NodeOS is the component of the JanosVM that maps the NodeOS API from C into Java. This component also works without Moab, on standard JVMs, by mapping the NodeOS API calls onto the standard Java APIs.

Our distribution of AntsR includes a large regression testing suite, six example protocols, DANTE support, and a privileged routing table protocol. AntsR currently runs all of its example applications on the JDK-compatible version of the Janos Java NodeOS. When run on the JanosVM with Moab and the OSKit, AntsR runs many of the applications.

5.2 Evaluation

5.2.1 Separable Components

Our goal of building separately usable components is meeting with initial success as other active network research projects have started using our components. As mentioned above, Ants 2.0 is based on AntsR and the standard JVM version of the Janos Java NodeOS. Additionally, Network Associates' AMP project [35] is currently using the Janos Java NodeOS and plans to use the JanosVM, as does Princeton's Scout-based NodeOS project [31]. Finally, the University of Kentucky's CANEs EE [26] is actively being ported to Moab.

5.2.2 Metrics

Forwarding Path	Rate (Kpps)
OSKit	75.7
Moab cut-channel	48.7
C-based EE on Moab	45.0
Java-based EE on Moab	19.5

Table 1: Packet forwarding rates at various levels in Moab, in thousands of packets per second.

A more detailed description of the raw performance of Moab is available in the NodeOS comparison article [31] elsewhere in this journal. Table 1 summarizes the results of forwarding minimum-sized IP packets with a minimum of processing on each packet through the raw OSKit, through a Moab cut-channel, through a Moab EE, and through a Janos Java NodeOS-based EE, on a 600MHz Intel Pentium III with five Intel EtherExpress Pro/100+ PCI Ethernet cards.

Share	Measured throughput (Bps)
1	2,053,608
2	4,094,836
3	6,145,442

Table 2: Relative output bandwidth usage for resource limited out-channels trying to independently saturate a single physical link. The measured throughput is reported in bytes per second.

Table 2 shows the results for a test of relative share scheduling on a single congested link. In this test, each channel is sending 1024-byte packets as fast as it can (slightly faster than the link can handle, in fact). The 1:2:3 relative share ratio corresponds to 1/6, 1/3, and 1/2, respectively of the total resource. The measured throughput closely matches the relative share assignments. Note that 12.3 MBps (the total of all shares) is the maximum physical bandwidth of the link.

In terms of memory controls, the JanosVM controls per-domain memory usage as effectively as KaffeOS [6], and with less overhead due to the reduction in write barriers. Due to the "zero-malloc" implementation of Moab, the majority of a domain's memory is allocated within the scope of the JanosVM. This greatly simplifies our design because enforcement and accounting can be colocated in the JanosVM. However, for memory allocated within the OSKit, Moab does not yet reconcile those allocations with the per-domain limits. Since the OSKit-level allocations are not directly accessible to the user, only a strictly bounded amount of memory can be used this way.

Despite the additional layer of the Janos Java NodeOS, AntsR, when run on standard JDKs, performs comparably to older versions of Ants. This reinforces our earlier claim that Ants already contained ad-hoc implementations of NodeOS-like abstractions. Still, AntsR has many obvious optimizations that must be completed to fully take advantage of the lower layers of Janos. Specifically, AntsR alone contains at least four copy operations per packet in the common case. First, a copy is made to switch buffer object representations; this copy should be relatively easy to remove. The second is more subtle, because AntsR (and Ants before it) creates a new Java object to represent the packet and de-serializes the buffer into this object. This process is reversed when a packet is sent. When run in user mode, there is of course the additional copy of the data into and out of the process's address space. As noted in Section 3.3, changing Ants to accept resource controls, and do so efficiently, may require changes to its model.

These initial results give us confidence that we have designed a system that supports resource management at the appropriate layer, and that the fully-assembled system will function well.

6 Related Work

Research in active networks is ongoing and lively. We organize the work related to Janos into three groups. We list other node operating systems, other execution environments, and finally other Java operating systems.

6.1 Node Operating Systems

There are currently four major NodeOS implementations: our Moab, Princeton's Scout-based system, Network Associates' exokernel-based AMP system [17, 35], and the Bowman NodeOS [27] from Georgia Tech and Kentucky. While Moab focuses on resource control and domain management, the Princeton system integrates NodeOS abstractions in Scout's paths [30], and AMP focuses on security. An in-depth comparison of the related aspects of these three NodeOS implementations appears elsewhere in this journal [31]. Bowman predates the DARPA NodeOS specification, but contains many of the same abstractions and features. Bowman relies on POSIX interfaces and runs in user mode on Solaris and Linux.

6.2 Execution Environments

As noted earlier, our ANTSR execution environment is directly derived from ANTS 1.1, and ANTS 2.0 is derived from ANTSR.

The ASP EE [11] from USC-ISI is a Java-based EE targeted to signaling applications in the network, giving it a slightly different focus than ANTS and ANTSR. Also, the ASP developers have provided a more concrete specification of its programming API [32]. ISI's ASP developers are porting ASP to Janos's Java NodeOS bindings.

There are several other EEs that focus on active networking issues such as security (e.g., AMP [35]), administration, or protocol development (e.g., CANEs [26]). Ideally, the ideas and interfaces from those projects could be folded into ANTSR.

SwitchWare [3] is a project that, like Janos, encompasses all aspects of a node between the active code and the hardware. Unlike Janos, however, in SwitchWare's PLANet all packets contain their forwarding code in place of traditional headers. PLAN [4, 23] is based on the Caml [13] programming language. The SwitchWare developers have said they intend to port their PLAN execution environment to Moab.

6.3 Java Operating Systems

The JANOSVM is a Java operating system—a Java language runtime that supports OS abstractions—and builds upon several previous projects in this area. The JANOSVM is the fourth in a line of Java OS's we have designed and implemented at Utah, each exploring a different part of the design space, especially in terms of the class and memory sharing model. The JANOSVM directly uses the multiple Java heap implementation from the K0 [7] and KaffeOS [6] systems, and as they do, builds on the base Kaffe [45] Java virtual machine. We also drew experience and insight from Cornell's J-Kernel [21] and our Alta [40] system. All except the J-kernel support a more general process model than Janos, allowing direct sharing between processes, although they all support such sharing differently and with different restrictions. Janos, on the other hand, provides a more restricted process model that is customized for the active network domain.

All of these systems build on early work in language-based operating systems [33, 38, 46]. Inferno [15] is such a system that is unusual for its attention to efficient automatic memory management through a combination of reference counting and a cycle-collecting garbage collector. None of these pioneering systems, however, provide the domain separation and resource control, nor the orientation to network protocol processing, that has been the focus of our work.

7 Conclusion

We have described the architecture of the Janos active node operating system and its rationale. Janos exploits a custom JVM and resource-aware operating system and runtime layers to provide strong resource controls and accounting over all active applications on a single node in an active network. In doing so, Janos provides both of the major software layers currently defined as the canonical active node's infrastructure: the NodeOS and execution environment.

Acknowledgments

We would like to thank the anonymous reviewers for providing detailed, critical feedback, and our shepherd, Ken Calvert.

References

[1] Active Network NodeOS Working Group. NodeOS interface specification. Available as http://-

www.cs.princeton.edu/nsg/papers/nodeos.ps, Jan. 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. S. Alexander, W. A. Arbaugh, M. W. Hicks, P. Kakkar, A. D. Keromytis, J. T. Moore, C. A. Gunter, S. M. Nettles, and J. M. Smith. The SwitchWare active network architecture. *IEEE Network, Special Issue on Active and Programmable Networks*, 12(3):29–36, 1998.

[4] 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.

[5] B. Alpern, C. R. Attanasio, J. J. Barton, M. G. Burke, P. Cheng, J.-D. Choi, A. Cocchi, S. J. Fink, D. Grove, M. Hind, S. F. Hummel, D. Lieber, V. Litvinov, M. F. Mergen, T. Ngo, J. R. Russell, V. Sarkar, M. J. Serrano, J. C. Shepherd, S. E. Smith, V. C. Sreedhar, H. Srinivasan, and J. Whaley. The Jalapeño virtual machine. *IBM Systems Journal*, 39(1):211–238, 2000. Reprint Order No. G321-5724.

[6] G. Back, W. C. Hsieh, and J. Lepreau. Processes in KaffeOS: Isolation, resource management, and sharing in Java. In *Proceedings of the Fourth Symposium on Operating Systems Design and Implementation*, pages 333–346, San Diego, CA, Oct. 2000. USENIX Association.

[7] G. Back, P. Tullmann, L. Stoller, W. C. Hsieh, and J. Lepreau. Techniques for the design of Java operating systems. In *Proceedings of the USENIX 2000 Annual Technical Conference*, San Diego, CA, June 2000. USENIX Association.

[8] G. V. Back and W. C. Hsieh. Drawing the red line in Java. In *Proceedings of the Seventh Workshop on Hot Topics in Operating Systems*, pages 116–121, Rio Rico, AZ, Mar. 1999. IEEE Computer Society.

[9] B. Blanchet. Escape analysis for object oriented languages. Application to Java. In *Proceedings of the 1999 ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '99)*, pages 20–34, Denver, CO, Nov. 1999.

[10] J. Bogda and U. Hölzle. Removing unnecessary synchronization in Java. In *Proceedings of the 1999 ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '99)*, pages 35–46, Denver, CO, Nov. 1999.

[11] B. Braden, A. Cerpa, T. Faber, B. Lindell, G. Phillips, and J. Kann. Introduction to the ASP execution environment. Technical report, USC/Information Science Institute, Feb. 2000. Available from http://www.isi.edu/active-signal/ARP/.

[12] C.A.D.I.A. Packet sizes and sequencing. Available at http://www.caida.org/outreach/learn/packetsizes/.

[13] Caml language Web site. http://pauillac.inria.fr-caml/.

[14] J.-D. Choi, M. Gupta, M. Serrano, V. C. Sreedhar, and S. Midkiff. Escape analysis for Java. In *Proceedings of the 1999 ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '99)*, pages 1–19, Denver, CO, Nov. 1999.

[15] S. Dorward, R. Pike, D. L. Presotto, D. Ritchie, H. Trickey, and P. Winterbottom. Inferno. In *Proceedings of the 42nd IEEE Computer Society International Conference*, pages 241–244, San Jose, CA, Feb. 1997.

[16] P. Druschel and G. Banga. Lazy receiver processing (LRP): A network subsystem architecture for server systems. In *Proceedings of the Second Symposium on Operating Systems Design and Implementation*, pages 261–275, Seattle, WA, Oct. 1996.

[17] D. R. Engler, M. F. Kaashoek, and J. O'Toole Jr. Exokernel: An operating system architecture for application-level resource management. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, pages 251–266, Copper Mountain, CO, Dec. 1995.

[18] Flux Research Group. The Janos project Web site. http://www.cs.utah.edu/flux/janos/.

[19] 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, Oct. 1997.

[20] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*. The Java Series. Addison-Wesley, 1996.

[21] C. Hawblitzel, C.-C. Chang, G. Czajkowski, D. Hu, and T. von Eicken. Implementing multiple protection domains in Java. In *Proceedings of the USENIX 1998 Annual Technical Conference*, pages 259–270, New Orleans, LA, June 1998.

[22] C. Hawblitzel and T. von Eicken. Tasks and revocation for Java (or, hey! you got your operating system in my language!). Submitted for publication, Nov. 1999.

[23] M. Hicks, P. Kakkar, J. T. Moore, C. A. Gunter, and S. Nettles. PLAN: A packet language for active networks. In *Proceedings of the Third ACM SIGPLAN International Conference on Functional Programming (ICFP '98)*, pages 86–93, Sept. 1998.

[24] R. Jones and R. Lins. *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*. John Wiley & Sons, 1996.

[25] S. J. Leffler, M. K. McKusick, M. J. Karels, and J. S. Quarterman. *The Design and Implementation of the 4.3BSD Unix Operating System*. Addison-Wesley, Reading, MA, 1989.

[26] S. Merugu, S. Bhattacharjee, Y. Chae, M. Sanders, K. Calvert, and E. Zegura. Bowman and CANEs: Implementation of an active network. In *37th Annual Allerton Conference on Communication, Control, and Computing*, Monticello, Illinois, September 1999.

[27] 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, Mar. 2000.

[28] R. Morris, E. Kohler, J. Jannotti, and M. F. Kaashoek. The Click modular router. In *Proceedings of the 17th ACM Symposium on Operating Systems Principles*, pages 217–231, Dec. 1999.

[29] D. Mosberger. Message library design notes. Available at http://www.cs.arizona.edu/xkernel/-message.ps, Jan. 1996.

[30] D. Mosberger and L. L. Peterson. Making paths explicit in the Scout operating system. In *Proceedings of the Second Symposium on Operating Systems Design and Implementation*, pages 153–167, Seattle, WA, Oct. 1996. USENIX Association.

[31] 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.

[32] G. Phillips, B. Braden, J. Kann, and B. Lindell. Writing an active application for the ASP execution environment. Technical report, USC/Information Science Institute, Feb. 2000. Available from http://-www.isi.edu/active-signal/ARP/.

[33] D. D. Redell, Y. K. Dalal, T. R. Horsley, H. C. Lauer, W. C. Lynch, P. R. McJones, H. G. Murray, and S. C. Purcell. Pilot: An operating system for a personal computer. *Communications of the ACM*, 23(2):81–92, 1980.

[34] A. Reid, J. McCorquodale, J. Baker, W. Hsieh, and J. Zachary. The need for predictable garbage collection. In *Second ACM SIGPLAN Workshop on Compiler Support for System Software*, Atlanta, GA, May 1999.

[35] S. Schwab and S. Rho. AMP security architecture and system design report. NAI Labs Technical Report, in preparation.

[36] O. Shivers, J. W. Clark, and R. McGrath. Atomic heap transactions and fine-grained interrupts. In *Proceedings of the Fourth ACM SIGPLAN International Conference on Functional Programming (ICFP '99)*, Sept. 1999.

[37] O. Spatscheck and L. L. Peterson. Defending against denial of service attacks in Scout. In *Proceedings of the Third Symposium on Operating Systems Design and Implementation*, New Orleans, LA, Feb. 1999. USENIX Association.

[38] D. C. Swinehart, P. T. Zellweger, R. J. Beach, and R. B. Hagmann. A structural view of the Cedar programming environment. *ACM Transactions on Programming Languages and Systems*, 8(4):419–490, Oct. 1986.

[39] P. Tullmann and C. Hawblitzel. *Janos Java Documentation*. University of Utah, Sept. 1999. Available from http://www.cs.utah.edu/flux/janos/.

[40] P. A. Tullmann. The Alta operating system. Master's thesis, University of Utah, 1999. 104 pages. Also available at http://www.cs.utah.edu/-flux/papers/tullmann-thesis-base.html.

[41] M. Welsh and D. Culler. Jaguar: Enabling efficient communication and I/O in Java. *Concurrency: Practice and Experience*, pages 519–538, Dec. 1999. Available from http://www.cs.berkeley.edu/~mdw/-proj/jaguar/.

[42] 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.

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

[44] J. Whaley and M. Rinard. Compositional pointer and escape analysis for Java programs. In *Proceedings of the 1999 ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '99)*, pages 187–206, Denver, CO, Nov. 1999.

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

[46] N. Wirth and J. Gutknecht. *Project Oberon: The Design of an Operating System and Compiler*. Addison-Wesley Publishing Company, 1992.