Role of Operating Systems in Active Networks

Goutham Morab
Computer Science and Engineering
University of Texas at Arlington

Abstract

Active Networks are an exciting and novel approach to network architecture in which the intermediate switches (routers) perform customized computations on the packets flowing through them. The packets contain both data and code. The code in the packets executes on active network nodes that support a generic execution platform. We explore the role of operating systems in Active Networks by presenting the design and implementation of the Janos Operating System for the Active Networks nodes. The primary focus of Janos is strong resource management and control of untrusted active applications written in Java.

Index Term - Active Networks, Programmable Networks, Operating Systems, High-level, Languages, Java

1. Introduction

Active Networks are very similar to traditional networks. Traditional networks just store and forward the packets whereas active networks store, compute, and forward the packets flowing through them. The basic idea is that the active networks have the ability to perform customized computations on the received packets. As an example, a user of the active network can send a "trace" program to the active nodes and arrange for the program to be executed when the user packets are received on the nodes. These active routers can also interoperate with the legacy routers which forward packets in a traditional manner. These networks are called active because each node can perform customized computations on, and modify, the packet contents. Also, this packet processing can be customized on a per-user or per-application basis. When compared to traditional packet switched networks, such as the internet, computation on the packet contents is very limited. The router examines and modifies only the packet header while the user data is passed opaquely without examination and modification. [5]

Defense Advanced Research Projects Agency community held many discussions in 1994 and 1995 on the future directions of the networking systems. These discussions gave birth to the concept of active networking. They also identified several problems with today's networks. They are shown below: [5]

- Difficulty in integrating new technologies and standards into the shared network infrastructure
- Poor performance due to redundant operations at several protocol layers
- Difficulty accommodating new services in the existing architectural model

Several strategies to address these difficulties emerged and these are collectively referred to as active network technology. The fundamental idea of packets carrying both procedures and user data can be seen as the next step towards extending the traditional packet and circuit switched networks. The computation of the packets requires the presence of an execution environment which is provided by the operating system running on the active nodes. The software in the active node can be described in terms of a model that divides the system into three logical layers. The three logical layers are the NodeOS, the execution environment (EE), and the active application (AA) layers.

NodeOS - This layer abstracts the hardware and provides low level resources management facilities.
Execution Environment – The EE accepts packets and programs that it deems valid. This sits on top of NodeOS and provides the basic application programming interface. There can be multiple non-interfering execution environment on the platform each providing a separate programming environment.

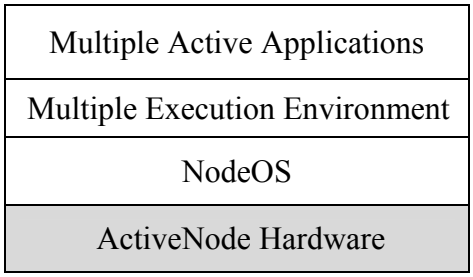

Figure 1: Active Network Model – Multiple Active applications running on multiple non-interfering execution environments supported by the NodeOS. [14]

This paper describes the role of operating systems in the active nodes with the help of Janos operating system which is primarily sponsored by DARPA with additional support from Novell, Compaq, and the University of Utah.
Janos is an operating system for the active nodes, implementing both the NodeOS and execution environment layers of the active network node model. The rest of the paper describes the components of Janos in greater detail, with emphasis on design that allow Janos to provide strong resource control over active applications running on the active network node. Section two gives general information about Janos Operating system. Section three presents the goals of the Janos architecture. Section four presents the system's design. Section five gives the summary of work related to Janos. Section six gives the open issues. Section seven gives the conclusion. [14] [9]

2. General Information about Janos

Janos is a Java oriented operating system for the active nodes. The primary focus is resource management and control, with secondary objectives of information security, performance, and technology transfer of broadly and separately useful software components.

Janos implements both the NodeOS and the execution layer (EE) of the active network model shown in figure 1. Active applications for Janos run on the ANTSR runtime system. ANTSR runtime runs on top of resource conscious Java Virtual Machine called JanosVM. ANTSR and JanosVM together constitute the execution environment layer of the active network model as shown below.

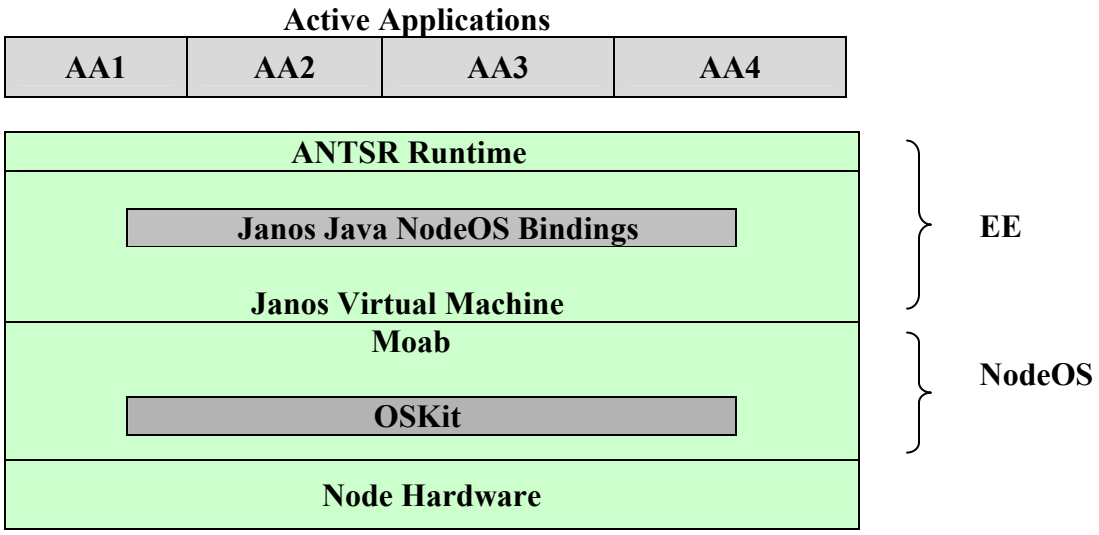

Figure 2: Janos Architecture and the Corresponding DARPA active network node architecture. Bottom Layer is the node hardware. Above it is the NodeOS layer called Moab. It is built on top of OS component library called OSKit. The execution environment layer is composed of JanosVM, Janos Java NodeOS, and ANTSR runtime. Active applications are executed on this platform. [14]

The active applications are supported by the JanosVM and ANSTR runtime. ANSTR provides a set of libraries. Simple mechanisms such as dynamic, on-demand code loading, implicit registration of packet matching keys, and dispatch of packets are provided to the active application programmer with the help of ANSTR API. [14] [9]
Janos is designed in such a way that separate active applications running on the node do not interfere with each other. Strong control over active applications' resource usage is given in the hands of the node administrator. The administrator of the Janos node must be able to control access to the node, assign sufficient resources, and query the node about its state. In Janos, the administrator is given control interface to the ANTSR runtime. With the help of this interface, he can start and terminate a domain (equivalent to a process in a conventional operating system), gather statistics about the node, change resource limits, and modify access privileges. [14]

Janos is built from three separately reusable components (Moab, JanosVM, and ANTSR). The important challenge in building Janos from separate components is to ensure that features are provided at the appropriate level and that there is no redundancy among the components. In addition to supporting active network users and administrators, Janos was designed to support research and development of other active network systems. [14][5]

3. Goals of the Janos Architecture

One of the major research focuses of Janos is to provide precise control over the execution of untrusted Java bytecode. In order to meet this demand effectively, Janos is obliged to provide sufficient security and sufficient performance. This section presents the goals of Janos and details of how these goals are met. [14]

A. Untrusted Code Support

As an active node operating system, Janos must support the execution of untrusted code. This calls for safety and security goals for Janos. The user code must be allowed to see only those packets that it is allowed to see. The user code must be able to send only those packets that it is allowed to send: it should not spoof packets. Also, to preserve resource constraints and timely dispatch of incoming packets, all user code must execute quickly or be preemptible. Most importantly, like in other OS, 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 have the capability to terminate the user code.

B. Resource Management Goals

Janos should control the use of resources by the active applications and should also be able to reclaim resources from terminated active applications. Resources control in Janos involves three independent resources: memory, central processing unit, and outgoing network bandwidth.
Memory – Janos has three management goals. First is to guarantee and enforce limit on each domain's memory requirements. Second is to reclaim memory from terminated domains. Third is to provide active applications with sufficient infrastructure to allow them to manage their memory internally.

Janos also has the responsibility of controlling the amount of CPU time by individual domains. This goal has two sub goals. First is to guarantee and limit access to CPU time. Second is to reduce, amortize, or eliminate CPU time that cannot be charged to any domain. 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 controlled resource is outgoing networking bandwidth. As a network router, Janos must manage and constrain each domain's access to the network. The domains should be able to manage independent outgoing streams separately. Janos cannot artificially limit a domain's incoming network usage. A domain should 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).

C. Performance Goals

There is considerable belief that Janos will perform well. This is because Janos is specifically designed to service active network applications. As a result, the nature of the network interface in a router is much more constrained than in a general purpose operating system. Thus, the difference between control and data streams can be exploited for performance gains.

D. Separable Components

The final goal of Janos is that each component should be useful independently. This increases improvements in research by sharing the infrastructure with other designers and active network researchers. Java-based execution environments should be sufficiently portable to run on the JanosVM, while other execution environments written to the NodeOS API should be easy to port to Moab.

4. Design

Janos is built by making use of the existing infrastructure. In turn it provides several new facilities and features that others can make use of. The existing features that are included in Janos are obtained from two software projects: the OSKit component libraries and Kaffe, a free java virtual machine.
OSKit - The OSKit is built from existing components such as device drivers, POSIX API's, file systems, and network protocols, obtained from currently existing systems and made interoperable.
Kaffe – 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 Janos is built. [14] [13]

Conventional operating systems use separate virtual address spaces or system call traps to separate applications and the kernel. Java operating systems do not use virtual address spaces or system calls traps but instead use type-safety to enforce protection and separation. C is used only for implementing trusted portions of the system. All other untrusted code to be run on Janos must be written in Java.

Janos design is divided into three separate components: Moab, JanosVM, and ANTSR. Each component's architecture and contributions are explained below. [14]

A. Moab

DARPA has proposed the active network program's NodeOS specification. Moab is complaint with DARPA's NodeOS specification. Moab is implemented on top of the OSKit. The OSKit provides Moab with a lot on components like the device

drivers, several complete file systems, a threading implementation, the FreeBSD networking stack, and many support modules such as boot loaders and remote debugging support. The benefit of building on top of OSKit is that POSIX-reliant system 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 OS; invocations of NodeOS functions are direct function calls and do not trap into the OS.
In this paper, the design and motivation of the NodeOS API is not dealt in detail. This paper only deals with areas in which Moab has deviated from the NodeOS API specification. A short review is presented below. [14][13][3]

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.

Moab is not currently fully compliant with the NodeOS API specification as it deviates in some of the minor interfaces. Moab has deliberately deviated from the specification for reasons specified below: [14] [3]

1. **Single, Trusted Execution Environment**
   NodeOS specification states that a NodeOS should be capable of supporting multiple execution environments. But in case of Moab, only one execution environment is supported. This single execution environment means that Moab does not need to mediate between execution environments. [14] [3]

2. **Resource Specification**
   Current NodeOS specification specifies imprecise, hardware-independent resource specifications. On the contrary, Moab resources are precisely specified in terms of a local node's hardware. As an 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 execution environment to present this hardware-specific information to the application programmers and node administrators in a meaningful fashion. [14] [3]

3. **Memory Accounting**
   The one of the memory goals of Janos is to allow the applications to manage their own memory usage. To meet this goal, Moab has been designed in such a way that almost all memory management can be performed by execution environment. In other words, all the memory used by Moab on behalf of the execution environment is provided explicitly by the execution environment. As an example, creating a thread requires the execution environment to provide memory for the thread control block and memory for the thread's stack. [14] [3]

4. **Packet Buffers**

Packets are unique in Moab. The packet design is oriented around a single problem: the NodeOS must receive an incoming packet into the memory before deciding which domain owns the packet. Moab is designed to implement Zero buffer copies in the case of forwarding the incoming packets, even if untrusted user code makes the decision on forwarding and modifying the packets. To avoid copies, we need to have the incoming packets to be received into a buffer so that those packets can be handed over to the respective domain. An execution environment associates buffers with each of its input channels. To maintain resource limits, a buffer is only handed 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 the incoming packets, and the domain neither gains nor loses buffers. [14] [3]

5. **CPU Accounting**

Moab deviates from the NodeOS thread pool specification to attain the goal of allowing the users of the NodeOS to manage CPU usage within a domain. Thread pools in Moab are bound to specific input channels. This allows the CPU 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. [14] [3]

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 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. The current memory management interfaces (at all levels) deal with memory as a guaranteed, strictly accounted resource. [14] [3]

7. **Click Channels**

The click modular router is software from MIT that enables administrators to dynamically create Linux-based packed 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. [14] [3]

B. JanosVM

The JanosVM is the most critical part of a Janos node. This is where the C based interfaces are mapped 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. [14] [3] [9]

Foremost, the JANOSVM is based directly on the KaffeOS implementation. KaffeOS is a JVM 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. The 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 preprocess basis. 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. [9]

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. As it is not possible to describe all of the JANOSVM in this short paper, I think it is better to 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. [14][3][9]

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 interdomain 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 [13] [9].

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.

Buffers are sharable between domains, though the memory cost is always borne entirely by a single domain (the *owner* of the buffer). 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 [13].

3. Resource Specifications

As discussed in Section 4-A2, 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 [13].

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 [13].

C. ANTSR

The ANTSR Java runtime is based on ANTS 1.1 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. 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 the 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, a new domain was created early and made downloading and installing of its code the first task of the new domain. This design implicitly restricts code loading resources to the new domain's limits [14][6].

D. 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 [13].

5. Related Work [14]

Active research is going on in active networks area. The research work related to Janos that is going on can be grouped into three categories.

1. Node Operating Systems

There are four major NodeOS implementations. They are Moab, Princeton's Scout-based system, AMP system, and the Bowman NodeOS. While Moab focuses on resource control and domain management, the Princeton system integrates NodeOS abstractions in Scout's paths, and AMP focuses on security. 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 [14].

2. Execution Environments

As noted earlier, ANTSR execution environment is directly derived from ANTS 1.1, and ANTS 2.0 is derived from ANTSR. There are several other EEs that focus on active networking issues such as security, administration, or protocol development. Ideally, the ideas and interfaces from those projects could be folded into ANTSR. SwitchWare 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 is based on the Caml programming language. The SwitchWare developers have said they intend to port their PLAN execution environment to Moab [14].

3. Java Operating Systems

The JANOSVM is a Java OS—a Java language runtime that supports OS abstractions—and builds upon several previous projects in this area. The JANOSVM directly uses the multiple Java heap implementations from the K0 and KaffeOS systems, and as they do, builds on the base Kaffe Java virtual machine. 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. Inferno 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, provides the domain separation and resource control, nor the orientation to network protocol processing that has been the focus of our work [14].

6. Open Issues

The present 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 discuss some of these issues below and touch on certain areas that should be explored in the future. Janos makes available a relatively restricted model for the composition of AAs and for data sharing between applications. While it is expected that the model is suitable for a wide variety of practical applications, it remains to be seen how the model suits the needs of third party developers in practice. The 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 [14]. Because we have provided few concessions to interdomain 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. [14] [3] [2]

Janos also presents interesting issues in the area of Java code optimizations [12]. There are many optimizations that need to be done to JanosVM. There are issues involving compilation techniques for handling special data structures such as packet buffers. An active application should not be given direct access to any unprotected system resource, including packet buffers since it is untrusted. 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. This can be implemented in future. [14][2][13]

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.[14]

7. Conclusion

The architecture of the Janos active node operating system has been explained in this paper. 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. Thus, Janos performs well in playing the role of an operating system for the active nodes [14].

References

1. (2001, Jan) NodeOS Interface Specification. Active NodeOS Working Group.[Online].Available: http://www.cs.princeton.edu/nsg/papers/nodeos.ps
2. (1999, July) Active Networks Architecture Documents. Active Network Working Group. [Online]. Available: http://www.darpa.mil/ito/research/anets/Arcdocs.html
3. Larry Peterson, Yitzchak Gottlieb, Mike Hibler, Patrick Tullman, Jay Lepreau, Stephen Schwab, Hrishikesh Dandekar, Andrew Purtell, John Hartman, "An OS Interface for Active Routers", IEEE journal, March 2001
4. P.Tullmananc C. Hawblitzel. (1999,Sept.) Janos Java Documentation
5. David L.Tennenhouse, Jonathan M.Smith, W.David Sincoskie, David J.Wetherall, Gary J.Minden,"A survey of Active Network Research", IEEE Communications Magazine, Jan 1997.
6. David J.Wetherall, John V.Guttag, David L. Tennenhouse, "ANTS: A Toolkit for Building and Dynamically Deploying Network Protocols", To appear in IEEE OPENARCH'98 San Francisco, CA, April 1998.
7. Commentaries on "Active Networking and End-to-End Arguments", IEEE Network Magazine, May/June 1998.
8. Tennenhouse,D.L. and D.J.Wetherall, "Towards Active Networks", 1996, MIT Laboratory for Computer Science.
9. Godmar Back, Patrick Tullman, Leigh Stoller, Wilson C Hsieh, Jay Lepreau, "Techniques for the Design of Java Operating Systems", To appear in the proceedings of the USENIX Annual Technical Conference, June 2000.
10. M.Hicks et al, "PLANet: An Active Network Testbed," http://www.cis.upenn.edu/~switchware/papers/planet.ps,1998
11. D.Scott Alexander, William A. Arbaugh, Michael W.Hicks,Pankaj Kakkar, Angelos D.Keromytis, Jonathan T.Moore, Carl A.Gunter, Scott M.Nettles, Jonathan M.Smith, "The SwitchWare Active Network Architecture"
12. Godmar Back, Patrick Tullman, Leigh Stoller, Wilson C.Hsieh, Jay Lepreau, "Techniques for the Design of Java Operating Systems"
13. Active Network Working Group, "Architectural Framework for Active Networks Version 1.0"
14. Patrick Tullman, Mike Hibler, Jay Lepreau, "Janos: A Java-oriented OS for Active Network Nodes"
15. Yechiam Yemini, Sushil da Silva, Department of Computer Science, Columbia University, "Towards Programmable Networks"