Commercially Viable Active Networking

Stuart Eichert[1,2], Osman N. Ertugay[2], Dan Nessett, Suresh Vobbilisetty
3Com Corporation, Technology Development Center
5400 Bayfront Plaza
Santa Clara, CA. 95052
seichert@coopcomp.com, ertugay@dsl.cis.upenn.edu, dnessett@home.com,
vobbilis@ix.netcom.com

Abstract

Abstract - Active Networking is a new technology receiving significant attention from the research community. To this point, however, it has not been examined from the perspective of commercial viability. This paper presents an analysis of active networking issues with a view to its possible uses in a commercial environment. It then describes a prototype system built to address these issues.

1 Introduction

While the Internet has grown significantly over the past several decades, there are signs that applications would benefit from a richer set of internet services than those currently available. For example, the expected growth in voice over IP (VoIP) traffic on the Internet will change the service requirements presented to it. While real-time delivery requirements for IP traffic have been rare up to now, VoIP packets must be delivered within fairly strict time constraints. Since the volume of VoIP traffic is likely to grow as a percentage of internet traffic, the need for hard or at least soft real-time guarantees will become increasingly important. As other real-time traffic, such as video, becomes prominent, the effect on internet services will be even more dramatic.

Responding to these changing requirements, network engineers and researchers are investigating ways to provide improved network service. One interesting technology in this regard is **active networking** [16]. Active networking uses a programming language for mobile code to dynamically change network device behavior. Special messages (**active packets**) carry small programs (**active code**) or references to such programs, which are executed by active networking devices (**active nodes**). These programs may execute only once or they may be loaded into node memory and execute many times. In the latter case they may be loaded as shared libraries, being made available to many packet flows transiting the network device, or they may be loaded as private code called by active packets belonging to one flow. An application implemented using active code is known as an **active application**.

Not all devices in an active network must be active. Active networks can (and probably will for the foreseeable future) consist of active nodes interconnected by passive networking hardware. From an abstract viewpoint, the passive parts of an active network act as links between active nodes.

Active networking supports a range of services characterized by the type of system that initiates an active application. At one end of this range, end systems send active packets, each of which carries a program fragment that processes the packet at the active nodes visited along its path. We call this approach **capsule-based** active networking. At the other end of this range, a privileged end system dynamically augments network routers by loading new programs into them while they run. We call this paradigm **programmable devices**.

We are interested in active networking for many reasons. However, the focus of this paper is an assessment of its commercial viability. Consequently, we concentrate on factors such as the motivation for active networking, technical issues that must be solved to make active networking practical, and how active networks can provide performance, security, interoperability, and cost/functionality tradeoffs that are as good or better than passive networks.

In the next section we examine some advantages and disadvantages of active networking. In section 3 we explore technical issues associated with this technology that are important to its commercial viability. Section 4 describes an investigation we undertook to test some of the ideas we describe in section 3. Following that, we present

[1] Work done while a student intern at 3Com. Current address: Cooperative Computers, Inc., 650 Castro St. Ste 120-216, Mountain View, CA 94041
[2] Work done while a student intern at 3Com, and doing graduate work at the University of Pennsylvania.

some related work and then conclusions we have drawn from this study.

2 Some Advantages and Disadvantages of Active Networking

At first sight active networking may strike some as a bad idea. The prospect of dynamically loading programs into network devices thereby changing their behavior and consequently changing the overall behavior of the network in unforeseeable ways might seem to be a prescription for disaster. While appropriate security and testing controls can greatly reduce the probability of chaotic network behavior, there must be advantages that offset the additional costs and risks introduced by active networking.

It is beyond the scope of this paper to give a complete and comprehensive analysis of active networking advantages and disadvantages, a subject that would require a full length paper in its own right. Instead, we briefly indicate how active networking differs from passive networking and discuss the tradeoffs between them.

Active networking has at least the following advantages:

- New code can be loaded into a device without rebooting it. This means new functionality can be added to a device without disrupting on-going activity, such as routing protocol update exchanges, critical financial transaction traffic or real-time process control traffic. Note that keeping the system safe and consistent while adding and removing device functionality is an issue that must be addressed by the active networking architecture.
- Active nodes can provide a wider variety of functionality than passive devices with the same amount of program memory. A passive network device must contain all of the code implementing its full feature set[3]. An active node, on the other hand, need only have resident the code necessary to support those features in use.
- Once a passive device is deployed, its feature set may only be modified by loading a new version of the software. Generally, such software upgrades add multiple features to a device, which means they are infrequently rolled out. An active device allows incremental feature upgrade, thus fostering more frequent enhancements.
- Development of network services and applications for different platforms is easier due to the platform-independent languages that active networking is based on. The use of programming languages supporting mobile code eases the deployment of new protocols on heterogeneous platforms.

[3] Of course, it is possible to have different software loads for a given hardware platform, which mitigates this problem somewhat. However, all of the code necessary to support the features of a particular software load must reside on the platform whether or not all of those features are used.

These advantages are balanced by the following disadvantages:

- Extra functionality is required on active devices to run active code. This functionality requires space in the device's program memory, extra processing power to execute active code, and field support to correct any implementation deficiencies, all of which introduce additional cost.
- It is hard to test all combinations of features implemented by active code. In practice this means that the number of such features supported by vendors may be limited.
- In order for active code to provide significant value-added services, there must be a set of interfaces between the active code support base and native device functionality. This requires extra development and maintenance by the device vendor, which again increases cost.
- Network forwarding performance of active packets is inferior to that of passive packets forwarded by hardware. However, how disadvantageous this is depends on the types of applications that will utilize active networking. Network control applications should not be impacted greatly by low forwarding performance.
- Exporting a powerful programming interface to the network infrastructure raises significant safety and security issues. Addressing these requires additional functionality, which increases cost.

Roughly, these issues are summarized as tradeoffs between flexibility and cost. Active networking delivers more flexibility to end users, but it increases networking vendor engineering and maintenance costs. Therefore, either active networking products will be priced higher than passive networking products in order to cover the increased costs, or vendors will find ways to use active networking to decrease other costs or increase their revenue in other ways, such as increasing volume.

3 Technical Issues

Active networking introduces a number of significant technical issues over and above those associated with passive networks. These include:

- how to structure active applications,
- the general implementation strategy for the active node, i.e., basing it on a general purpose operating system or on an existing network device,

- how to handle the relatively new performance, security, interoperability, and cost issues specific to active networking,
- how to structure the execution environment provided for active code, and
- how to structure interfaces providing access to underlying device functionality.

3.1 Active Application Architecture

The commercial viability of active networking will rest on the usefulness of the applications it supports. Very little has been written to date on requirements and design issues of active applications intended for a commercial (as opposed to a research) environment.

In most cases commercial applications are:

- Developed by vendors, who must maintain and enhance them,
- Deployed by administrators, who must manage them, and
- Used by users, who depend on them.

Note that in a commercial environment the intersection between the set of individuals associated with vendors and administrators or administrators and users is a small percentage of the total number of individuals in their union. This is generally not true in a research environment. Consequently, much work to date on active networking has not addressed commercial requirements.

Since each community involved with a commercial application generally has different objectives and requirements, individuals working with a commercial application will do so in very different ways than those in a research community. Specifically:

- Vendors must fix application bugs and provide customers with feature enhancements. Vendor customer service organizations must identify the version of an application used by a customer when a problem occurs in order to provide field support as well as communicate bug reports back to the appropriate development team.
- Administrators must be able to stand-down versions of an application in order to replace them with a newer version (to fix bugs or deploy new features). They also must install new applications or new versions of them. In large deployments, it is administratively difficult to install an application on each individual end-system, which leads to the practice of installing them on servers from which they are downloaded or remotely executed.
- When applications are upgraded to fix bugs or make enhancements, it should not be necessary for users to choose between the old and new versions when executing them (although, it may be desirable to allow them to do so). When a new version of an application is installed, it should not interfere with current executions based on older versions.

In a commercial setting, these requirements apply to active as well as traditional applications and thereby encourage certain active application architectural choices. In particular:

- Vendors should tag or otherwise associate an active application with versioning information. When deployed by administrators, active applications should support version identification for bug reporting/fixing.
- When active applications are deployed, administrators should be able to flush older versions from the network and establish environmental controls that prevent older versions from being reintroduced. Since they are intended for research environments, the active networking systems previously developed do not provide mechanisms that achieve this objective.
- When administrators install a new version of an active application, it must not interfere with current executions of older versions. Synchronization of active code execution by application components is a major issue. For example, when two versions of the same active code run on an active node, they may interact with common state in a way that interferes with both of their executions.

3.2 Active node implementation strategies

Active nodes may be based on general-purpose operating systems, such as one of the varieties of Unix, or they may be based on an existing networking device, which generally uses a proprietary or commercially supported real-time operating system (**RTOS**). We use the term **software-based active node** to describe the former and the term **hybrid active node** to describe the latter. In both cases only part of the device is dedicated to active networking (however, see [7] for a description of an object oriented node OS that fits very neatly into the active networking paradigm). We believe that while this paper concentrates on routers with special purpose hardware and a real-time OS, most of our conclusions are, at some level, also applicable to general-purpose OS-based end systems which are active networking enabled.

Software-based active nodes are useful to investigate active networking when packet forwarding performance is not an objective. These systems are easy to program and debug, are well known to a wide range of programmers and exist in varieties that cost little or in some cases are free. However, since forwarding decisions are made in software, these nodes cannot deliver price/performance advantages comparable to those of contemporary high-performance networking devices.

Hybrid active nodes utilize most of the functionality of an existing networking device and layer active networking services on top of it. In many cases the active networking services run on a management processor, while forwarding and lower-level services run on one or more communication interface processors. In very high performance network devices, packet forwarding is handled by hardware [11].

When constructing a hybrid active node from such a device, the execution environment must provide interfaces to the forwarding, filtering and other underlying data and control services. Active code then controls the device by using these interfaces to manage the hardware and software of the underlying device.

Given the almost universal practice in commercial high-performance routers and switches of moving forwarding into ASICs or other hardware, it is unlikely that active networking used in the data plane (i.e., on the forwarding fast path) will be commercially viable for the foreseeable future. Consequently, we believe that the commercial viability of active networking for high-performance routers/switches rests on how useful it is in the control plane.

3.3 Performance

Active networking utilizes a platform-independent programming language to control a wide range of network devices. While not strictly a requirement, in practice this language will probably be interpreted and thus not provide very good performance. On the other hand, to be competitive, network devices must deliver very high performance.

There are a number of ways to address the performance issue. One is to target active networking devices to those parts of the network where features are more important than performance (e.g., low-end office routers, workgroup switches). However, these devices are often cost sensitive. They are thus unlikely candidates to introduce leading edge features, such as active networking, the value of which is uncertain.

Another approach is to work on the mobile code performance problem. There is already work in this area, including Just-In-Time compilers, more efficient byte code interpreters and incremental compilation strategies. On the other hand, problems such as how to build garbage collectors for real-time systems are not yet fully understood, so this approach is not entirely satisfactory.

An approach that works well for many legacy network devices is to use the hybrid architecture described in section 3.2. This limits any performance degradation to the control plane, which has much less stringent performance requirements than frame/packet forwarding or filtering.

3.4 Security

A major issue in active networking is how to perform resource access control on network devices. Resources may be fundamental device assets, such as link bandwidth, node memory and node processor cycles or may be more abstract, such as programming resources, (e.g., object classes, object methods and object state), data resources (e.g., packets, persistent storage), and communications resources (e.g., packet flows).

Active networking introduces new requirements for network devices that lead to more sophisticated access control mechanisms. For example, active code executing in one thread may wish to communicate with code executing in another. These two threads may represent computations carried out on behalf of two different principals. The access control facilities in the execution environment must control such communications according to the appropriate policies.

Furthermore, providing access control to shared state may be difficult. Interestingly, this problem was addressed in the late 60s and early 70s in research systems, such as Hydra [9], but little progress has been made subsequently, since commercially successful operating systems (e.g., Unix) have more primitive access control mechanisms. Consequently, researchers are reexamining problems that have not received significant attention for several decades.

3.5 Interoperability

It is possible that in a large network different administrative domains will support different active networking infrastructure (e,g., ANTS [17] or ALIEN [1, 3]). It is also possible that some domains will support more than one kind of infrastructure.

Heterogeneity of active networking infrastructure creates more problems for programmable devices than capsules. In the latter case, active nodes that do not support the appropriate infrastructure treat a capsule as if it was a passive message. Correspondingly, from the capsule's point of view, these active nodes behave like passive nodes[4].

For programmable devices, however, when two domains of an active network support different active networking infrastructures, coordinating their management requires an understanding of how services in one might be provided, or at least emulated, in the other. Otherwise, it is difficult to define active code that achieves a specific objective in both domains.

[4] This assumes an active node will forward packets that carry no active information they recognize using traditional packet handling services.

3.6 Cost/functionality tradeoffs

Providing active networking functionality is not free. Not only must network devices support the appropriate active networking transport protocols, execution environments and device service interfaces, they also must contain the additional resources necessary to run active code. This introduces extra cost that must be balanced by equally valuable features. Furthermore, end systems (in the capsule approach) and network management platforms (in the programmable device approach) must be developed or modified so that they create and send active packets. Finally, an active network must supply storage for active code, which must be highly available and accessible to all systems that require access to it.

3.7 Execution environments

When an active packet arrives at an active node, it is processed by an **execution environment**. An active node may support more than one execution environment, although whether this is commercially viable is unclear.

An execution environment provides the following general services:

- The execution of portable platform-independent code.
- A multiplexing substrate for executing threads. This allows code from multiple active packets to execute concurrently so they may block waiting for events and resources without adversely affecting the executions associated with other active packets.
- Storage of active code. Active code may be cached in the execution environment for efficiency.
- A security subsystem for controlling access to active node resources. These resources may be native to the node, such as clocks, buffer memory, or network interfaces or they may be logical resources created by shared libraries or representing exported objects of executing active code.

The commercial viability of a particular execution environment architecture depends on how well it supports the active application requirements specified in section 3.1. An execution environment should

- Support versioning,
- Support the retirement of old versions of active code, and
- Allow currently executing active applications using old versions to complete before the old version implementations are removed.

3.8 Service interfaces

For both software-based and hybrid active nodes, an execution environment must provide interfaces to device services. For software-based nodes, these will control the underlying operating system services. For hybrid nodes, they will control similar functionality, but the interface implementations will generally manipulate specialized hardware and software that controls those services.

While a device may support multiple execution environments, it is unlikely it will provide more than one interface set to its services. Furthermore, porting execution environments to different devices requires matching their device service calls to the interfaces provided. Standardizing these interfaces makes porting easier.

One effort at standardizing the interface between an execution environment and the node OS is the NodeOS Interface Specification of the DARPA Active Networking community [13]. This service interface defines five resource abstractions: 1) thread pools, 2) memory pools, 3) channels, 4) files, and 5) flows. The specification defines a set of service primitives for each resource abstraction.

The NodeOS Interface Specification is a useful tool for designing software-based active nodes, for which all aspects of the active node implementation are available for modification. For hybrid active nodes, there is less flexibility in the execution environment and node OS design space, making it difficult and sometimes impossible to follow the specification architecture. The network device on which we layered our experimental execution environment (see section 4.1) has pre-existing models for threads, memory allocation, channels, files and flows. These models were significantly different than those presented in the NodeOS Interface Specification.

For example, the device software runs over a commercial real-time OS (RTOS) on which is layered a Java Virtual Machine. Java threads (the primitive execution resources in our execution environment) are directly mapped to RTOS threads. Resources are allocated to RTOS threads using a pre-existing set of service primitives that are not easily accessible to the execution environment. This is true for each of the other resource abstractions as well. Consequently, we did not follow the NodeOS Interface Specification. Instead, we developed service primitives that seemed better suited to a hybrid active node with a pre-existing resource architecture.

We designed a set of Java interfaces for controlling services on our hybrid active node. We then implemented these interfaces using a combination of Java, C++ and C, the latter two languages being used for adding functionality to the existing device software.

Our interface set operates on 3 resource abstractions: 1) **packet flows** (not to be confused with *flows* of the NodeOS Interface Specification – packet flows closely resemble *channels* defined in that document), 2) **MIB resources**, and 3) **communication resources**. Our node OS model assumes that packet flows are primitive resources established by the node OS, not by the execution environment. Our packet flow service primitives allow an active application to specify packet classification and handling information, which is used by the node OS to configure the device hardware to apply special processing to the packets moving through these flows.

There are three interface sets, one each for: 1) **filtering and forwarding management**, 2) **MIB management**, and 3) **message communication**. Filtering and forwarding is combined into one interface set because managing forwarding information is logically equivalent to managing filters driven by only destination addresses. MIB management gives active code access to a wide range of device functionality. Finally, message communication services allow active code to send raw IP packets[5].

3.8.1 Filtering and forwarding management

Filtering and forwarding management (FFM) gives active code the ability to manipulate **filter groups** consisting of a set of filters. A filter group is associated with an **action** that will be applied to packets that match the filter group. A filter group can have either **hit** or **miss** semantics. In hit semantics, a packet satisfies a filter group if it matches any filter in the group. In miss semantics, a packet satisfies a filter group if it matches no filter in the group. The associated action is performed on packets matching the group.

One of four types of action may be associated with a filter group:

- **Drop Action**: satisfying packets are dropped,
- **Forward Action**: satisfying packets are forwarded to a specified interface,
- **Prioritize Action**: satisfying packets are forwarded to an identified interface with a specified priority,
- **Registered Action**: satisfying packets are processed by a specified routine. The routine can be a C function compiled into the system, or a Java class which implements a well-known method. Section 4.1.3 describes the Registered action type more completely.

FFM supports both layer-3/4 and layer-2 filter groups. A layer-3/4 filter operates on the address values carried in the IP/UDP/TCP header. Layer-3/4 filters are specified using a source address, source address mask, destination address, destination address mask, source port lower bound, source port higher bound, destination port lower bound, and destination port higher bound. Source and destination ports are only meaningful for UDP and TCP; these values are ignored for groups specifying other protocols. A layer-2 filter is specified using source and destination addresses only. A filter group is comprised of either layer-3/4 or layer-2 filters, mixtures of these two types are not allowed.

Both layer-3/4 and layer-2 filters are divided into two categories: interface-based filters and flow-based filters. Interface-based filters are applied to all packets passing through a given interface in a specified direction (in or out). Thus interface-based filters are suitable for situations where all the packets coming in or going out of an interface are of interest. Flow-based filters, on the other hand, are applied to packets moving through the device, regardless of the interface used. Flow-based filters apply only to packets arriving from another node, i.e. packets generated by the active node are not processed.

The FFM service interface supports:

- adding filters to a group,
- deleting filters from a group,
- searching for filters in a group, and
- creating actions and associating them with a group.

A layer-3/4 filter group is identified by a protocol number and cast type (unicast vs. multicast.) In the case of interface-based filters, filter group identification also includes an interface number and the direction of packet flow (i.e., into or out of the interface).

To associate different processing with packets satisfied by the same filter group, a registered action must act as a dispatcher, sub-multiplexing packet processing to other procedures. For example, the FFM interface supports only one filter group for unicast flow-based UDP packets. Consequently, if different sets of unicast UDP packets must be processed using different procedures, the action routine associated with the unicast UDP filter group must classify the packets passed to it, separate them into the appropriate set, and call the appropriate routine.

At first sight this might seem to be a poor design. However, we carefully chose this approach to solve the rule list ambiguity problem. Specifically, if we had chosen to associate actions with filters, instead of filter groups, an ordering of filters in a group would be required to disambiguate the action to take. That is, if a packet satisfies two filters, each associated with different actions, one must take precedence over the other. In the past, this has led to confusion when trying to understand rule list semantics [5].

Of course, this approach moves the problem of disambiguation up to the demultiplexing code. However, this code can implement arbitrary algorithms, while filter lists are processed according to a fixed algorithm, making it

[5] There is no reason why active code cannot send both active and passive packets. To keep the interfaces simple, we assume executing active code sends an active packet by first formatting it appropriately and then presenting to the send interface method as a simple byte buffer. Active code can receive messages using the filtering/forwarding management interfaces.

difficult and in some cases impossible to achieve the desired semantics.

It is not possible to sub-multiplex drop, forward or prioritize actions, since these are generally supported by the hardware. The only way to solve this is to use a registered action that performs these operations in software.

While our approach is less flexible than associating actions with filters, there is no possibility of ambiguity when filters are dynamically added and removed from filter groups. Whether this advantage outweighs the loss of flexibility is untested (however, see section 6).

3.8.2 MIB variable management

The MIB management service interface allows active code to manipulate the SNMP MIB variables implemented by the underlying device software. It supports the following three SNMP operations: 1) get, 2) get-next, and 3) set. All operations take a MIB object identifier (OID) as an input parameter, "set" also taking a value as input.

The values associated with SNMP MIB variables can be one of 3 simple types: 1) Integer32, 2) Octet String, and 3) Object Identifier[6]. Consequently, the result communicated through this interface consists of a type identifier and a value. For "get" operations, the type is returned first, allowing the caller to choose the appropriately typed method to obtain the variable value.

The "get-next" operation returns a result that contains the OID and value of the variable "after" the one identified in the call. Continued use of "get-next" supplying the OID returned by the previous call allows active code to traverse a MIB table or array. The "set" operation is implemented as three separate methods, each corresponding to one of the supported MIB variable types.

When designing the MIB variable management service interface, we encountered the following problem. MIB variables are defined using ASN.1 syntax and the mapping between ASN.1 syntax and its in-memory representation as a class hierarchy is non-standard. In order to ensure the usability of the MIB variable management interface on a wide number of devices, the interface makes no assumption how MIB variable values are represented by internal program data structures.

3.8.3 Message communications

The packet communication interface provides a way for active code to send packets to other nodes. Currently we support layer-3 packets, although the code is architected to add support for layer-2 frames. In addition to the standard Java Network API, our layer-3 communication service interface allows active code to send raw IP packets. The interface also supports creating IP and UDP headers, computing checksums, and creating ANEP packets.

[6] While SNMP also supports MIB objects of type Integer16, its use is rare and objects of this type do not appear in standard network management MIBs.

4 An Implementation and Results[7]

We implemented an active networking infrastructure and built two active networking applications in order to investigate the commercial viability of this technology. Our infrastructure used commercially available platforms, such as a commercial-grade layer-3 switch and an LDAP directory system. Of the two active applications we built, one uses capsules, while the other is based on programmable device architecture.

4.1 An experimental execution environment

We implemented an execution environment (**EE**) based on the Java Programming Language on a commercial-grade layer-3 switch. This implementation consists of Java software that manages the execution of active code and C/C++ software that inserts the appropriate functionality into the device for FFM, MIB and communications services.

As noted in section 3.1, developing, deploying, and using active applications (**AA**s) are distinct activities that are performed by different groups of people. We kept this in mind while designing the EE. Development involves AA implementation, packaging, and distribution. In most cases, development will also include creating applications for end systems that will use AAs. Deployment involves administering the use of AAs in an active network.

4.1.1 Active code transport

The first design issue we faced was whether to carry active code or a reference to it in capsules. In the first approach, an end system that sends a capsule must store (at least temporarily) the code locally. This can be achieved by directly installing the code on every end system that will use it, having the end systems retrieve the code from a server whenever they need it, or combining the two approaches using a caching strategy. Having multiple copies of active code on a large number of end systems introduces code maintenance problems. Furthermore, having each capsule in the same flow carry code is not bandwidth-efficient, since code may be cached at active nodes. One variation of this approach has an end system place code in capsules until all intermediate active nodes have cached it. Subsequently, the end system would send out capsules with references to the code. However, it is hard

[7] This work was done at a time when 3Com had a router business, which it exited in 2000. Consequently, we will not investigate most of the open questions identified in the paper's text. They are presented for the benefit of those who might wish to explore them further.

for end systems to determine when all active nodes along a path have a cached copy.

In the second approach capsules carry a reference to active code and active nodes retrieve it using the reference, caching it for future use. For each capsule, the node executes the code directly if it is in its cache, otherwise it contacts a server (or possibly another active node) to retrieve it. With this approach, the burden of retrieving code is on the active nodes. A subsidiary characteristic of this approach is it makes it less likely that capsules are fragmented across multiple IP packets. Furthermore, it allows nodes to use TCP-based protocols to reliably fetch active code.

The tradeoffs between these two approaches are driven by the ratio of end systems to active nodes. If there are more end systems using an AA than active nodes running it, there is less network traffic if active nodes retrieve code. In our design, we assume that the average number of end systems using an AA will be larger than the average number of active nodes running it. Consequently, capsules carry code references rather than code.

4.1.2 Active code storage, distribution and naming

Our code transport design requires a scheme for retrieving code based on a reference. This involves two issues: code storage and distribution, and code-naming.

We address the code storage and distribution issues as follows. Capsules carry references specifying where and how to retrieve the active code. As stressed in section 3.1, we believe AA maintenance is an important issue for commercially viable active networking. AA maintenance is the problem of ensuring that when a new AA version, which fixes a bug or enhances some feature is released, the active network eventually removes all old versions of the AA residing in active nodes.

One factor that is closely related to this issue is how and where the infrastructure stores AAs. It is important that bug fixes propagate to all active nodes quickly, and when an AA expires that it becomes inaccessible or unusable by active nodes. Storing AAs on a large number of end systems makes it difficult to achieve this objective, since end system management is rarely integrated with network management in a commercial environment.

If AAs are stored on servers supporting replication, updating AAs and ensuring that active nodes use the current version is facilitated. Our experimental infrastructure uses LDAP servers for AA storage. LDAP supports replication and since standards-based network policy management will use LDAP services, using them makes it easier to deploy and use active networking in policy-based networks.

Code-naming is also an important issue. We use a naming scheme based on Java fully qualified class names. Our active code packages contain a number of class files, one of which is distinguished. It is used by the EE initially (through a well-known entry point) to process a capsule. We call this class an **iclass** (initial class). Each iclass extends an abstract base class provided by the EE, and implements a method called *process* whose signature the base class enforces. We name the code package by the fully qualified name of the iclass and its version, which is maintained by a versioning system. Since class names in Java are hierarchical, lookups are efficient in hierarchical directory infrastructures like LDAP. An iclass name is intended to indicate what the code does. Global uniqueness is achieved only if the Java class naming convention is strictly obeyed and the versions are correctly maintained by developers.

Classes in a package other than the iclass are the closure of classes required by the iclass for processing capsules (with the exception of system classes provided by the EE). Whenever an iclass is changed, its closure classes may also change, which is why we use the iclass name and version identifier to identify the active code package. The package carries version information for each class that it contains.

A capsule carries an iclass name and version as the active code reference. When the EE sees a new reference, the code package is retrieved using the information from the capsule. The code package is carried in a signed JAR file. When the package is retrieved, the signature is verified and the classes (and their versions) are extracted.

The byte codes for classes are stored by the EE in a byte code store for later use by a class loader. A byte code image in a code package is stored only if it is not already in the byte store (i.e., the class name and version do not match those of any existing entry). Otherwise, a usage counter for the byte code entry is incremented to indicate the number of iclasses that depend on it.

Since a class with the same version number may appear in multiple code packages, carrying the byte codes for all closure classes in a package increases the required storage space (only at the server, not in the active node) and the bandwidth necessary to retrieve it. On the other hand, the number of bytes in an iclass package is likely to be quite small (e.g., the JAR file for the Power Traceroute application we developed is 5.7K bytes), so this is unlikely to be a practical problem. The redundant storage problem could be prevented by storing each class separately in the server in its own signed package. The active nodes could then fetch only the classes that are required by an iclass but not found in the node. However, this approach would introduce considerable overhead in the number of server requests made by the active node (one request to find the required classes, one request for each not-already-node-resident class) In addition, each closure class would require signature verification increasing the processing necessary to retrieve an iclass. Note that we trust the signed code in terms of resource usage, security, and safety. Hence signature verification is very important for ensuring that the code does not harm the active node or other active applications.

Another important feature of our EE is it allows dynamic unloading of unused AAs (i.e., all non-system classes within an iclass closure). A separate class loader is used for each iclass. The class loader is created with a table of references to the byte codes required by that iclass. Thus, each iclass executes in its own name space, and class unloading becomes possible at the iclass level. Java unloads classes in groups, where the classes loaded by the same class loader constitute a group. Whenever the class loader for a group becomes unreachable its classes are unloaded as a unit.

Our EE does not completely solve the AA maintenance problem. Firstly, active nodes are not notified when a new version of a class is installed in the LDAP servers. Secondly, capsules referring to the old version may continually refresh the cached iclass copy. Finally, active nodes will not use a newer version of the iclass when they fail to find an older version in the LDAP server(s). We have developed approaches to each of these problems, but they are untested.

4.1.3 The service interfaces

Implementing the MIB management interface was straightforward. Each service request (i.e., get, get-next, and set) is implemented as a class constructor. The object returned by the constructor has methods for getting the operation status and its results (in the case of get and get-next).

Our implementation of FFM supports both flow-based and interface-based layer-3/4 filters (layer-2 filters are only partially implemented). Implementation of registered actions operates as follows. A **registering iclass** is one that creates a registered action. A character string identifies a registered action. During iclass registration, if the registered action name matches that of a C function in the EE's list of native packet processing functions, the action is internally flagged as native. Otherwise, the action is flagged as a Java class and the string is assumed to contain the fully qualified name of an iclass and its version. This iclass is called a **registered iclass**.

When a packet satisfies a filter group associated with a registered action, the action flag is checked. If the registered action is a native C function, the execution environment calls it with the packet as a parameter. If the registered action is a registered iclass, the execution environment calls the *process* method of an instance providing the packet as a parameter. If an iclass execution context does not already exist, the code package for the registered iclass is retrieved from the server that contained the registering iclass package. Then, an execution context is created for the registered iclass. Note that registered iclasses are loaded by a different class loader than the registering iclass. So, if the registering iclass is unloaded, the registered iclass is still available.

In our implementation, we retrieve the registered iclass code package when the first packet satisfying the filter group arrives. However, we could have chosen to do the retrieval when the iclass is initially registered. There are tradeoffs between these two strategies. The former does not use memory in the node until the iclass is used, but the latency for processing the first packet is high due to code retrieval. In the latter case, these tradeoffs are reversed.

It is important to keep in mind the difference between capsule and FFM processing. Each capsule contains the name of the iclass to be executed when it arrives. Packets matching a filter group initiate execution of an iclass only by virtue of this relationship, which is ephemeral (i.e., filter groups can be registered and unregistered). While the same iclass may be named in a capsule and associated with a filter group, its execution in the two cases occurs for very different reasons.

Other action types (Drop, Forward, and Prioritize) are performed by the device without EE intervention[9]. The primary goal of these types is to provide active code the ability to control and manage packet flows without introducing significant packet flow performance degradation.

The FFM implementation mediates creation of filter group instances and stores them in a global table. Operations on each of these instances are synchronized. This allows the FFM implementation to ensure two different threads are not modifying the same filter group at the same time.

Our implementation of the message communications interface provides active code with the ability to send raw IP packets by placing an IP header in front of a given payload. The code can specify TTL, protocol number, source address, and destination address field values. Although code may use standard Java classes for sending UDP packets, our implementation provides interfaces for creating and encapsulating a UDP packet in a raw IP packet, since the standard classes do not allow manipulation of an IP or UDP header.

4.1.4 Capsules

We use ANEP version 1 [2] in our implementation and UDP packets with destination port number 3324 to transport ANEP packets.

Our EE defines four new ANEP options. All capsules must carry an iclass option, containing the iclass name and version. Capsules may also carry:

[9] Of these we have only implemented Drop.

- A server-info option, containing information on how and where to retrieve the code package;
- A signer-info option containing an identifier indicating the identity of the principal who signed the code package;
- An auto-forward option, indicating whether to forward the capsule based on its end-to-end addressing information before it's processed by the EE.

UDP packets with a destination port of 3324 are captured by the underlying system hardware without impacting the performance of other packet flows. If the packet contains an ANEP (version 1) header, it is processed further. If not, the UDP packet is returned to the underlying system for forwarding. ANEP version 1 packets are checked to see if their type ID matches that assigned to our EE and if they contain an iclass option. If not, the UDP packet is returned to the underlying system for forwarding. Packets with our type ID and an iclass option are passed to the EE. The EE first extracts the iclass option and checks to see whether it already has an execution context associated with it in the node. The execution context for an iclass consists of a class loader, references to class byte codes from the code package stored by the EE, a timer, a thread group, server information used for retrieving the code package, and a class object for the iclass. A new iclass instance is created for every capsule to ensure processing of one capsule does not block processing of subsequent capsules.

If the execution context exists, the iclass expiration timer is refreshed, a new instance of the iclass is created and a specific method of the instance is invoked with the ANEP options and payload as parameters. The EE performs this invocation in a new thread of execution residing within the thread group for the iclass. If an execution context does not exist, the EE creates one and the capsule is processed as above. The EE removes an execution context when its timer expires, provided there are no active threads in the thread group. Otherwise, the timer is refreshed and the same check is performed at the next expiration. Note that we trust the signed code not to produce run-away threads.

4.1.5 Security

Our approach to active networking security relies on functionality provided by Java. A long-term strategy is to utilize the fine-grained access control features of Java 2.0 [6]. These utilize access control policy information maintained by the Java Runtime to establish permissions associated with a protection domain. In the current implementation of the Java 2.0 security architecture, protection domains are collections of classes and objects, which are assigned permissions based on the location from which the Java Runtime fetched a class implementation as well as the signature used to sign it. The Java 2.0 architecture allows future implementations to associate a principal identity and delegated permissions with a protection domain.

In our implementation, each active node is configured with a single static public key. The execution environment uses this public key to verify a JAR file's signature and the verified identity to perform authorization. We do not currently support the use of certificate chains. However, we could easily change our implementation to configure active nodes with multiple trusted identities with their associated public keys and to process certificate chains for signature verification.

4.2 Applications for Active Networking

Given this execution environment it is straightforward to create applications for the management and configuration of network devices, the installation of new networking protocols, and the creation of new networking services. We developed two sample applications to investigate the power of active networking. First is a dynamic policy management application that builds on previous work with multi-layer firewalls. Second is a power traceroute application that retrieves MIB variable data at each active node it visits on the path to its final destination.

4.2.1 Dynamic network policy management

We implemented a dynamic policy management application, which tested the ability of active code to control network device behavior in response to local events. This application enhanced our multilayer firewall (MLF) prototype, which implements firewall functionality within a network, rather than at its borders [12]. We modified the MLF prototype to support timed filters using active code downloaded from the MLF management system to active nodes.

Timed filters are firewall rules enforced only during a certain period of time, like Monday through Friday of every week, or nine-to-five each day. Our implementation of timed filters has three components: a Graphical User Interface, filtering code to set and clear the filters, and active objects.

The Graphical User Interface is written in Java using Swing, the GUI library for JDK 1.1.6 and higher. It presents a network administrator with a table of filter rules, which define MLF filtering policy. To support timed filters, we added another column to the MLF filter rule table, which allows a network administrator to specify a period of enforcement for each rule. Each period definition includes a start date/time, an end date/time and a repetition term. While the initial two items are obvious, the repetition term allows an administrator to specify how often to repeat the enforcement of a rule. An administrator has the option of selecting a period of enforcement value of ***always***, which means the rule is not time based.

As described in section 4.1, our prototype active node supports IP layer packet filtering through a combination of hardware and software services. Our **active MLF** prototype uses an early version of the interfaces described in section 3.8.1 to set and clear filters that drop packets.

We use **active objects** to implement the timed filters defined by the active MLF management system. An active object has data members containing the filter and enforcement period information corresponding to one rule in the active MLF firewall table. When an active node receives an active object, the execution environment creates a thread by passing the run method of the active object to the thread constructor. An active object is serializable, meaning its state can be captured, preserved, and then used to reconstitute it on a different system.

When the active MLF management system processes its firewall rule table, the active objects it creates (one per rule) are serialized and placed in a Java archive (JAR). The JAR file is deposited in an LDAP directory system by the active MLF management system, which then signals the appropriate enforcement device that it has a new archive to retrieve by sending it the archive's distinguished name. Prior to creating new threads for each active object, the execution environment terminates the threads associated with the existing MLF active objects.

Our experience with active MLF taught us the following:

- In the original MLF prototype, filter data is written to flash memory so that security policy remains enforced even when the device reboots. However, in the active MLF prototype, filter data is encapsulated in objects. Consequently, for a practical deployment of active MLF, active code may need to be stored and retrieved from local storage as well as central servers in order to deliver the same security guarantees. This introduces new complexity into the AA maintenance problem, which we did not foresee.
- Another objective of our active MLF prototype was to learn how quickly we could modify the filtering controls in the underlying device. Since active MLF changes firewall rules in an active device by eliminating those currently effective and then establishing the complete filter rule set once again, measuring how long it takes to transition from one policy to another overestimates filter rule modification time. However, an independent experiment demonstrated that we can add or delete a filter to/from a filter group in less than 10 milliseconds. While these times are dependent on our implementation as well as that of the underlying device, it provides evidence that practical control plane interfaces are possible.

4.2.2 Power traceroute

Power Traceroute is a capsule-based active application for retrieving information from intermediate nodes along a path between a source and destination. The capsule specifies a list of SNMP MIB variables, and the active nodes reply with a list of MIB variable and value pairs. The Power Traceroute iclass is dynamically loaded into intermediate active nodes when required, and dynamically removed when the iclass timer expires.

Power Traceroute sends one capsule to investigate node information along a path. The alternative would be to send out a set of SNMP requests to each intermediate node. This would require the end system to discover them and then send each a separate SNMP request. Power Traceroute thus reduces network traffic considerably. Furthermore, since the Power Traceroute iclass is signed, administrative domains have a higher degree of confidence that the application will not access MIB data in an inappropriate way. On the other hand, allowing end systems to use SNMP to access intermediate node MIBs does not constrain them in how they use that access privilege.

Power Traceroute is implemented as a single class. The payload of the ANEP packet carries several flags, a sequence number, the number of variables requested, and a list of those variables. The iclass option in the ANEP header refers to the signed Power Traceroute package. The server type and location information in the server-info option indicates where the iclass is stored.

The Power Traceroute active code forwards the capsule to the next node after incrementing the sequence number. Forwarding is based on a parameter passed to the code by the EE, which specifies the destination IP address of the capsule. The values for the listed MIB variables are retrieved using the MIB service interface and the results put in a UDP packet. This is sent back to the original source of the capsule using information carried by options passed to the iclass by the EE, specifying the source IP address and UDP port.

The client application at the end system listens on the UDP port on which it sent the capsule. As replies from active nodes arrive, they are parsed and displayed.

We conducted several experiments using our Power Traceroute prototype to gain an understanding of active application performance. We first measured the effect of active networking on the performance of other traffic passing through the node. We performed experiments on the testbed shown in Figure 1.

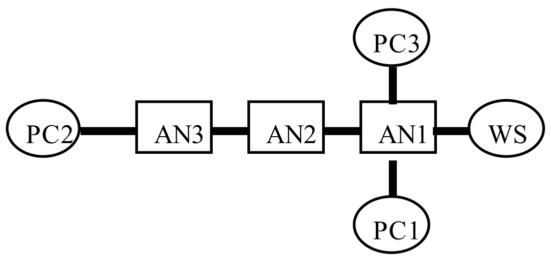

Figure 1 – Active Network Testbed

All links between systems are 100 Mbps Ethernet. In our experiments, a workstation (WS) sends packets to PC2 as fast as it can, utilizing all the available bandwidth. The observed throughput between PC2 and WS is 95 Mbits/sec.

In our first experiment, we initiated PC2 to WS traffic, then attempted to interfere with it by sending Power Traceroute capsules from PC1 to PC3. As soon as AN1 sends a response back to PC1, PC1 sends another capsule towards PC3. As expected, this had no effect on the traffic between WS and PC2. Capsules are captured by the hardware and forwarded to the management processor, which does not interfere with the hardware processing of other packets, except when resources are depleted (see below).

We performed another experiment with the same configuration, except PC1 sent capsules to PC3 as fast as possible without waiting for responses from AN1. Initially, the traffic between WS and PC2 was not affected, but eventually throughput decreased to zero.

We also measured the performance of some of our service interfaces. Setting the value of an OctetString variable takes approximately 170 ms. Getting the value of an OctetString variable takes 50 ms and getting the value of an Integer variable takes 35 ms. Adding or removing a filter to/from a filtergroup takes 5 ms on average. Also, a Power Traceroute capsule is processed in approximately 200 ms (without any SNMP variable queries).

Our experience with the Power Traceroute prototype taught us the following:

- The prototype is fairly simple and could be improved in a number of ways. For example, it might be enhanced to support table and array retrieval, OID prefix traversal and determining whether a particular feature is enabled along a path.
- Power Traceroute illustrated one deficiency of our EE design. While we authenticate and authorize active code implementations, we do not do this for capsules. Providing this functionality is straightforward (e.g., capsules can be signed by a principal and checked for authorization at each active node).
- Experimenting with the Power Traceroute prototype also demonstrated that we need better resource management controls. As described above, we sent Power Traceroute capsules along a path as fast as an end system could generate them (without waiting for replies). As expected, this overwhelmed the management processor running the EE, leading to a large number of active packets waiting in buffers at lower levels of the system. This drove device throughput to zero, since no buffer space was left for forwarding. Consequently, a practical implementation needs ways to throttle capsule traffic so this cannot happen.

4.3 General observations

The experience we gained during the implementation of our execution environment as well as during the development of and experimentation with our two active applications leads us to the following conclusions:

- Hybrid active nodes are feasible. Not only is it possible to build them, it is also possible to use them to support interesting active applications. While a number of problems remain to be addressed by our hybrid active node prototype (e.g., capsule authentication, robust control plane resource management), we are convinced these are tractable.
- The hybrid active node approach isn't free, however. Each service interface exported for use by active applications requires implementation of a significant amount of device specific code. Therefore, the number of service interfaces should be kept as small as possible. We believe the MIB service interface is powerful and can be used to support a wide range of device interaction requirements.
- While we did not comprehensively stress test our execution environment implementation, the experimentation we carried out with our active applications demonstrates that control plane based active networking can deliver high forwarding performance and yet provide the flexibility that justifies active networking.
- We believe the general categories of service interfaces we developed (filtering and forwarding management, MIB management and communications) are sufficient to build commercially viable active applications. We have successfully used our MIB management service interface and believe it is well-suited for a wide range of active applications. However, we think the interface could be improved by supporting SNMP trap functionality. We are not satisfied with the current architecture of our FFM service interface and have several ideas how it might be improved.
- We found it hard to debug active applications. Debugging our hybrid active node required us to follow execution paths through Java application code, Java execution environment implementation code, and frequently down into the device software. While appropriate quality assurance testing might eliminate the last case in a product, use of a Java remote debugging tool may introduce product support problems, as from its perspective the execution environment and active application are a single Java execution. For example, it is unlikely that vendors will openly publish source code for their EEs, which may interfere with active application debugging. Consequently, we believe there needs to be significant investigations examining how to support active application debugging in a commercial environment.

- We learned we didn't need to explicitly support shared libraries. We use the class loader features of Java to separate active application address spaces, thereby creating containment domains that isolate their executions. In effect, our byte code store implements something similar to shared libraries, except it doesn't support state sharing. We have an initial design of a global shared state service, but it is untested.
- We discovered that naming active code packages by iclass name and version is insufficient to properly manage active applications. Specifically, if two principals sign a package named by the same iclass name and version number, there is ambiguity as to which package takes precedence. Therefore, we now believe active code packages should be named by iclass name, version and signer identity.

5 Related work

Jaeger, et. al. [8] describe another implementation of active networking on a commercial router platform. However, their published work does not investigate the commercial viability of this technology in detail. Active networking has enjoyed a significant growth in interest over the last several years and a number of efforts address problems similar to ours.

ANTS [17] is an active networking system developed at MIT that uses Java as its mobile programming language. When an ANTS capsule requiring non-resident active code arrives at an active node, a request for the code is sent back to the active node that forwarded the capsule. Since the antecedent node followed the same procedure, it normally has a copy. While this is an elegantly simple distribution technique, ANTS does not support versioning. Consequently, there is no way to identify which active application code is more recent. Capsules referring to old versions will continue to pull that code into the network after newer versions are deployed.

ANTS names active code using an MD5 message digest of the code content. This provides highly probable global uniqueness, but is not indicative of what the code does. In addition, looking up a name is not very efficient because of the flat name space.

ALIEN [1, 3] uses Objective CAML as its mobile programming language and its execution environment provides a system facility, called the Core Switchlet that arbitrates active code access to execution environment resources. The Core Switchlet maintains and enforces policy that controls which functions are available to active code. Our security and resource-multiplexing model is much less sophisticated than ALIEN's. This was a deliberate choice. We believe an initial deployment of active networking can use a simple trust model for which the principal signing active code is trusted to behave properly (both from a security as well as a resource management perspective).

For example, initially vendor equipment might only run active code signed by it. The vendor then takes the responsibility to ensure active applications do not interfere unnecessarily with one another and do not introduce security hazards. While in the long run a more sophisticated security model is desirable, signed active code and signed capsules would allow vendors and users to gain experience with active networking while the research community investigates this issue. We think that the security model provided by Java 2.0 (with enhancements supported by its architecture) provides a good foundation for a future active networking security model.

Smart Packets [15] provides a service interface allowing active code to access MIB data. It also authenticates capsules using an integrity check of their contents signed by a private key. We have already stated our view that a MIB service interface provides a powerful node management capability. The capsule signing approach used by Smart Packets seems promising.

Raz and Shavitt report work they have conducted on applying active networking to network management [14]. They also describe a service interface to underlying device functionality through its MIBs. Their system model divides an active node into two entities, an IP router and an adjunct active engine.

We have taken a similar approach to theirs, although our implementation integrates both entities on a single device. Unlike our prototype, theirs does not support capsules. Their model allows active code to intercept non-active packets and manipulate them, as does ours. However, it appears that all non-active packets are manipulated by software, whereas our prototype allows active code to manipulate non-active packets by controlling the underlying forwarding hardware.

RCANE [10] allows active code to specify a filter that will be applied to active packets received on one of its interfaces. When a packet or frame satisfying the filter arrives, it is passed to the active code for processing. This service interface is similar to that provided by our FFM facility. However, we allow active code to assign an independent iclass to the processing function, which remains associated with the filter group even after the code that registers it is unloaded.

CANES [4] allows capsules to select one of a set of fixed functions available at active nodes to process their content. It does not allow capsules to carry code or references to code, thereby providing strict controls on active application behavior. While there are specific security and efficiency advantages to this approach, we don't see why the use of signed code cannot be used to deliver equivalent assurances.

6 Conclusions

The goal of the work described in this paper was to assess the commercial viability of active networking. While we cannot claim to have settled this issue, we believe we have made some progress towards an answer.

Commercially viable active nodes will almost certainly place processing by active code in the control plane. In the most recent high-performance switches and routers, fast path packet and frame forwarding is handled by very high-performance hardware that must be simple and therefore passive. Thus, we believe hybrid active nodes will play a prominent role in the commercial arena. This is not to say there is no role for software-based active nodes, but we believe they are better suited for research or experimentation with active networking or for low-end devices that emphasize features over performance.

Commercial viability also depends on active networking infrastructure that supports active application maintenance. Identifying this requirement and investigating it is a major contribution of our work.

We explored several service interfaces that may be used to implement active applications. We strongly believe a MIB service interface is a powerful and effective way for active code to interact with and control network devices.

We did not comprehensively explore the performance and security issues that will affect commercial viability. We believe vendors can deliver an initial deployment of active networking based on the simple trust model used in our prototyping. We believe the security architecture of Java 2.0 is powerful and could form the basis of a sophisticated active networking security facility. A more sophisticated resource management model than the simple one we use in our prototype may eventually be necessary, although we are skeptical that one based on charging for node resources will be practical.

Heterogeneity is an important issue that we did not address. We expect solutions to it will require some form of standardization, either of the active networking infrastructure or of service interfaces. Ultimately, we do not think running multiple EEs on a network device will be supported in commercial deployments of active networking.

Active networking technology is potentially very powerful and we believe researchers and engineers have only scratched the surface in regard to its uses. Nevertheless, other equally powerful technologies have failed in the marketplace of ideas (e.g., timer-based transport protocols, dynamically microprogrammable systems). Thus, we believe the jury is still out on the question of whether the expense of active networking technology is sufficiently justified for commercial deployment.

7 Acknowledgements

We thank Ted Czotter for commenting on our original service interface design, which led to several improvements and Peter Wang for comments that led to several improvements of this paper. We also thank Wenjun Luo, who did some very early work on our experimental execution environment. As always, the referees provided useful comments, which also improved the paper.

8 Bibliography

[1] D. Scott Alexander, "ALIEN: A generalized computing model of active networks," PhD Thesis, University of Pennsylvania, Philadelphia, December 1998.

[2] D.S. Alexander, R. Braden, C.A. Gunter, A.W. Jackson, A.D. Keromytis, G.J. Minden, D. Wetherall, "Active Network Encapsulation Protocol (ANEP)," http://www.cis.upenn.edu/~switchware/ANEP, August 1997.

[3] D.S. Alexander, Jonathan M. Smith, "The architecture of ALIEN," Proceedings of the First International Working Conference, IWAN '99, Berlin, Germany, June/July, 1999, (also appears as Lecture Notes in Computer Science 1653, Springer-Verlag, Berlin).

[4] Samrat Bhattacharjee, Kenneth J. Calvert, Ellen W. Zegura, "Implementation of an active networking architecture," presented at the Gigabit Switch Technology Workshop, Washington University, St. Louis, July, 1996.

[5] D. Brent Chapman, "Network (in)security through IP packet filtering," Proceedings of the Third USENIX UNIX Security Symposium, Baltimore, MD., Sept. 1992. 1996.

[6] Li Gong and Roland Schemers, "Implementing protection domains in the Java development kit 1.2," Proceedings ISOC Symposium on Network and Distributed System Security, San Diego, CA., 1998.

[7] J. Hartman, U. Manber, L. Peterson, T. Proebsting, "Liquid Software: A new paradigm for networked systems," Technical Report 96-11, Department of Computer Science, University of Arizona, June 1996.

[8] R. Jaeger, S. Bhattacharjee, J. K. Hollingsworth, R. Duncan, T. Lavarian and F. Travostino, "Integrating active networking and commercial grade routing platforms," Proceedings of the USENIX Special Workshop on Intelligence at the Network Edge, March 20, 2000, San Francisco, CA.

[9] Anita Jones, "Protection in programmed systems," Ph.D. Thesis, Carnegie-Mellon University, Pittsburg, Pennsylvania, June, 1973.

[10] P. Menage, "RCANE: A resource controlled framework for active network services," Proceedings of the First International Working Conference, IWAN '99, Ber-

lin, Germany, June/July, 1999, (also appears as Lecture Notes in Computer Science 1653, Springer-Verlag, Berlin).

[11] Dan Nessett, “Commercial use of active networking,” presented at the OpenSIG Workshop, University of Toronto, October 5-6, 1998.

[12] Dan Nessett and Polar Humenn, “The Multilayer Firewall,” Proceedings ISOC Symposium on Network and Distributed System Security, San Diego, CA., 1998.

[13] L. Peterson, ed., "NodeOS interface specification," AN NodeOS Working Group Draft, July 1999.

[14] D. Raz, Y. Shavitt, "An active network approach to efficient network management," Proceedings of the First International Working Conference, IWAN ’99, Berlin, Germany, June/July, 1999, (also appears as Lecture Notes in Computer Science 1653, Springer-Verlag, Berlin).

[15] B. Schwartz, W. Zhou, A.W. Jackson, W.T. Strayer, D. Rockwell, C. Partridge, "Smart packets for active networks," 2nd IEEE OPENARCH, 1999.

[16] D.L. Tennenhouse, J.M. Smith, W.D. Sincoskie, D.J. Wetherall, and G.J. Minden, “A survey of active networking research,” IEEE Communications Magazine, Jan., 1997.

[17] D.J. Wetherall, J.V. Guttag, D.L. Tennenhouse, "ANTS: A toolkit for building and dynamically deploying network protocols," IEEE OPENARCH '98, San Francisco, CA, April 1998.