An Active Network Services Architecture for Routers with Silicon-Based Forwarding Engines

R. Jaeger[1], R. Duncan[2], F. Travostino[2], T. Lavian[2], J. Hollingsworth[1]
[1]*University of Maryland, College Park, MD 20742*
[2]*Nortel Networks, 4401 Great America Parkway, Santa Clara, CA 9505*
{rfj,hollings}@cs.umd.edu {rduncan, travos, tlavian}@nortelnetworks.com

***Abstract*--Current network devices enable connectivity between end systems with support for routing with a defined set of protocol software bundled with the hardware. These devices do not support user customization or the introduction of new software applications. Active networking devices allow for the dynamic downloading of customized programs into network devices allowing for the introduction of new protocols and application services. The Oplet Runtime Environment (ORE) is a programmable network architecture built on a Gigabit Ethernet L3 Routing Switch to support downloadable services. Complementing the ORE, we introduce the JFWD API, a uniform, platform-independent portal through which application programmers control the forwarding engines of heterogeneous network nodes (e.g., switches and routers). The ORE architecture enables industry and customer specific services to run on silicon-based network devices.**

***Index terms*-- Distributed Applications, Networking Protocols, ANTS, NodeOS, ORE, Programmable Networks, JFWD**

I. INTRODUCTION

Active Networks enable the introduction of user computing to network devices to provide new and improved services and distributed application functionality. Traditional networks deliver packets between endpoints using protocol software that is bundled with the hardware. Network devices do not support user customization or introduction of new software applications nor do they facilitate introducing new protocols quickly.

This paper are the implementation of a Java network services architecture to concurrently run multiple non-interfering execution environments (EEs) on a hardware-based gigabit router as well as the JFWD API to control forwarding plane operations. Also significant is the explanation of the issues related to constructing a stable framework that will support Active Networking on routing switches with silicon-based forward engines and real-time operating systems. We demonstrate on a commercial-grade routing hardware that we can support multiple Java-based services and implement the ANTS execution environment as an ORE service which is capable of interoperating with other ANTS implementations.
This document introduces the JFWD network API and describes an application to control packet forwarding in a silicon based forwarding Routing Switch.

The Nortel ORE architecture for programmable network services enables customized user software to be run as services on Active Network devices to perform monitoring, routing, diagnostic, or other user specified functions. Customized ORE services running locally on a network device can monitor and change specific Management Information Base (MIB) variables through the MIB API [4]. Direct access to MIB variables on network nodes greatly improves scalability and reduces network traffic compared with using SNMP agents to perform remote "get" and "set" operations. The capability to perform MIB queries locally on the network device opens up a new set of applications [4].

The JFWD API is a networking interface that allows ORE services to modify how packets are handled by the forwarding engine. Packets can be diverted to the control plane or copied to the control plane while also forwarding the packets on an output link. Services can also modify packet processing in the forwarding engine through the JFWD. A packet signature can be an IPv6 flow id, a combination of IPv4 header fields (IP source and/or destination, protocol, port, etc), or even MAC layer information.

The remainder of this paper is organized as follows. Section 2 introduces the Active Networking model. Section 3 discusses the architecture of the Oplet Runtime Environment (ORE) and its support for services including execution environments such as ANTS. Section 4 introduces the JFWD networking API. Section 5 discusses implementation of the ORE and future enhancements followed by the conclusion in Section 5.

II. ACTIVE NETWORKING ARCHITECTURE

The goal of the Active Networks architecture is to provide a platform on which multiple concurrent non-interfering execution environments (EEs) can coexist on an active node to support active applications (AA). Each EE acts as a

"shell" to AAs providing an API that supports network services.

Figure 1 shows the logical Active Network architecture. The bottom layer is the System Resource Manager, or node operating system, that provides mechanisms to prevent one EE from interfering with another. These mechanisms include limiting resources to each EE as well as dispatching packets to the correct EE. System resources include CPU cycles, memory, persistent storage, and access to I/O bandwidth. Active packets are dispatched to EEs based on a TypeID field included in the Active Networking Encapsulation Protocol (ANEP). The System Resource Manager dispatches ANEP packets to the corresponding EE based on the TypeID field. Also contained in the ANEP packet is the IP address of the original packet source and the IP address of its final destination.

An execution environment accepts programs and packets that it deems valid. Active applications (AA) can be persistent in which case they may keep state within the EE, or may be ephemeral and die after execution. EEs embody both the local processing and network view of Active Networking.

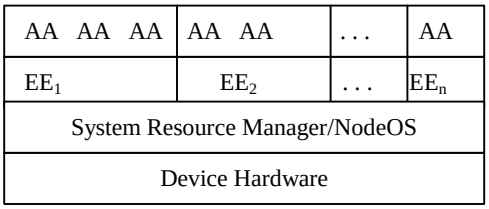


Figure 1: Active Network Reference Model
Active Applications (AA) running on EEs supported by the system resource manager (NodeOS operating system)

The Active Networking model has five main design objectives for the EE to Operating System interface:

1) create a minimum set of simple standardized protocols to permit the development and implementation of end-to-end services by both research and commercial organizations
2) maximize flexibility in the services which can be supported concurrently
3) allow interconnectivity of networks under separate administrative domains
4) support scaling of the network in terms of size and speed
5) encompass current protocols, specifically, IP. [25]

The ORE achieves these Active Network goals through the service interface it provides.

III. The Oplet Runtime Environment (ORE)

The Oplet Runtime Environment (ORE) described in this paper supports injecting customized software services, including EEs, into network devices. The implemented architecture is composed of an embedded Java Virtual Machine (JVM) and the ORE. The ORE component provides the substrate on the network device to support the downloading, installation, and execution of services. Active Network execution environment and active applications can be written as ORE services. The goal of the ORE is to provide for sharing of resources among competing services running concurrently in the JVM. The ORE partitions the system into manageable pieces and is designed to be robust, scaleable, and secure. The ORE makes the system maintainable, components reuseable, and facilitates the development of services through a simple ORE-to-service interface.

ORE services compete for memory, CPU, persistent storage, and link bandwidth resources. These are the same resources managed by NodeOS, the node operating system defined by the Active Networks working group. The motivation for defining the NodeOS was to provide for the concurrent deployment of multiple non-interfering EEs in a portable manner. To achieve this goal, both NodeOS and the ORE must manage system resources. In addition to the fundamental system resources, the ORE services and the ORE itself are resources that must be managed. Resource management involves both the allocation of resources as well as the control of those resources. The ORE architecture attempts to mediate the shared resources between mutually suspicious services and to provide for inter-service communication including the sharing of objects and calls into other services.

The ORE architecture consists of the Oplet Runtime Environment, oplets, and services. Oplets are self-contained downloadable units that encapsulate one or more services, service attributes, authentication information, and resource requirements. One oplet attribute is a list of services upon which this oplet depends. Before installing an oplet, these required services must already be running in the system. Oplets also publish those services they provide to the ORE. The ORE tracks these dependencies between oplets and services, so that if a service becomes unavailable the oplets that are dependent on it are notified.

The ORE manages the oplet lifecycle and maintains a registry of the active services. The ORE provides mechanisms to install, activate, update, deactivate, kill, and uninstall oplets on the network node. Persistent oplets are automatically installed and activated when the ORE is started. Prior to activating an oplet, all services upon which it depends must be activated. When the oplet is activated it can then register the services it provides with the ORE.

The ORE mandates that all services run in the system conform to the oplet service structure requiring a minimum of boilerplate wrapper code. The ORE provides the machinery to download oplets from URLs (http, file, or ftp). For example, an Active Networking EE run on the ORE system must be run as an ORE service. Thus, only EEs

written in Java that implement the oplet specification can be run by the ORE.

Since the ORE and its services are constrained to running in the JVM, system stability of the core network device operations is not affected. Native code applications are prohibited because their presence would potentially compromise the stability of the system. A failure of the JVM or ORE services will not impact other tasks running on the router.

IV. The ORE Implementation

Embedded and real-time environments are especially concerned with reliability, determinism, and constrained resources. Java was chosen to build the ORE for security and stability reasons. Since all ORE services run within the JVM, the non-Java tasks in the network device cannot be harmed. We reuse the security mechanisms of the class-loader, bytecode verifier, and security manager.

The ORE provides a platform on which to run concurrent non-interfering services on Java-enabled network devices. Although the JVM provides mechanisms to run user services safely on active nodes, the environment has several deficiencies. Java does not inherently provide mechanisms for sharing resources among mutually suspicious services. The allocation and control of memory, CPU, bandwidth, and persistent storage resources is required to build a stable system to support non-interfering services and avoid denial of service attacks. An extensive exploration of denial of service issues is described in [1]. The ORE abstracts some of the system resource management facilities in order to provide resource protection.

As far as possible, the ORE uses the mechanisms provided by the standard JVM to protect itself from the oplets, and to protect them from each other. These mechanisms include features of the Java language itself (type safety, bytecode verification), the built-in sandbox security manager support, signed code, and the strong cryptography infrastructure.

These facilities are buttressed by giving the ORE control over the allocation of as many of the system resources as possible and requiring the oplets to go through the ORE in order to acquire them.

Nevertheless, for complete protection against misbehavior by malicious or buggy oplets, some extra support is needed from the JVM that is non-standard. The most significant modification to the conventional JVM is the ability to perform accounting for both CPU and memory consumption. One promising possibility for memory accounting is the ability to partition the object heap to enforce limits on the memory usage by an oplet. More details on this are given in the following sections.

A. Computation Management

CPU attacks are a threat that must be dealt with given multiple threads sharing a single JVM. Running each service in a separate off-the-shelf JVM can alleviate the interference between services. However, running in this manner is not desirable due to dependencies on the underlying RTOS task scheduling. Another difficulty with running services in separate JVMs involves communication and sharing of data between services. Running multiple services, each instantiating a separate JVM, would consume significant resources and is not scaleable [1]. A better approach is to run all services under a single JVM process running in the RTOS.

The ORE controls thread creation. When an oplet needs a new thread it must request it from the ORE. The ORE will determine whether to grant the request based upon current thread usage, and the credentials of the requesting oplet. However, once allocated, the ORE has no way to account for or limit the consumption of computing resources by the thread. Denial of service caused by a rogue service consuming CPU resources must be addressed. The ORE needs JVM support that does not exist today.

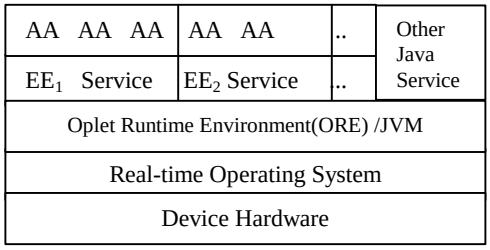


Figure 2: ORE Implementation
Java Services including EEs & AAs running on the ORE substrate

Sharing threads between oplets presents two main problems: (a) when an oplet relinquishes control of the current thread by calling a method in another oplet the possibility exists that the caller will block forever if the callee never returns; (b) if the caller oplet has retained a reference to the thread object, it can kill it or otherwise interfere with it even while it is within a critical section of the callee oplet's code.

To reduce the ability of a caller oplet to break the callee by exploding its critical section, the ORE security manager denies oplets the ability to manipulate a thread's running status.

The ORE attempts to protect itself from problem (a) by never interacting directly with the oplets that it loads. Instead it creates a trusted proxy which it uses to delegate its commands to the untrusted oplet.

The proxy uses a separate thread to call a method on the untrusted oplet and sets a timeout for returning from the call. If the timeout expires, the ORE can infer that something has gone wrong with the oplet and can take evasive action. The thread cannot simply be killed, however, because this would break any critical sections that the thread is in. Instead, the best that the ORE can do is to inform the offending oplet that it is being killed, and then to remove all access the offending oplet has to the resources it controls such as file descriptors. If the oplet ignores this, the thread is set to run with the lowest priority, to minimize the impact it has on the rest of the system.

The next generation ORE will extend this mechanism to automatically generate proxies for communication between untrusted oplets themselves. The proxies are delegated the responsibility to run the methods in the called service. These mechanisms allow for safe communication among tasks. However, this safety is expensive in terms of performance.

The cost for the thread hand-off mechanism is believed to be tolerable based on the timing numbers observed in [26]. First, communication with a trusted service need not involve the hand-off. Also, inter-service calls requiring the hand-off is not anticipated to comprise a significant portion of the service processing time. We are willing to accept this performance hit since our primary goal is to provide a safe environment in which oplet services can run.

B. *Memory Management*

Denial of service attacks can be mounted based on memory consumption given that all threads in the standard JVM [1] share the JVM heap and garbage collector. The ORE solution to the memory resource consumption problem requires that each oplet be given a separate memory budget. This might be implemented by modifying the JVM to provide a separate heap for each oplet. Separate heaps introduce a new set of problems, however. For example, if they are to be garbage collected independently the system must ensure that there are no cross-heap object references.

The proxies introduced to mitigate thread interference can be extended to also prevent references from one heap being passed into another one. In [26], the authors discuss the JKernel protection mechanisms by which revocable objects are passed by reference and others require passing a copy of the referenced object. Revoking an object makes its memory available for garbage collection.

Revocation of objects is also a powerful way to give an oplet the ability to control access to its own resources. For example, if an oplet determines that another client oplet is no longer permitted to use a resource reference, the reference can be revoked. Without a mechanism of this sort, oplet services must be designed to support a model that requires them to continue to grant access to their resources indefinitely, even in the face of malicious clients.

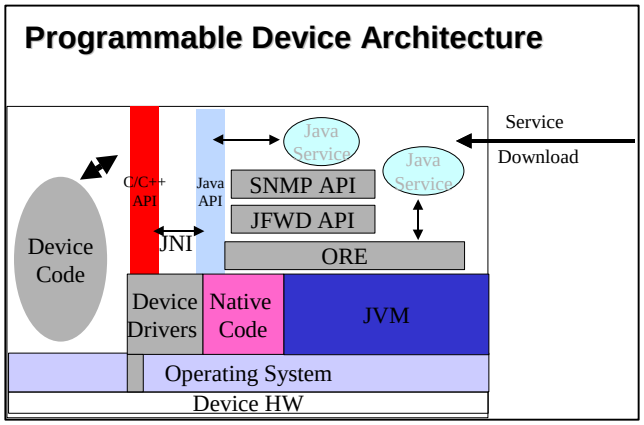


Figure 3: Programmable DeviceArchitecture including the ORE, MIB API, JFWD API

C. *Future*

The ORE can control the allocation of file descriptors and socket resources. However, the size of the persistent storage used by a service is not controllable by the ORE. While bandwidth consumption could be controlled by the ORE if all calls were forced to use an ORE I/O class, we have not felt the need to implement this feature.

In the ORE architecture, denial-of-service attacks are still possible including triggering the garbage collector, memory fragmentation attacks, exploiting conservative based garbage collectors, and stalling finalization of objects[1]. We believe that the memory consumption issue will be resolved when JVMs support multiple heaps and that the revocation and copy semantics of the JKernel approach will resolve those issues. Future enhancements will allow more direct control over computational resource consumption. For now, we accept only services whose source is one that is trusted and whose authenticity and integrity can be verified[1]. Today, a trusted application with bugs that attempts any of these "attacks" is not stopped.

V. JAVA FORWARDING API

We have implemented a network API on a commercial-grade ASIC-based routing switch that allows ORE services to request alternative packet handling from the hardware forwarding plane. The JFWD API specifies interfaces for Java applications to control a generic, platform-neutral forwarding plane. The JFWD API consists of a collection of Java classes specifications. The implementations of these classes is not considered part of the JFWD API technology, because it is code that, unlike the API, is highly platform-specific and thus lend itself to other distribution strategies.

A. *Java Forwarding Architecture*

JFWD is the first Java API package to explicitly address the needs of contemporary forwarding planes of network nodes. Increasingly, these forwarding planes are implemented in silicon, must support multiple protocols, and must realize Quality-of-Service (QoS) low-level functionality such as filtering and early drop. To drive an ever-growing number of options, more diversified control data will have to be fed into control planes.

The JFWD implementation across multiple hardware platforms brings a) the abstraction of a platform-neutral forwarding engine, b) a framework within which new protocols and control data can be described, and c) a new breed of shrink-wrapped Java programs for controlling network nodes. Such shrink-wrapped, often mobile Java programs can just-in-time nest into Java-enabled network nodes and realize innovative paradigms, like in the following scenarios:

- *Ad-hoc routing, policy enforcement*: Control plane programs self-deploy in a number of participating nodes and realize custom routing behaviors within a Virtual Router that is serving a Virtual Private Network;
- *Control-plane-less nodes*: It is fairly easy to scale towards high-end configurations by clustering together several control-plane-less network nodes and a high powered legacy CPU which acts as a server for all control plane needs (see Figure 3);
- *Intrusion detection*: By correlating JFWD-structured control data from different nodes, control plane programs can transcend platform dependencies and pinpoint intrusion patterns more easily. In this case, JFWD discloses a more privileged perspective on system behaviors than management applications would get through MIBs;
- *Peering protocol dialects*: Some protocol specifications leave room for proprietary dialects that corporations use as lingua franca among their boxes (e.g. PANL is Nortel's favorite superset of Frame Relay); these protocol dialects typically require code being active at both ends. JFWD allows one to quickly deploy the environment and code for peering a protocol dialect on any box supporting JFWD;
- *Spare control-plane*: The Java co-processor in a network node may execute a stripped-down control plane in Java while the full-featured control plane is being serviced. The stripped-down control plane may be loaded on demand and discarded after use;
- *Watch-me-fail*: A network node can set watchdogs that monitor forwarding engine's behaviors and then move these watchdogs onto neighboring nodes. Exiled watchdogs will wait and catch failures as they are perceived from the outside of the network node.

These examples support the claim that JFWD does indeed enable new paradigms that improve customizability, fault resilience, and adaptability.

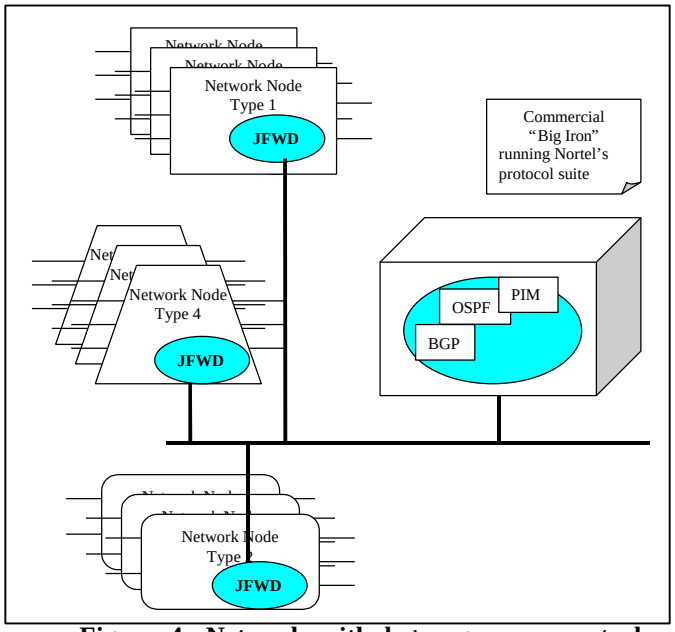


Figure 4: Network with heterogeneous control-plane-less network devices. The forwarding plane is local, control plane is centralized and remote

B. *JFWD Implementation*

The JFWD API provides ORE services opportunities to modify the way in which packets are forwarded. Specifically, the ORE services can use the JFWD API to request receipt of packets with a particular signature. The forwarding engine then delivers matching packets to the requesting service. A service can request that the matching packet be delivered only to the control plane or that it be forwarded normally and duplicated as well. Thus, packets are copied or simply diverted to the control plane. For IPv4, the packet signature can include IP source and/or destination address, the TCP or UDP source and/or destination port, protocol, MAC source and/or destination, etc. Oplet services have been written that use the JFWD API to set IP filters in the data forwarding hardware. These filters specify that packets on a particular port are forwarded normally and copied to the control plane for processing by the oplet service.

The underlying code to implement the JFWD API is implemented on the Nortel Accelar routing switch with a silicon-based forwarding plane. Our model supports the Active Networking paradigm in which active packets are only delivered to the EE for processing. In this scenario, the forwarding plane is instructed to divert all UDP packets arriving on the Active Network port to the control plane. We call this the *divert* port because packets arriving on this port are not forward directly by the hardware. They are diverted to the control plane following the Active Network

paradigm of store-execute-forward. However, addressing the next active node to receive an active packet is not required. Packets can be forwarded on the ANEP port and the next active node in the direction of the destination will capture the packet based on the destination UDP port. This is explained more fully in the following ANEP section.

To support a new breed of active network services that do not want to delay packet forwarding, we use the JFWD API to request that active packets are copied to both the forwarding plane output queue at wire speed and also to the control plane. The ANEP service which demultiplexes traditional active packets can also demultiplex the new active packets to the right EE. We define a second active network port similar to the ANEP port called the CarbonCopy (cc) port. The cc-port delivers packets both to the output queue for forwarding and to the control plane for processing by the requesting service. Using the cc-port, we are able to create ORE services (e.g. active applications) that do not delay packet delivery to the destination but provide services the ability to view packets and perform some processing.

In [14], the authors present a control-on-demand architecture in packets are blindly forwarded and control of a flow is performed on a *best effort* basis. Our architecture is suited for this type of flow control since we can dynamically install a controller service which can enable and disable forwarding plane filters to get packets based on packet signatures. The installed service may receive some packets from a flow, identify the type of flow (e.g. RTP), monitor flow performance, and adjust queues sizes or priority queue delivery of these packets to change the flow's performance in some way. For example, a flow which is not receiving enough bandwidth could be put into a higher priority queue or have the ToS bit adjusted based on observed performance or flow classification.

Thus, the cc-port option for active packets creates an opportunity for non-delayed forwarding of active packets while still performing some action to influence flow performance.

C. *Active Network Encapsulation Protocol (ANEP)*

The Active Networking community developed the Active Networking Encapsulation Protocol (ANEP) to provide interoperability among active nodes. ANEP is a packet encapsulation mechanism used to forward active packets to the next active node in the network. An ANEP process on the active node receives packets destined for that node arriving on the ANEP port. The ANEP process delivers the active packet to the correct EE based on the Type ID field in the ANEP header.

Current active network implementations require knowledge of the active node topology since active nodes forward packets to the IP address of next active node. This forwarding mechanism requires the ANEP header to preserve the original IP source and destination of the packet. At each active node, the forwarded packet is addressed with the current active node as the source and the next active node as the destination.

The ORE ANEP service on a node receives packets whose IP destination address is the node and whose port number is the ANEP port. ANEP packets arriving at the ANEP service are evaluated and dispatched to the proper EE. Our ANEP dispatcher implementation dispatches the packet directly to the proper EE based on the ANEP header TypeID field. Each EE is given a queue where active packets destined for them will be stored. It is the EE's responsibility to pull packets out of the queue. If no service is registered to receive packets with a particular ANEP TypeID, the dispatcher will quietly discard the active packet.

With the addition of the JFWD API, the ORE ANEP service can be programmed to capture all packets arriving on the ANEP port regardless of the destination IP address. This is significant because active node topology is no longer required for active nodes to address the packets to the next active node. Active packets can be blindly forwarded towards the final destination with the knowledge that the packet will be received by the next active node along the path. Forwarding to non-JFWD enabled nodes still requires active node topology information. However, the ORE supports EE services that can perform topology discovery tasks as performed on any other active node.

VI. CONCLUSION

We implemented a stable active networks platform capable of concurrently running multiple services on a commercial-grade Gigabit router. The ORE supports the creation of services in Java that are extensible, portable, and easily distributed over the network. The ORE provides a platform for safely testing and deploying Active Network execution environments without disturbing the existing non-Java applications. The ORE provides some safe resource allocation and enforcement within the Java environment creating a platform for running concurrent services, including execution environments. We have implemented the ANEP daemon and the ANTS Execution Environment as an oplet services and shown interoperability with ANTS distribution code. With the addition of the JFWD API, the packet capture and processing modification capabilities are realized at the active node. While the active network architecture is powerful and supports multiple languages, the ORE we have developed is a powerful and stable network services platform operating on a Gigabit IP routing switch.

VII. REFERENCES

[1] P.Bernadat, D. Lambright, and F. Travostino, "Towards a Resource-safe Java for Service-Guarantees in Uncooperative Environments," IEEE Symposium on Programming Languages for Real-time Industrial Applications (PLRTIA) 98, Madrid, Spain, Dec. 98.

[2] AN Node OS Working Group, NodeOS Interface Specification", June 15, 1999

[3] Active Networks Working Group, "Architectural Framework for Active Networks Version 0.9", August 31, 1999

[4] T. Lavian, R. Jaeger, J. Hollingsworth, "Open Programmable Architecture for Java-enable Network Devices", Stanford Hot Interconnects, August 1999.

[5] D. Wetherall et al. ANTS: A Toolkit for Building andDynamically Deploying Network Protocols. OPENARACH'98

[6] S. Bhattacharjee et al. On Active Networking and Congestion. Technical Report GIT-CC-96/02, College of Computing, Georgia Institute of Technology, 1996.

[7] D. L. Tennenhouse and D. Wetherall. Towards an active network architecture. In Multimedia Computing and Networking 96, San Jose, CA, Jan 1996.

[8] D. A. Wallach et al. ASHs: Application-specific handlers for high-performance messaging. In SIGCOMM'96. ACM, 1996.

[9] Y. Yemini and S. da Silva. Towards Programmable Networks. In IFIP/IEEE Intl. Workshop on Distributed Systems: Operations and Management, 1996.

[10] E. W. Zegura et al. How to Model an Internetwork. INFOCOM'96. IEEE, 1996.

[11] B. Bershad et al. Extensibility, Safety and Performance in the SPIN Operating System. In 15th Symp. on Operating Systems Principles, Dec. 1995.

[12] D. D. Clark and D. L. Tennenhouse. Architectural Considerations for a New Generation of Protocols. In SIGCOMM '90, 1990.

[13] K. L. Calvert et al. Modeling Internet Topology. IEEE Communications, June 1997.

[14] G. Hja'lmtysson, S. Bhattacharjee, Control on Demand: An efficient approach to router programmability, April 1999.

[15] D. R. Engler et al. Exokernel: An Operating System Architecture for Application-Level Resource Management. In 15th Symp. on Operating Systems Principles,1995.

[16] M. E. Fiuczynski and B. N. Bershad. An extensible protocol architecture for application-specfic networking. Proceedings of the 1996 Winter USENIX Conference, 1996.

[17] S. Floyd et al. A Reliable Multicast Framework for Light-weight Sessions and Application Level Framing. ACM SIGCOMM'95, 1995.

[18] L.-W. Lehman et al. Active Reliable Multicast. In INFOCOM'98, 1998.

[19] A. Mallet et al. Operating System Support for Protocol Boosters. Technical Report MS-CIS-96-13, CIS Dept., Univ. of Pennsylvania, 1996.

[20] J. Smith et al. SwitchWare Accelerating Network Evolution. Technical Report MS-CIS-96-38, CIS Dept.,Univ. of Pennsylvannia, May 1996.

[21] D. Tennenhouse et al. A Survey of Active Network Research. IEEE Communications Magazine, 1997.

[22] D. L. Tennenhouse and D. Wetherall. Towards an active network architecture. In Multimedia Computing and Networking 96, San Jose, CA, Jan 1996.

[23] Y. Yemini and S. da Silva. Towards Programmable Networks. In IFIP/IEEE Intl. Workshop on Distributed Systems: Operations and Management, 1996.

[24] E. W. Zegura et al. How to Model an Internetwork. INFOCOM'96. IEEE, 1996.

[25] Jonathan Smith, et al, Activating Networks: A Progress Report, COMPUER, April 1999.

[26] C. Hawblitzel, C. Chang, G. Czajkowski, D. Hu, T. von Eicken, "Implementing Multiple Protection Domains in Java", USENIX Technical Conference Proceedings, June 1998