Integrating Active Networking and Commercial-Grade Routing Platforms

R. Jaeger[1,2], S. Bhattacharjee[1], J. K. Hollingsworth[1], R. Duncan[2], T. Lavian[2], F. Travostino[2]
[1]*University of Maryland, College Park, MD 20742*
[2]*Nortel Networks, 4401 Great America Parkway, Santa Clara, CA 95052*
{rfj,bobby,hollings}@cs.umd.edu {rduncan,tlavian,travos}@nortelnetworks.com

Abstract

Current network nodes enable connectivity between end-systems by supporting a static and well-defined set of protocols. The forwarding service provided by these network nodes is fixed, simple, and increasingly being implemented in hardware. Active network nodes, on the other hand, enable the unattended, dynamic instantiation of custom programs into the network node, allowing for the introduction of new protocols and services at runtime. Current prototype implementations of active network nodes achieve this flexibility by injecting a significant amount of software into the forwarding path.

This paper describes an Active Network platform that is ideally suited for integration into modern, commercial-grade network nodes, such as router and switches with silicon-based forwarding paths. This Active Network platform supports the dynamic introduction of application services that can alter packet processing; it comprises the Oplet Runtime Environment (ORE) and the Java Forwarding (JFWD) API. The ORE is the substrate that provides for the secure downloading, installation, and safe execution of network services. The JFWD API is a uniform, platform-independent portal through which software services can control the forwarding path of heterogeneous network nodes. We describe how existing active networking environments can be ported onto this Active Network platform and present performance results for dynamically loaded network services on the Accelar Gigabit Ethernet Routing Switch product.

***Index terms*-- Active Networks, distributed applications, networking protocols, NodeOS, ORE, Programmable Networks, JFWD**

1. INTRODUCTION

Traditional network nodes (e.g. routers on the Internet) enable end-system connectivity and sharing of network resources by supporting a static and well-defined set of protocols. The "virtual machine" defines the service provided by the network to transient traffic at each router; the customization of this machine is strictly limited to the configuration hooks that were envisioned at design time. The trend in commercial-grade routers and switches has been to implement ever more functionality of this virtual machine in hardware; hardware implementations have, in turn, enabled ever faster realizations of network nodes. However, the gain in raw performance due to hardware implementations is—almost by necessity—paired with a loss of customization options supported by the router on the data path. As more of the router's virtual machine is *frozen* in silicon, less are the opportunities to introduce new services inside the network.

1.1 Flexible Forwarding: Active Networks

Active Networks (AN) blurs the dichotomy between transient data packets and strictly node-resident software. Unlike traditional networks, AN enable the introduction of network services "on-the-fly". For instance, AN support per-flow customization of the services provided by a network node, according to the various notions of flow being used. The tenet of active networking is as follows: the utility of the service rendered by the network to individual applications is maximized if applications themselves are given the opportunity to define this service. In their most elaborate form, AN introduce a Turing-complete virtual machine at each router. Network users inject a "program" along with their data into the network. This program defines exactly how the network should process a user's data. Depending on the definition of user and the granularity of customization supported by the network interface, the network service in an active network may even be customized on a per-packet basis.

Obviously, such a broad definition of services supported by the network make the implementations of active network nodes difficult. Depending on the user-network interface, the active network implementation—almost by necessity—has to incorporate a substantial software component into the data path [16].[1] Implementations of traditional and active networks must confront the age-old tradeoff between performance and flexibility. In this paper, we explore one point in the performance-flexibility space: we describe an implementation of active network techniques on a commercial routing platform.

1.2 Active networking applied to commercial hardware

This work captures the experiences and lessons-learned while porting our AN platform to the Nortel Networks Accelar Gigabit Routing-Switch. The primary goal of our work is to build a working platform for implementing programmable services on a commercial-grade, best-of-breed router. In doing so, we have tried to (a) preserve the router hardware fast-path for data packets, and (b)

---

[1] In cases of both traditional and active networks, it may be possible to provide fast and customizable forwarding by incorporating hardware that is both programmable over a relatively fine time-scale and is able to forward packets at line-speeds (e.g. fast and programmable FPGAs).

leverage existing active networking research as much as possible.

Obviously, (a) implies that certain computations that require data plane flexibility are not possible in our implementation. A paramount goal of our work is to identify sets of computations that become possible as additional functionality is placed into hardware. As part of our work, we also identify the broad "classes" of computations that are excluded by our implementation.

To support goal (b), we have implemented a layer over which existing active network implementations can be ported. In active networking parlance, this work does not introduce a new AN execution environment (EE). Rather, we present a Java-based run-time environment (the Oplet Run-time Environment) for security and service management over which existing EEs can be deployed and executed as network services. There is an important point of departure in our approach compared to the current work in the active network NodeOS community: we do not simply *trust* (even) Java-based EEs to conform to the resource limit policies set by the node provider. Instead, the ORE checks and enforces these limits on a per-EE basis at run-time. Further, the ORE can, as a matter of node policy, revoke resources and privileges granted to an EE during its execution. We have ported the ANTS EE to run within the ORE on the Accelar router. Our implementation architecture could allow porting the entire DARPA NodeOS interface onto our platform and support both Java-based and "native" EEs.

1.3 Roadmap

In Section 2 we provide a brief overview of the DARPA active network architecture, followed by a description of the internal architecture of the Accelar router. We discuss the issues that must be resolved before the DARPA AN architecture can be realized on a commercial-grade, hardware-intensive routing platform and describe our mapping of the DARPA AN architecture to the Nortel Networks Accelar Gigabit Routing-Switch. In Section 3 we describe how we realize selected portions of the AN architecture on the Accelar, provide details of each component of our implementation, and describe the interfaces supported by each layer. In Section 4 we present a set of performance results from our implementation. In Section 5 we present related work and compare our implementation with existing work both in an architectural context and with respect to supported in-network computations. We present conclusions in Section 6.

2 BACKGROUND

In this section, we provide a quick synopsis of the DARPA active networking architecture and of the internal architecture of the Accelar platform.

2.1 DARPA Active Network Architecture

Figure 1 shows the node architecture for active networks developed by the DARPA active network research community [1].

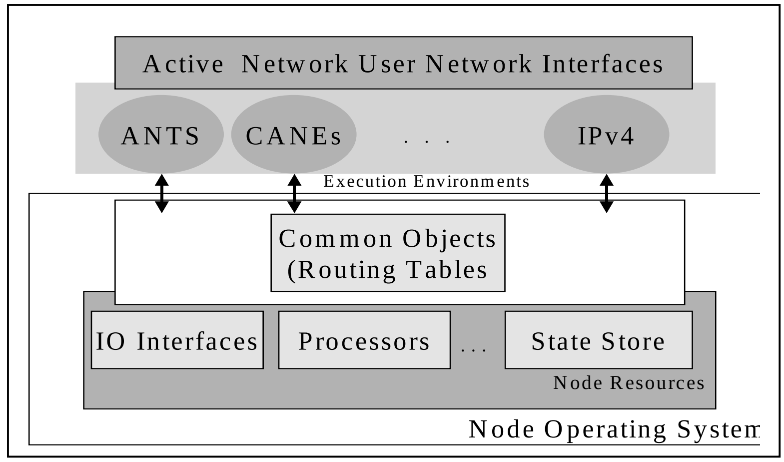

Figure 1: DARPA Active Network Architecture.

The DARPA active network architecture defines a framework through which APIs are exposed to applications by the active network node. The architecture must entertain more than one type of API; the so-called *Execution Environments* (EEs) have the mission to realize the various, specific APIs. The type of APIs and the number of EEs are not necessarily known a priori. This implies that the architecture must define the "virtual machine" supported by network nodes in the network, and the extensibility paradigms associated with such a virtual machine.

EEs can implement a wide range of APIs that exploit different points in the trade-off between performance and flexibility, e.g. IPv4 can be considered a high performance EE that does not provide much flexibility while the ANTS [16] is an EE that provides a Java virtual machine at each node and sacrifices some performance for enhanced flexibility. By supporting multiple EEs, the architecture allows the user of the network to make an application-specific choice in this spectrum between performance and flexibility.

The native computation, communication, and storage resources at an active node are controlled by a *Node Operating System* (Node OS). The node OS provides an interface that exposes the resources available at the active node and mediates access to these resources. The node OS demultiplexes incoming packets to specific EE(s); EEs specify the subset of packets that must be handed-off to them. The node OS also provides support for *common objects* such as routing tables that are likely to be shared across EEs.

2.2 Nortel Accelar Router

The Nortel Networks Accelar family of L3 Routing Switches employs a distributed ASIC-based (Application Specific Integrated Circuit) forwarding architecture with a 5.6-256 Gbps per second backplane. Each ASIC is responsible for four physical 10/100 Ethernet ports or a single gigabit port. The switches scale up to 384 10/100 ports or 96 Gigabit ports (or some combination of the above). There are up to eight hardware-forwarding queues per port corresponding to normal and high priority packets. The hardware is controlled using the VxWorks real-time OS.

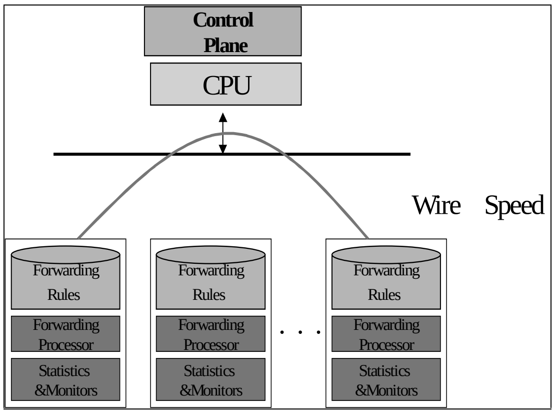

Figure 2: Architecture of the Accelar Router.

Native applications monitor and control the ASIC hardware via a switch-specific API. This API provides access to hardware instrumentation variables to give native applications a one-to-one mapping to hardware functions. For example, the switch hardware provides functionality to set certain bits on an IP packet header that match a specific filter. This functionality is driven through a switch-specific API. Through this API, native applications can install packet filters that can inspect and modify packet headers at wire-speed.

3 IMPLEMENTATION

In this section, we describe how we realize selected portions of the AN architecture on the Accelar platform. In order to transform the Accelar routing switch to be a programmable network service platform, we implemented a run-time environment over which existing active network EEs can be executed. In general, this would require the implementation of the active network NodeOS API over the Accelar embedded real-time OS. However, the AN NodeOS API [2] was still evolving when we started our work and most EEs are implemented either within a JVM[16, 17] or over legacy OS interfaces. Thus, the path we chose was not to port/implement the NodeOS API and to limit support to Java-based EEs. There are two required steps in order for Java-based EEs to execute on the Accelar: (1) a JVM must be ported to the Accelar, and (2) a Java-compatible interface must be provided to the low-level hardware. Figure 2 shows a schematic diagram of the different components of our approach.

The embedded Java VM required by step (1) is a fairly well understood engineering task. JVMs can easily be ported to run as one of VxWorks' "tasks". The service degradation due to a single (possibly malfunctioning or malicious) JVM task on VxWorks is constrained. This is because the JVM runs as just another task in the real-time VxWorks O/S with a fixed and upper-bounded processor share and priority.

Step (2) required us to define a Java API to access low-level forwarding paths like the ones found on the Accelar. As forwarding paths can be heterogeneous—e.g., they can be implemented in software, ASICs, or network processors, and can have vastly different feature sets—it was crucial to come up with a forwarding API of wide applicability—i.e., not a point solution for the Accelar. The details of the resulting Java Forwarding API (JFWD) are captured in Section 3.3.

Though not technically a necessity, we added separate layer between the JVM and the EE. This layer —the Oplet Runtime Environment (ORE)—provides security and management services that may eventually be subsumed by the AN NodeOS and was deemed to be a necessity for the commercial viability of the activated routers. As mentioned before, the ORE enables a stricter intra-node trust infrastructure allowing for different per-node resource allocation policies without cooperation from EE writers. Thus, ORE provides mechanisms for nodes to enforce per-EE resource limits without having to trust the EE. A nice corollary is that the ORE allows multiple EEs (or multiple instantiations of the same EE) to be spawned within a single Accelar with different privileges. In the next section, we present details of the ORE and JFWD API.

3.1 ORE: The Oplet Run-time Environment

The ORE is a platform for secure downloading, installation, and safe execution of Java code (called *services*) within a JVM. A service is a monolithic piece of code that implements specific functionality. A service may *depend* on other services in order to execute. In order to securely download and impose policy, we introduce the notion of "Oplet". Oplets are self-contained downloadable units that embody a non-empty set of services. Along with the service code, an Oplet specifies service attributes, authentication information, and resource requirements. Note that Oplets can encapsulate a service that depends on some other service; in these cases, Oplets also contain dependency information. In general, the ORE must resolve and download the transitive closure of Oplet dependencies before executing a single service.

The ORE provides mechanisms to download Oplets, resolve dependencies, manage the Oplet lifecycle, and maintain a registry of active services. The ORE uses a public-key infrastructure to download "trusted" Oplets. In brief, the security infrastructure provides authentication, integrity, and non-repudiation guarantees on downloaded Oplets. Due to space restrictions, we will not elaborate more on the secure downloading, execution, or resource management features of Oplets.

3.2 Oplet Execution Safety

The ORE must provide safe execution and impose resource limits. As far as possible, the ORE uses the mechanisms provided by the Java language (type safety) and the standard JVM (bytecode verification, sandbox, security manager) to provide execution safety. The ORE controls allocation of system resources by intercepting allocation calls from the service code to the JVM.

To protect itself from denial of service attacks, deadlocks, and unstable states, the ORE implements mechanisms for thread safety and revocation. The ORE controls thread creation by requiring Oplets to request new threads from the ORE. The ORE determines whether to grant the request based upon a node policy that takes into account current thread usage, and the credentials of the requesting Oplet. Once a thread is allocated, however, the current implementation of the ORE has no mechanism in place to account for or limit the consumption of computing resources. In its most general form, the ORE must address denial of service caused by a misbehaving service unduly consuming CPU resources. To handle these issues, the ORE needs JVM support for CPU accounting [14].

Sharing threads between Oplets presents two main problems: (a) deadlock caused by a callee not returning and (b) caller Oplet killing the shared thread while it is executing in a callee Oplet's critical section. The ORE protects itself from the first problem by never interacting directly with any Oplet that it loads. Instead it creates a trusted proxy which the ORE 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 thread call does not return after a conservatively set timeout, a fail-stop situation is assumed and the thread is killed. The second problem is handled by the ORE by revoking Oplet's ability to manipulate a thread's running status.

The ORE uses object revocation to control access to its own resources. If the ORE determines that a specific service is no longer permitted to use a resource reference, the reference can be revoked. For example, a service may carry a "handle" to a data structure exported by another Oplet that no longer exists. The ORE can detect these cases and revoke access to "stale" objects. However, for absolute protection, non-standard support is required from the JVM implementation. Significant modification would include the ability to perform accounting for both CPU and memory consumption and support for per-thread heap allocation and garbage collection [14].

The ORE is currently under active development. At present, it supports secure downloading of services, resolves service dependencies, and allows access to native router functionality through the JFWD API. However, the current ORE version is still vulnerable to several flavors of denial-of-service attacks. These include spurious triggering of the Java garbage collector, memory fragmentation attacks, and stalling finalization of objects[14]. Several memory related safety hazards confronting the ORE will be resolved as JVMs support multiple heaps, revocation and copy semantics of the JKernel [8].

3.3 JFWD: The Java Forwarding API

The JFWD API specifies a platform-independent interface for Java applications to control a virtual forwarding path of commercial-grade strength. The platform-independent nature of JFWD rests upon a) an extensible behavioral model of the forwarding path, and b) an extensible data model of control data (e.g., routing tables) that need to be fed into a forwarding path to affect its behavior. To port JFWD to any given platform, an engineer has to contrast the features of the target forwarding path with the ones modeled in the JFWD API specification, and then proceed to either pruning JFWD classes that are not applicable to the target forwarding path, or sub-classing existing JFWD classes to cope with platform-specific forwarding idiosyncrasies. Subsequent ports of the JFWD contribute feedback and new classes back into the JFWD API specification and its models, and this way the JFWD API evolves towards new forwarding technologies.

A selected set of JFWD classes has been ported to the Accelar. The implementation of these JFWD classes is highly platform dependent; on the Accelar, the JFWD classes turn out to be a wrapper around the hardware instrumentation interface. In the rest of this section, we highlight the main mechanisms that are provided by the JFWD API on the Accelar switch.

Among other things, the JFWD API can be used to instruct the forwarding path to alter packet processing through the installation of hardware filters. The hardware filters execute "actions" specified by a filter policy. On the Accelar, the filter can be based on combinations of fields in the MAC, IP, and transport headers. The policy can define where the matching packets are delivered and can also be used to alter the packet content. Packet delivery options include discarding matching packets (or conversely, forwarding matching packets if the default behavior was to drop them) and diverting matching packets to the control plane. Diverting packets to the control plane allows applications, such as AN EEs to process packets. Additionally, packets can be "carbon copied" to the control plane or to a mirrored interface. Packets may also be identified as being part of high priority flows; these packets can be placed in a static hardware high priority queue.

The filter policy can also cause packet and header content to be selectively altered (e.g. the Type of Service bits on matching packets can be set). The existing hardware is capable of re-computing IP header checksum information at line speeds even if the IP header has to be altered.

3.4 Network services supported by the JFWD and the ORE

In this section, we explore the set of possible and precluded computations on the platform defined by ORE and JFWD API. Note that the ORE does not, a-priori, exclude any computation; instead, it enforces node policy that may cause certain (e.g. processor-intensive) computations to not be started or terminated during execution. Computations are, instead, constrained by the JFWD API since this API defines those capabilities that are exported by the hardware and can be used to build network services.

Thus, some computations, e.g. certain video transcoding techniques that must process every packet, cannot efficiently be implemented in our system regardless of node policy. Not all precluded computations involve data transformation; certain network based anycasting/routing schemes in which a program must be executed to find the outgoing switch port cannot be supported either. The reason is that you are putting inherently slow computations into the forwarding process which is not sustainable at high data rates.

In general, all control-plane only computations, e.g. installing new routing tables or parsing a new ICMP message type, can be rather easily accommodated by the ORE/JFWD API. An important ability enabled by the JFWD API is to *selectively* route (or copy) packets to the control plane —as we will see, this does significantly enlarge the set of services that can be implemented on the Accelar. In the rest of this section, we identify a specific set (non exhaustive) of services that can be implemented using the current version of the JFWD API.

- **Filtering firewall** - One simple application would be a firewall that allows or denies packets to traverse on specified interfaces depending on whether the packet's header matches a given bit mask.

- **Application-specific firewall** - It is relatively straightforward to extend the filtering firewall implement certain application-specific firewalls. For example, an FTP gateway that dynamically changes the firewall rules to allow *ftp-data* connections to a ''trusted" host can be implemented. Security functions like stopping TCP segments with no (or all) bits set can also be dynamically programmed on the Accelar. Almost all modern routers allow for a filtering firewall and application-specific firewall functionality. On the Accelar ORE/JFWD platform, it is imperative to note that these services can now be added, modified, and deleted dynamically, on demand, and without human intervention. The next three services are example of features that, in general, are not yet available in most commercial routers.

- **Dynamic RTP flow identification** - RTP over UDP flows are identified by an ephemeral UDP port number. In general, some host chooses this port number and it is not well known. We have implemented several mechanisms to identify RTP flows traversing the Accelar. Using the JFWD API, control protocol (SIP/RTSP/H.323) messages can be intercepted and parsed for RTP port numbers. We are currently implementing a more dynamic solution that samples packets on specified interfaces and uses probabilistic techniques to identify/mark RTP flows.

- **DiffServ: Classifier, Marker** - The Accelar can be turned into a DiffServ[6] Classifier by suitably programming its hardware filters. Further, the hardware (and in turn, the JFWD API) provides mechanisms to change, at line-speed, selected bits in the IP header. This ability can be used to implement parts of DiffServ ingress/egress marker capabilities on the Accelar. A subtle benefit of this solution is that new firmware or hardware does not have to be shipped each time a new DiffServ scheme/PHB becomes popular. Instead, using existing ORE service instantiation mechanisms, only the service-specific logic has to be uploaded onto the router. This can be accomplished on-line, without interrupting existing flows or services.

- **PGM-like Reliable multicast** - The packet filtering capabilities of the Accelar allows certain packets to be copied on for inspection by the service code. This mechanism can be used to divert (negative) acknowledgements from multicast sessions to the control plane. The service code can, much like the PGM reliable multicast scheme, send one copy of the NAK upstream and suppress duplicate NAKs. Unlike PGM, modulo resource constrains, it is possible to implement reliable multicast services that keep a small packet cache and immediately re-transmit a lost segment. Other services, such as multicast ancestor discovery, can also be efficiently implemented by providing the service code interfaces to the routing table.

We conclude this section with a "wish-list" of a set of functions that, if implemented in hardware and exported by the JFWD API, would enable a new breed of network services. This wish-list is not meant to pinpoint shortcomings of the particular commercial platform that we have used and that is quite good at delivering the services that a traditional customer basis demands. To the contrary, the wish-list represents a constructive hint to those engineering teams endeavoring on new projects explicitly aimed to programmable, active network nodes.

There are two types of functions that are required: functions that are "better" substitutes for existing functionality, and functions that are not available, in any form, in the existing implementations.

There are two main elements in the first class of functionalities with marginal improvement. The first one is the replacement of static priority output scheduler with a better scheduling algorithm (e.g. weighted fair queuing). This would enable RSVP[5] functionality to be implemented as a service. The second one is the ability to discard frames with a given probability function. To implement RED[7] and its variants, a primitive of this kind needs to be added to the discard/divert/forward/copy semantics of permissible actions upon hardware filter match.

We conclude this section with two useful functions that do not exist, in any form, in the Accelar hardware.

- **Token Bucket** - The Accelar hardware could be augmented to provide support for a set number of token buckets, each with a configurable buffer and draining at a specified rate. Obviously, DiffServ shapers and assorted RSVP policy can be implemented using this mechanism.

- **Queue Exposure and manipulation** - The Accelar hardware/JFWD API does not provide any mechanisms for services to get a "sample" or snapshot of the set of currently queued packets. Application-specific congestion control functionality [9] can be implemented using an interface that allows services to periodically check if packets of a certain type (i.e. matching a specified ALF header) from a given flow are queued on an output port. An extension to the queue exposure interface allows services to delete (in the general case, transform) packets that are already queued. The queue exposure and manipulation techniques have been applied to significantly improve end-to-end quality of media streams [4, 9].

4 EXPERIMENTAL RESULTS

In this section, we describe a simple experiment on the Accelar platform. The experimental topology is shown in Figure 3. The hardware used in the experiment included a Accelar 1100B Routing Switch configured with 16 10/100 Mbps port and a 5.6Gbps backplane. The three hosts ran GNU/Linux (kernel version 2.2.5) over 233 Pentium II processors.

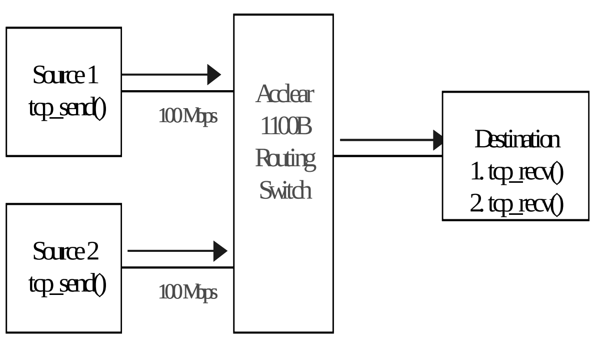

Figure 3: Experimental Setup.

During the experiment, we sent two TCP flows from the two sources to the single destination. In Figure 4 we show the results from a sample run of our experiment: the x-axis corresponds to absolute time at the receiver with respect to a clock that started when the first packet is received; the y-axis corresponds to measured bandwidth in application-space at the receiver averaged over 48 1200 byte segments. Note that for our purposes, the received clocks are synchronized as each process samples current time from the same hardware clock using the Unix `gettimeofday` library call. Once the second flow starts (at time 1.3 seconds), the source TCPs contend for bandwidth on the output link and stabilize their data rate at about 47 Mbps each. We then use an downloaded ORE service (at time 3.8 seconds) to dynamically increase the priority of the second flow. In this case, the service does not implement dynamic flow detection, instead it just uses a fixed source address based filter to discriminate packets from each of the sources. As expected, the received bandwidth on the second (high priority) flow increases and stabilizes at about 70 Mbps. On our testbed, this is the maximum end-to-end bandwidth attainable without any contention. After the second flow ends (at time 7.7 seconds), the low-priority TCP flow can increase its rate and increases its rate up to the expected 70 Mbps.

4.1 Discussion

In isolation, the experiment and the results described above do not qualify as new behaviors. The novelty, however, is derived from the fact that the priority assignment code was installed dynamically on a commercial-grade router capable of stably supporting a large workgroup. In this section, we discuss an immediate application of this functionality that we are using in our own research facilities.

An immediate benefit of on-line identification of flows and dynamic adjustment of packet priority is to support cluster computing. In cluster systems such as Condor[11], NOW[3], Stealth[10], and Linger-Longer[15] workstations are used to run jobs when the computer's primary user is not using their computers. To make these systems usable, the software that runs guest jobs on user's workstations goes to great lengths to ensure that the guest process does not interfere with the primary user. However, until now there has been no clean way to isolate guest use of a workstation from network traffic generated by normal users.

By using active networking at the local area network switch, we can dynamically identify the flows associated with guest jobs. Although these jobs typically have a set of well-known ports, they also can use other network services. To help identify these flows, the cluster scheduler software, can inform the switch when a particular node has started to run a guest process. For some clusters such as Condor and NOW, a node in a cluster is either running guest processes or local process and switches between them on a time-scale measured in minutes. For these types of systems, a simple filter to re-prioritize all traffic from the host running a guest process can be installed by the cluster scheduler. However, system such as Stealth and Linger-Longer allow fine-grain sharing of processors between guest and local processes. To accommodate these systems, the filter needs to be able to identify whether traffic associated from a node is due to a guest process or a local process. To do this a more complete dynamic flow detection —one that can now be implemented on the Accelar— is required.

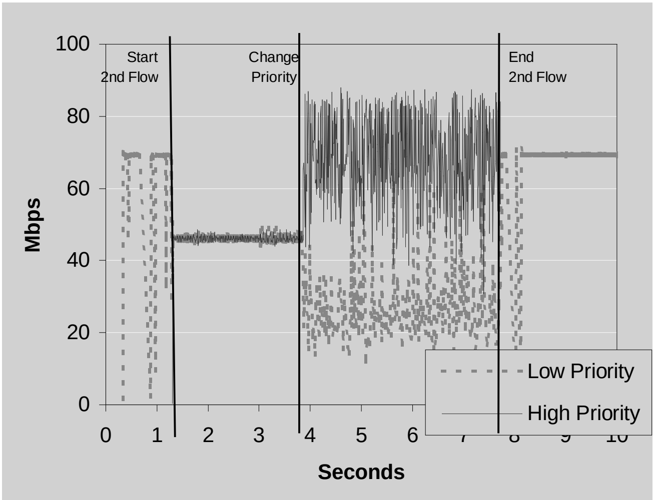

Figure 4: Experimental Results for Dynamic Assignment of Priority to Flows.

5 RELATED WORK

The only other commercial AN platform we know of is from 3Comm[18]. We are not aware other integrated active networking platform implementations on commercial-grade hardware. The active networking work on the Washington University Switch Kit employs locally connected machines as active processors[2]. The Tempest [13] provides a customizable control plane for ATM networks. The basic ideas of high-performance active networking by decoupling the forwarding path from a programmable control plane was introduced, in a software implementation, in the Control-on-Demand (CoD) [9] platform co-developed at AT&T Labs. In this section, we compare our approach to CoD, and discuss how existing active networking research fits within our framework.

The Control-on-Demand platform was developed and implemented over IPv6 as an extension to the Linux kernel [9][3]. Data packets were kept in the kernel in per-flow queues while active control could be applied to the data packets by dynamically loaded "per-flow controllers" that executed in user space. The per-flow controllers affected the data path using the CoD API. A meta-controller loaded each per-flow controller using a signaling protocol. CoD was developed to be specifically mapped onto hardware platforms and its relationship to our work is clear. Services on the Accelar map to per-flow controllers in CoD; the JFWD API on the Accelar maps to the CoD API; the ORE functionality on the Accelar is not completely replicated in CoD, though the meta-controller does provide a subset of the ORE functionality. As CoD was implemented in software; it provides all of the JFWD functionality, and also provides the queue exposure and manipulation facilities on our hardware wish list.

Active networking NodeOS's can potentially be implemented over VxWorks on the Acclear. There is one fundamental problem: the AN NodeOS architecture allows for all packets on specific channels to be delivered to the EE for further processing —this would negate the benefits of the hardware forwarding path available in the Accelar. However, the Accelar provides a perfect platform for implementing fast cut-through paths. The Bowman NodeOS[12] is a particularly good fit as it is specifically supports cut-through paths and is designed as a layer above a host OS that provides low-level hardware access. Thus, Bowman can directly be ported on to the Accelar using VxWorks as its host OS.

For other Node OS efforts, the VxWorks platform already implements much of the required functionality such as memory management. However, it is not obvious if some of the abstractions supported by these systems (e.g. the path abstraction in Scout) can directly be mapped on to the Accelar hardware features.

Java-based EEs can directly be ported on to the ORE. Once a functional AN Node OS has been ported to run over VxWorks, other "native" EEs such as CANEs can be implemented on the Accelar.

6 CONCLUSIONS

We presented a summary of the challenges of bringing Active Networking ideas to current high performance hardware-based routers and switches. In addition, we showed that while it is not currently feasible to support active packets for every packet at line speed on these systems (nor any system), it is possible to exploit existing hardware filtering mechanisms to allow a variety of scenarios that require active functionality on routers. To demonstrate the feasibility of our approach, we presented results from an initial implementation of Active Networking support on the Nortel Accelar. This example showed that it is possible for existing hardware to be able to support active networking environments such as ANTS. Also, we have described how the programmable features of existing ASIC-based hardware forwarding engines can be used as a building block for extensible networks services.

7 REFERENCES

1. "Architectural Framework for Active Networks Version 0.9," . August 31, 1999,.Active Networks Working Group.
2. "NodeOS Interface Specification," . June 15, 1999,.AN Node OS Working Group.
3. R. H. Arpaci, A. C. Dusseau, A. M. Vahdat, L. T. Liu, T. E. Anderson, and D. A. Patterson, "The Interaction of Parallel and Sequential Workloads on a Network of Workstations," *SIGMETRICS*. May 1995, Ottawa, pp. 267-278.

---

[2] See http://www.ccrc.wustl.edu/gigabitkits/kits.html

[3] Control on Demand was co-developed by S. Bhattacharjee

4. S. Bhattacharjee, *Active Networks: Architectures, Composition, and Applications*,Ph.D., Computer Science Department Georgia Institute of Technology,1999.
5. R. Braden, L. Zhang, S. Berson, S. Herzog, and S. Jamin., *Resource ReSerVation Protocol (RSVP)*, RFC 2205, , September 1997.
6. D. Black, S. Blake, M. Carlson, E. Davies, Z. Wang, and W. Weiss, *An Architecture for Differentiated Services*, RFC2475, , Dec. 1998.
7. S. Floyd and V. Jacobson, "Random Early Detection Gateways for Congestion Avoidance," *IEEE/ACM Transactions on Networking*, **1**(4), 1993, pp. 397-413.
8. C. Hawblitzel, C. Chang, G. Czajkowski, D. Hu, and T. v. Eicken, "Implementing Multiple Protection Domains in Java," *USENIX Technical Conference Proceedings*. June 1998.
9. G. Hja'lmtysson and S. Bhattacharjee, "Control on Demand: An efficient approach to router programmability," . April 1999.
10. P. Kruger and R. Chawla, "The Stealth Distributed Scheduler," *ICDCS*. 1991, pp. 336-343.
11. M. Litzkow, M. Livny, and M. Mutka, "Condor - A Hunter of Idle Workstations," *International Conference on Distributed Computing Systems*. June 1988, pp. 104-111.
12. S. Merugu, S. Bhattacharjee, E. Zegura, and K. Calvert, "Bowman: A Node OS for Active Networks," *to appear INFOCOM'2000*.
13. J. E. v. d. Merwe, S. Rooney, M. Leslie, and S. A. Crosby, "The Tempest - A Practical Framework for Network Programmability," *IEEE Network*, **12**(3), 1998.
14. 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)*. Dec. 98, Madrid, Spain.
15. K. D. Ryu and J. K. Hollingsworth, "Linger Longer: Fine-Grain Cycle Stealing for Networks of Workstations," *SC'98*. Nov. 1998, Orlando, ACM Press.
16. D. Wetherall and e. al., "ANTS: A Toolkit for Building and Dynamically Deploying Network Protocols," *OPENARACH'98*. 1998.
17. Y. Yemini and S. da Silva, “Towards Programmable Networks,” in IFIP/IEEE International Workshop on Distributed Systems: Operations and Management, L’Aquila, Italy, October, 1996
18. D. Nessett, “Commercial use of Active Networking,” OpenSIG Workshop, University of Toronto, October 5-6, 1998