A Scalable High-Performance Active Network Node

Dan S. Decasper and Bernhard Plattner, ETH Zurich
Guru M. Parulkar, Sumi Choi, John D. DeHart, and Tilman Wolf, Washington University

Abstract

Active networking in environments built to support link rates up to several gigabits per second poses many challenges. One such challenge is that the memory bandwidth and individual processing power of the router's microprocessors limit the total available processing power of a router. In this article we identify and describe three key components, which promise a high-performance active network solution. This solution implements the key features typical to active networking, such as automatic protocol deployment and application specific processing, and it is suitable for a gigabit environment. First, we describe the hardware of the Active Network Node (ANN), a scalable high-performance platform based on off-the-shelf CPUs connected to a gigabit ATM switch backplane. Second, we introduce the ANN's modular, extensible, and highly efficient operating system (NodeOS). Third, we describe an execution environment running on top of the NodeOS, which implements a novel large-scale active networking architecture called Distributed Code Caching.

ctive networks [1] are packet-switched networks in which packets can contain code fragments that are executed on the intermediary nodes. The code carried by a packet may extend and modify the net work infrastructure. The goal of active network research is to develop mechanisms to increase the flexibility and customizability of the network and to accelerate the pace at which network software is deployed. Applications running on end systems are allowed to inject code into the network to change the network's behavior in their favor.

Until recently, active networking research concentrated on two distinct approaches: "programmable switches" [2, 3] and "capsules" [4, 5]. These two approaches can be viewed as the two extremes in terms of how program code is injected into network nodes. Programmable switches typically upgrade by implicit injection of code by a network administrator. Research in the area of programmable switches focuses on how to upgrade network devices at runtime, on upgrades introduced by administrators which support end system applications (e.g., congestion control for real-time data streams), or on a combination of both. Example applications include self-learning Web caches, congestion control algorithms, online auctions, and sensor data mixing. Since the code is injected out-of-band, programmable switches provide no auto mated on-the-fly upgrading functionality. Capsules, on the other hand, are packets carrying small amounts of program code, which is transported in-band and executed on every node along a packet's path. This approach introduces a totally

new paradigm to packet-switched networks. Instead of "passively" forwarding data packets, routers execute the packet's code. The result of that computation determines what hap pens next to the packet. Applications include simple proof-of-concept ping applications, network diagnostic tools, active multicasting, and more. This approach has the potential to have an enormous impact on the future of networking. However, in the near future, security constraints will cause severe performance problems for capsule-based solutions. Capsules commonly make use of a virtual machine that interprets the capsule's code to safely execute it on a node. In order to ensure security, the virtual machines must restrict the address space a particular capsule might access, thus restricting the application of capsules. We expect network links to be 10 Gb/s or faster in the near future. With an optimistic average packet size of 512 bytes for IP traffic, a router has to process 2.6 million packets/s on every port, which is less than 380 ns/packet. A 300 MHz Pentium™ processor cannot therefore spend, on average, more than 114 cycles to receive, process, and forward a packet just to keep up with the link speed. Even if we assume that a significant fraction of the packets forwarded do not require active processing and can be handled in hardware, it seems obvious that active network architectures based on virtual machines are not well suited to a multigigabit scenario. They may, however, be relevant to network management.

Recently, convergence between the pure "programmable switch" and pure "capsule" approaches became visible. Most

of the research groups involved agree that some sort of code caching makes a lot of sense. The main motivation for this convergence is the realization that potential capsule code is more application-specific than user-specific. In the same way, users usually do not write their own applications, but use off-the-shelf software. They are not expected to inject their own programs into the network, but use code from a set of code modules written by specialists. This allows for various optimizations in the form of caching, as we will see in the related work section. We will also show how our approach aggressively builds on this same realization.

Another very important observation is that the deployment of multimedia data sources and applications (e.g., real-time audio/video, IP telephony) will produce longer-lived packet streams (flows[1]) with more packets per session than is common in today's Internet. Especially for these kinds of applications, active networking offers very promising possibilities: media gateways, data fusion and merging, and sophisticated application-specific congestion control. Both our hardware and software architectures support the notion of flows. In particular, the locality properties of flows are effectively exploited to provide for a highly efficient data path.

This article describes the design of a high-performance active network node (ANN) that supports network traffic at gigabit rates and provides the required flexibility of active network technology for automatic, rapid protocol deployment and application-specific data processing and forwarding. In the next section we look at ongoing research conducted in other laboratories. We then describe the hardware of our ANN, and the software platform running on top of that hardware. We show how we integrate our Distributed Code Caching (DAN) architecture [6] on top of our platform. Finally, we summarize our ideas and take a look at what we plan to do in the future.

Related Work

Active networking research has been ongoing for several years. Various research labs have described and implemented interesting approaches. In this section we give an overview of some of these efforts.

MIT

Tennenhouse et al., [1] proposed "capsules," that is, datagrams carrying small fragments of code, and an implementation in the form of an IP option [5]. The TCL language and a stripped-down TCL interpreter are used to provide safe execution of the code. Some simple well-known network utilities (e.g., traceroute) have been implemented using capsules. So far, this work is mainly focused on a proof of concept for the capsule idea. To overcome security issues, the capsules in this approach are interpreted by a virtual machine. This method of execution introduces performance problems.

Furthermore, this group proposed the ANTS [7] toolkit (downloadable code available). The main goal of ANTS is to provide an architecture for dynamic network protocol deployment. It introduces an optimization to the traditional capsule model. Instead of carrying code in every packet, packets carry pointers to code. This code is then loaded the first time it is needed from the previous hop along a packet's path. Java is used as a programming language for active code. This approach optimizes bandwidth usage with the drawback of a

considerable initial delay. Furthermore, the usefulness of active reliable multicast is shown using the ANTS platform [8]. Results are measured in [9] and show significant performance gains. Reference [9] also proposes various applications of active network technologies like sensor data mixing and inspired us to follow a similar approach. Recently, this group presented PAN [10], an approach similar to ANTS with the difference that instead of Java, machine code is transported in packets. This gives better forwarding performance, but security and interoperability issues are not addressed. Therefore, this approach has yet to prove its practical usefulness.

BBN

Smart Packets [4] is another capsule approach. The main focus is on implementation of extended diagnostic functionality in the network. A new compact programming language called Sprocket is specified and implemented. Its goal is to produce code that is compact enough to fit into an Ethernet packet. Sprocket programs are compiled into "Spanner" code that represents the assembly language for Smart Packets. Spanner code is interpreted in a virtual machine on the node receiving the packet. The programs are authenticated before interpretation and runtime-limited during execution. With its clear focus on network management, Smart Packets provide a very powerful improvement over Simple Network Management Protocol (SNMP), which is used for management of conventional networks. The group is in the process of deploying Smart Packets on the CAIRN network.

Georgia Tech

Zegura et al. [3, 11, 12] introduced a generic view of network code as a set of functions which are called depending on identifiers found in data packets. Application-specific data processing is implemented, as an example, for congestion control for MPEG video streams. The functions referred to in the data packets are loaded out-of-band into the network nodes.

This group also showed how self-organizing network caches could be built using active network technologies. They used simulations and analytical models to evaluate the performance gains offered by caching. This work shows that network caching is an application of active networks worth pursuing. Our system builds in part on the theoretical background and terminology introduced in [3], but significantly extends the system's capabilities.

University of Pennsylvania

The SwitchWare [13] project uses three important components: active packets, switchlets, and a secure active router infrastructure. Active packets are similar to MIT's capsules. Switchlets are dynamically loadable programs that provide specific services on the network nodes. Active packets are programmed in a simple language called Programming Language for Active Networks (PLAN). PLAN programs are strongly typed and statically type-checked to provide safety before being injected into the network. Furthermore, PLAN programs are made secure by restricting their actions (e.g., a PLAN program cannot manipulate node-resident state). To compensate for these limitations, plan programs can call switchlets. Switchlet modules are written in a language (CAML) which supports formal methodologies to prove security properties of the modules at compile time, and no interpretation is needed. The code fragments are authenticated by the developer and explicitly (not on demand) loaded into the switch. At the lowest layer, the Secure Active Network Environment (SANE) ensures the integrity of the entire environment. SANE identifies a minimal set of system elements (e.g., a small area of the BIOS) on

---

[1] Flows are sequences of packets with a common five-tuple of IP header fields consisting of source address, destination address, source port number, destination port number, and protocol.

which system integrity depends and builds an integrity chain with cryptographic hashes on the image of the succeeding layer in the system before passing control to that image. If an image is corrupted, it is automatically recovered from an authenticated copy over the network. Although the project shows very interesting properties, the main problem with this architecture seems to be that PLAN programs are not powerful enough for many applications. Therefore, switchlets have to be installed out-of-band to provide "handles" for the PLAN programs. This makes the system less flexible. Active Bridging [2] is an application of SwitchWare which shows reprogramming of a bridge with switchlets.

Another group at the University of Pennsylvania works on the Programmable Protocol Processing Pipeline (P4, [14]) project. They use reconfigurable field programmable gate arrays (FPGAS) to implement datagram processing functionality (forward error correction in their case) in hardware. Using this kind of hardware support looks very promising for the future of active networking. As far as we know, this is the only group besides us to consider hardware support for active networking at this point in time.

University of Arizona

Scout [15] is a communication-oriented operating system. The kernel is a customized composition of low-level communication primitives that are implemented as modules. Modules implement independent functionality, such as IP, UDP, or TCP protocols. Modules can be combined to form *paths* that build a logical channel over which I/O data flows. *Joust* [16] runs on top of Scout and consists of an implementation of the Java virtual machine (VM) including both the runtime system and a just-in-time compiler. The VM's application programming interface (API) has been extended to interact closely with Scout and to allow applications to access low-level resources. All fixed components are written in C or Java and compiled to machine code ahead of time. The Joust/Scout implementation of ANTS performs two to three times faster than an implementation using Sun's JDK and an off-the-shelf operating system like Linux. The Scout/Joust combination provides the fastest Java environment for active networks documented thus far. However, it still looks like it is not suited for high-volume high-bandwidth traffic.

Columbia University

Netscript [17] is middleware for programming functions of intermediate network nodes. The Netscript programming language allows script processing of packet streams with a focus on routing, packet analyzers, and signaling functions. Netscript programs are organized as mobile agents that are dispatched to remote systems and executed under local or remote control. The goal of Netscript is to simplify the development of networked systems and to enable their remote programming. The Netscript project envisions networks that support flexible programmability and dynamic deployment of software at all nodes. The Netscript language includes constructs and abstractions that greatly simplify the design of traffic-handling software. These abstractions hide the heterogeneous details of networked systems. Protocol messages are defined and encoded as high-level Netscript objects. Netscript programs are message interpreters that operate on streams of messages. Messages can be encoded either as high-level Netscript objects or in a format compatible with existing standards.

Implications on Our Work

The use of interpreted capsules has a lot of potential in areas like network management, where performance is not
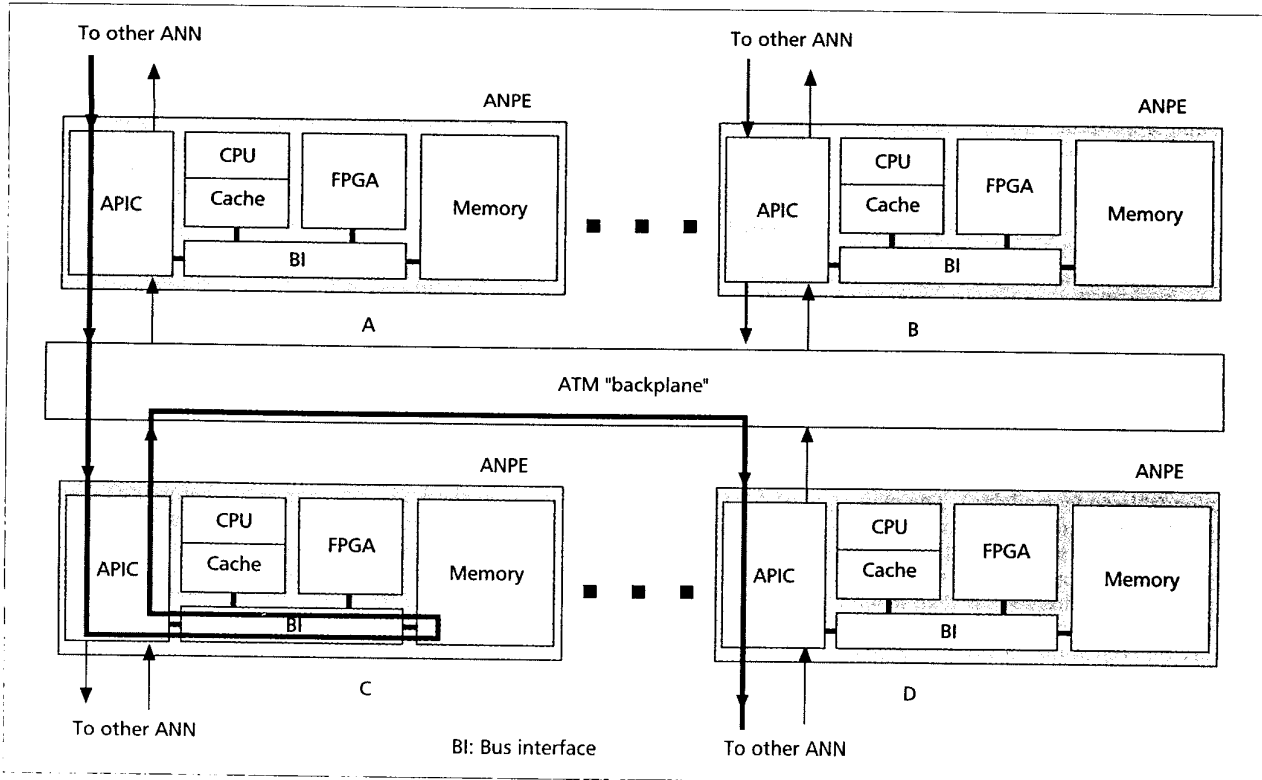
■ Figure 1. *Active network node (ANN)*.

a primary concern. As demonstrated by others, using an interpretation-based approach delivers far more flexible mechanisms,for network management than traditional approaches (e.g., SNMP). Java as the language of choice for capsules provides the advantages of having a lot of market and research momentum. This leads to a variety of available execution environments for Java capsules and increasingly higher quality of these environments. We therefore decided to support the ANTS execution environment on our ANN node architecture. However, for applications requiring a maximal amount of computation performance, minimal latency, and high bandwidth, our DAN architecture promises to be better suited. We describe the ANN's hardware and software architecture in the next two sections before we elaborate on the implementation of DAN on our platform.

The Active Networking Node Hardware

The current trend in Internet router technology is to implement an increasingly higher amount of datagram processing in hardware application-specific integrated circuits (ASICs). Most modern high-performance IP routers are capable of forwarding standard datagrams (without special features like IP options) entirely in hardware. This is required for large backbone routers to keep up with link speeds. These routers typically use ASICS on every port, which have high-bandwidth access to a local table of routes. The central CPU is only involved in processing of nonstandard datagrams and to implement control-path functionality such as routing protocols.

By definition, active networking extends the amount of processing spent on a single packet; and since the processing is application-specific for a potentially significant variety of applications, it cannot be implemented in ASICS. We believe, however, that with the following set of hardware design measures we can optimally address the problem. Active networking router hardware designed for high performance requires:
• A high number of processing elements compared to the number of router ports. A single central CPU attached to a backplane serving all ports is unable to keep up with link speed even for a small number of ports and relatively low bandwidth links (e.g. 10 Mb/s). We use a general-purpose CPU and an FPGA *on every port* of a switch backplane. We call the combination of CPU and FPGA the *processing engine*. The CPU takes care of the majority of active functions applied to a packet, while the FPGA implements functions which are particularly performance-critical in hardware. Both can be programmed on the fly.
• Tight coupling between a processing engine and the network, as well as between the processing engine and a switch backplane. Since the main limiting factors are processing power and memory bandwidth, one has to make sure that these valuable resources are used in the most effective fashion. Two selective measures are applied. First, we benefit from the fact that most network traffic is flow-oriented. Bursts of packets share important forwarding properties that are, once determined, common to all packets of a particular flow. Thus, the majority of non-active packets allow cut-through routing directly through the switch backplane without CPU intervention. Second, by tightly coupling the processing engine to the link, packets arrive at an ANN with minimal overhead through zero-copy direct memory access (DMA).
• Scalable processing power to meet the demands of active processing of packets. Computation on active flows must be evenly distributed over the processing engines available.

The top-level hardware architecture for the ANN is shown in Fig. 1. It is derived from our high-performance IP routing architecture [18], and has been refined and optimized for the purpose of active networks. The node consists of a set of active network processing elements (ANPEs, four in Fig. 1) connected to an asynchronous transfer mode (ATM) switch fabric [19]. ANNs are interconnected only through ANPEs. The scalable switch fabric currently supports eight ports with data rates as high as 2.4 Gb/s on each port. The ANPE comprises a general-purpose processor, a large FPGA (100,000 gates), and memory. The ANPEs are connected to the backplane via the ATM port interconnect controller (APIC) chip [20]. Other devices, like workstations and servers, are connected through a line card directly to the switch fabric (not shown in Fig. 1).

Scalability is guaranteed through:
• The ability to configure any number of ANPEs which can be added to the ANN
• A scalable switch backplane
• A load-sharing algorithm which dynamically distributes active flows over the ANPEs by configuring the corresponding APICs (setting/resetting cut-through switching of selected virtual channels, VCs) in order to move active flows from heavily loaded ANPEs to less loaded ones

Figure 1 shows an example data flow coming into the ANN at ANPE A and going out at ANPE D. The active processing is done in ANPE C since ANPE A is heavily loaded, and the load-sharing algorithm directed the flow to ANPE C, which finally directs the flow to the ANN connected to ANPE D (ANPE A and D switch the flow in hardware without CPU intervention through the APIC). We are developing an intra-ANN protocol to communicate the status of processing engine load between ANPEs on a reserved VC.
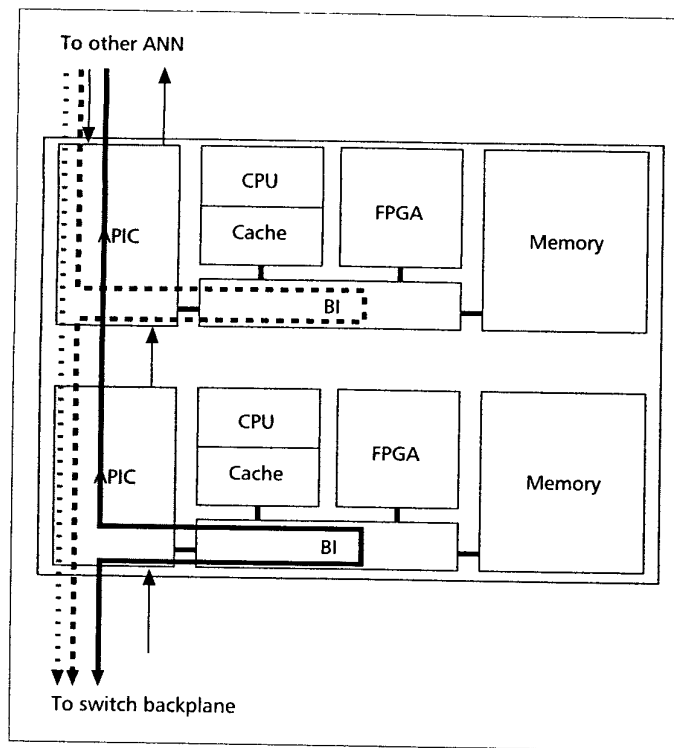
ANPE Architecture

The ANPE consists of an APIC ATM host adapter chip, a Pentium™ CPU, a large FPGA, and up to four gigabytes of DRAM.

The APIC is an ATM host–network interface device with two ATM ports and a built-in peripheral component interconnect (PCI) bus interface. Each of the ATM ports can be independently operated at full duplex rates ranging from 155 Mb/s to 1.2 Gb/s. The ATM cell handling is done entirely in hardware and structured so as not to affect the active networking software subsystem. The APIC further implements VC switching in hardware and is capable of forwarding cells directly without passing them to the processing engine. This is used to implement the load-sharing algorithm as shown in the previous section. It allows the ANPE to forward plain (nonactive) IP traffic without touching the processing engine (also called *cut-through forwarding*), which leaves valuable cycles available for active processing. The APIC provides a powerful host system interface featuring different modes of scatter-gather DMA to provide true zero-copy protocol processing. This allows the implementation of a very high-performance I/O subsystem that supports high bandwidth and low latency.

The processing engine consists of an Intel Pentium™ CPU and a 240-pin FPGA, which can have a very large number of gates (up to 100,000). The CPU runs our NodeOS, which is an optimized version of NetBSD derived from our Router Plugins [21] architecture. We will describe this architecture in the next section. The FPGA can be programmed by the CPU on the fly to implement the most performance-critical algorithms in hardware. The APIC can distribute individual flows to the CPU or the FPGA on a per-VC basis. A packet can first go to the FPGA and then be either passed to the CPU

Figure 2. *ANPE card.*

To other ANN

CPU
Cache
FPGA
Memory
APIC
BI

CPU
Cache
FPGA
Memory
APIC
BI

To switch backplane

or forwarded straight through the APIC to the link, depending on whether there is additional processing required. We expect this combination of FPGA/CPU/cut-through processing to provide excellent performance for software-based packet forwarding.

One or multiple ANPEs can be physically placed on one ANPE card (Fig. 1 shows only one CPU/FPGA/APIC ANPE card). The main advantage of having multiple CPU/FPGA/APIC combinations on one card is that the load-sharing algorithm can save switch backplane bandwidth by distributing the active flows to processing engines on the same ANPE card. Figure 2 shows an ANPE card with two ANPEs and three flows of data packets traversing the card. The three flows are drawn as a dotted, a dashed, and a solid line. The lines show the three different modes of operation of an ANPE card. The flow shown as a dashed line is routed through the first APIC into the first processing engine. The processing engine processes the packets and forwards them to one of the ports of the attached switch backplane. Note that the second APIC on the ANPE card routes this flow in cut-through mode. The flow shown as solid line is an example of a flow that is diverted to the second processing engine on the same ANPE card. This happens in case the first processing engine is heavily loaded. In this case, the load-sharing algorithm picks the second processing engine to process the packets and configures both APICs appropriately. Finally, the dotted flow is cut-through routed through both the APICs without any CPU intervention. This could be either a regular IP flow not requiring active processing or an active flow that is diverted to another ANPE card attached to another switch port. This would happen if both processing engines on the card are heavily used by active flows and there are other CPUs on other ANPE cards in the same ANN which are lightly loaded. Usage of these different modes of operation of the individual ANPE cards leads to optimal performance and scalability of the ANN as a whole.

ANN Software Infrastructure

To utilize the hardware architecture described in the previous section in an efficient way, our software architecture must be optimized to a similar extent for high performance. The main design goal is to provide a highly efficient data path and a flexible control path. All high-bandwidth data path components are implemented in the system's kernel, whereas all management components are implemented in user space. We embedded the architecture described here into our Router Plugins research platform. We showed in [21] that a highly modular router software architecture could be implemented without any significant performance penalties. The architecture described here leverages the results from the Router Plugins work.

Within the active networking community, it is common to distinguish between the "NodeOS" [22] and "execution environments" (EEs) for active networking software architectures. The NodeOS represents the operating system components implementing services such as packet scheduling, resource management, and packet classification, which are independent of a specific active networking implementation. The NodeOS offers these services to the EEs running on top of it. An EE implements active networking protocol-specific processing. For example, there can be an independent EE for ANTS, for Smart Packets, or for SwitchWare.

Our software architecture is shown in Fig. 3. It supports two EEs, namely the ANTS EE and the Distributed Code Caching for Active Networks (DAN) EE [10]. IP can be viewed as another EE with the distinguishing property that the other EEs typically cannot work without IP since they use it for routing and forwarding.

We target the use of ANTS to network management tasks and experimental prototyping of network protocols. ANTS is described in [7]. DAN is described in more detail in the next section. It represents our own active network architecture specially targeted at high-bandwidth low-latency applications.

Before we give an overview of the individual NodeOS components as shown in Fig. 3, we introduce some of the general concepts of this architecture.
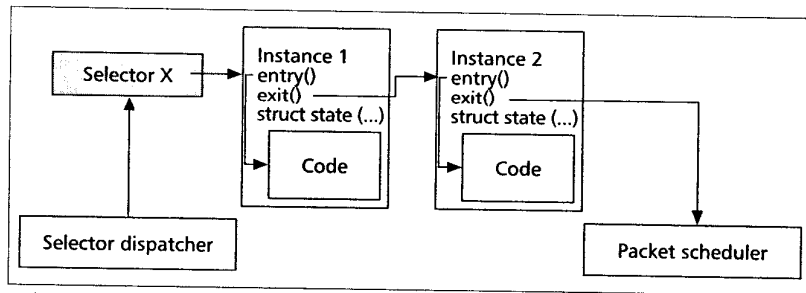
In the context of our architecture, we call code blocks implementing application-specific network functions *active plugins*. Active plugins contain code that is downloaded and installed on the node. The downloading is triggered by the occurrence of a reference in a datagram, as shown in the section on DAN, by a special configuration packet, or by an administrator. Active plugins can create instances. The terminology is intentionally derived from object-oriented programming since the semantics are similar. Instances are flow-specific configurations of active plugins. The individual properties of instances are EE- and plugin-specific. For example, an IP instance consists of the code that forwards the packet and the required information about the interface on which the packet has to be forwarded. However, all instances use the same well-defined API that embeds them into the system. The API consists of an *entry()* and an *exit()* function, among others. The *entry()* function is called to pass a packet to the instance. The *exit()* function is called by the instance when it is done with packet processing.

We make an important distinction between the first few packets of a flow and subsequent packets. The reception of the first packets of a flow usually causes the plugins to create an instance for the new flow. If the packet is passed to multiple instances, these instances are chained together by making
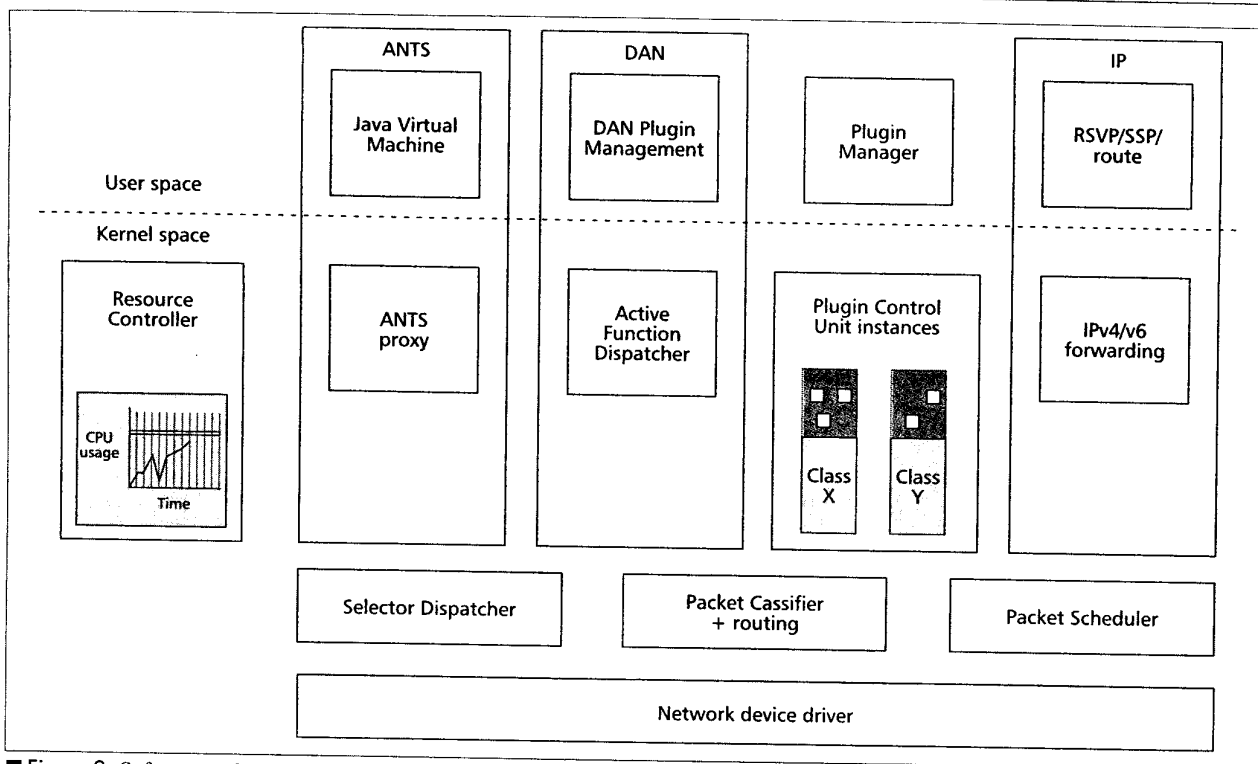
■ Figure 3. *Software architecture.*

the first plugin's *exit()* function call the second plugins *entry()* function, and so on. Every EE is allowed to request a *selector* from the NodeOS to label the chain of instances. The selector is sent to the upstream node by the NodeOS. The upstream node puts the selector in subsequent packets of the same flow, which allows the downstream node to efficiently lookup the state information using hashing, and directly assign the flows state information to the packet. Note that this is similar to tag switching. We introduced the Simple Active Packet Format (SAPF), which describes the format of the selector, in [23]. Figure 4 schematically depicts instance chaining and labeling using a selector. As described below, the selector dispatcher implements flow lookups based on selectors. The packet scheduler is called last to send the packet off to the network.

While per-flow instance creation and management introduces a certain amount of overhead, the payoff in the context of active networks is dramatic for subsequent packets. Other than the selector, no demultiplexing has to be performed, and the operations of instances are reduced to only those which vary from packet to packet of the same flow (e.g., there is no routing lookup). Note that all flow-specific information has soft-state characteristics: it is automatically removed when no packets of a given flow are received for a configurable amount of time.

For the rest of this section, we give a high-level overview of the NodeOS building blocks. The DAN EE is discussed later in this article, the ANTS EE in [7].

NodeOS Components

The kernel consists of the following components (from bottom to top):

■ Figure 4. *Instance chaining and labeling.*

Device Drivers (DD)/Layer 2 Processing — The DD are standard NetBSD device drivers implementing network hardware-specific send and receive functions. They are modified only in two ways. First, to allow packet scheduling, they do not implement packet queues on their own. Second, they pass incoming packets to the selector dispatcher instead of the IP stack if the packet contains a selector. If no selector is present, the packet is first passed to the packet classifier and then to the IP stack.

Packet Classifier (PC) — All packets not carrying a selector are passed to the packet classifier. We implemented a highly efficient packet classifier based on a directed acyclic graph (DAG) in the context of the Router Plugins platform. The packet is classified on a five-tuple of IP header fields and the interface on which it is received. The five-tuple consists of a pair of IP addresses, port numbers, and the protocol used. The PC allocates a flow record for every packet of a new flow. It tags all incoming packets with a flow index (FIX), which is carried in the packet's MBUF[2] and

---

[2] *The MBUF is a data structure that is used to store packets and packet-related information efficiently in BSD-derived operating system kernels.*

points to the packet's flow record. Plugins can access the flow record through the FIX. It stores all flow-related information.

Selector Dispatcher (SD) — The SD scans a data packet for its SAPF selector. Using the selector, it hashes into a table to find the flow's FIX. The same hash table stores the outgoing selector, which will replace the incoming selector in the packet before it is sent to the downstream node. Furthermore, it stores a pointer to the first instance in the flow's instance chain, as previously described and shown in Fig. 4. To negotiate new selector values, the selector dispatcher offers an API to send and receive messages from neighbor nodes.

Packet Scheduler (PS) — The PS in use is a modified version of a Deficit Round Robin (DRR [24]) scheduler which allows bandwidth reservations using filters in addition to fair queuing. Besides DRR, we plan to use a port of CMU's Hierarchical Fair Service Curves (HFSC, [25]) scheduler which represents the state of the art in flow-based packet scheduling, providing hierarchical scheduling and decoupling of bandwidth and delay. Both schedulers can be programmed through the PS API, thus allowing plugins as well as the administrator to reserve resources.

Resource Controller (RC) — The RC keeps track of the CPU cycles and memory consumed by active plugin instances. The RC is responsible for fair CPU time sharing between different instances. Since NetBSD is not a real-time operating system and does not have a preemptive or multithreaded kernel, we implement this by keeping track of the consumed CPU cycles on a per-flow basis. Incoming packets are enqueued in an input queue associated with the flow. Input queues are served in a round-robin fashion. On reception of a packet, the system goes through the following steps:
• Get packet from network card
• Find corresponding flow (call SD or PC)
• Enqueue packet in flow's input queue
• Pick packet to process from the set of all input queues
• Timestamp selected packet with CPU TCS[3]
• Continue processing with that packet
• Before enqueuing the packet on the output queue, read TCS again and add difference to flow total
• Enqueue packet in output queue

The RC implements the selection of the queue. We will investigate different schemes to pick the right packet. Besides per-flow CPU distribution, we will measure what fraction of the total time the system spends processing packets. This will give us an idea of how heavily our system is loaded at any given time. All RCs in an ANN periodically exchange this quantity with each other on a reserved VC to serve as input parameters for the load-sharing algorithm.

The second quantity worth observing is the active plugin's memory consumption. The plugin must be restricted to an upper limit of memory usage by policy or depending on the current average utilization of the networking subsystem. We modify the kernel's memory management to keep track of memory usage on a per-instance basis and possibly deny additional memory to greedy instances. An important resource management issue has to do with the ANN running out of resources such as CPU or memory capacity. Besides implementing the load-sharing algorithm, we explore both policies and mechanisms which can do:

---

[3] *The Pentium TCS register is a 64-bit register which is incremented by one on every clock cycle.*

• Effective "admission control" to ensure sufficient resources for admitted active connections
• Cache management for active plugins to decide which active plugins to replace to create room for the active plugins fetched on demand to be used immediately

Plugin Control Unit (PCU) — The PCU manages plugins, and is responsible for forwarding control path messages (e.g., for instance creation and registration messages) to individual plugins from other kernel components, as well as from user space programs using ANN library calls (we provide the ANN library with our system). Plugins register themselves with a callback function and an identifier (plugin code) when they get loaded into the kernel. All plugins must reply to a defined set of control messages (e.g., messages to create and free instances).

Plugin Manager (PM) — The PM is a user-space utility for configuring the system. It is a simple application that takes arguments from the command line and translates them into calls to the user-space library. This library implements the function calls needed to configure all kernel-level components. In most cases, the plugin manager is invoked from a configuration script during system initialization, but it can also be used to manually issue commands to various plugins. This is especially useful to test new active plugins.

The design of this software architecture is clearly driven by the goal of allowing efficient implementations of EEs and active plugins. While we favor our DAN architecture with this design, we make all components accessible to other EEs through a well-documented API.

The Distributed Code Caching Approach to Active Networking

Distributed Code Caching for Active Networks (DAN) was described in [6]. Since [6] was published, the architecture has matured significantly. We review the important ideas here, give an update reflecting the latest developments, and elaborate on how DAN is embedded into the NodeOS described in the previous section.

Before we present our approach, we review the basic requirement for active networking. It is to allow users and applications to control networking nodes, and how their packets are processed and forwarded. This necessitates computing and programmability at each network node. However, this requirement should not considerably degrade the performance of an EE through excessively complex and inefficient security mechanisms. In other words, per-packet processing should not require a long and inefficient software path. Thus, the fundamental challenge that high-performance active networking poses can be summarized as follows:

Allow relocating part of the processing from the end systems into the network; however, minimize the amount of processing on a single node and make the processing as efficient as possible while keeping the flexibility and customizability that the active networking paradigm introduces.

We believe that our architecture, which we call "Distributed Code Caching," does just that.

Distributed Code Caching

To overcome the performance-related problems that will exist, at least in the near term, for capsules, we think that a combination of the programmable switch and capsule approaches is very appealing. We replace the capsules' program code by a reference to an active plugin stored on a code server. On a reference to an unknown code segment in a router or an end

system, the code is automatically downloaded from a code server. It is important to note that the code fragment or plugin is dynamically linked and executes like native code on the router/node, and thus runs as fast as any other code. The security issues are addressed by usage of well-known cryptography techniques (as explained later in this section); thus, our scheme does not require slow virtual machines. This introduces some restrictions regarding the authorship and source of active network code for the benefit of security and performance, but we believe this to be an appropriate compromise. To explain our idea, we first analyze the layout of data packets and network nodes common in today's networks.

Each network node typically supports a particular set of functions that may be applied to data packets. One or more unique identifiers in the packet's header identify these functions. When a packet is processed, the referenced functions are applied to the data of the packet. The packet's data can be viewed as the input parameter to a function. An Ethernet packet, for example, contains a unique identifier for the upper-layer protocol (0x0800 for IPV4, 0x08dd for IPV6). By demultiplexing an incoming packet on this value, the kernel decides to which function or set of functions the packet gets passed next. Packets consist of a finite sequence of such identifiers for functions and input parameters. The functions are normally daisy-chained in the sense that one function calls the next according to the order of the identifiers in the data packet. The first function is determined by the hardware (the interface on which the packet is received) and the last function or set of functions is implemented in the application consuming the packet. Each of the functions may also decide not to call the next function for several reasons (e.g., forwarding the packet to the next hop and thereby skipping over the higher-layer data or detection of errors). Depending on the type of node and the packet's content, only a subset of these functions may be called. It is possible to think of these function identifiers in data packets as "pointers" to code fragments. In today's systems, the code that implements these functions must be available on the node processing the packet. In our system, the node contacts a "code server" for the necessary code in case the node does not already have the required code locally. In contrast to data servers, which provide a client with "passive" data, code servers provide active plugins stored in a database of code fragments. A code server is a well-known node in the network that provides a library of possibly unrelated functions for different types of operating systems from various developers.

Figure 5 shows an example of a client downloading real-time video through an active network node (ANN) which involves several steps:

• The ANN receives the connection setup request and forwards it to the video sever.
• The video server replies with a packet referencing a function for congestion control of the video stream.
• The ANN does not have the code referenced in its local cache and therefore contacts a code server for the plugin.
• The ANN receives the active plugin, dynamically links it in its networking subsystem, possibly applies the data to the congestion control function, and forwards the packet to the client. Once the plugin is downloaded, it is stored locally on the ANN, removing the need to download the same active plugin in the future.

Distributed code caching features the following important properties.

Active Plugins in Object Code — It is important to note that the active plugins offered by the code server are programmed
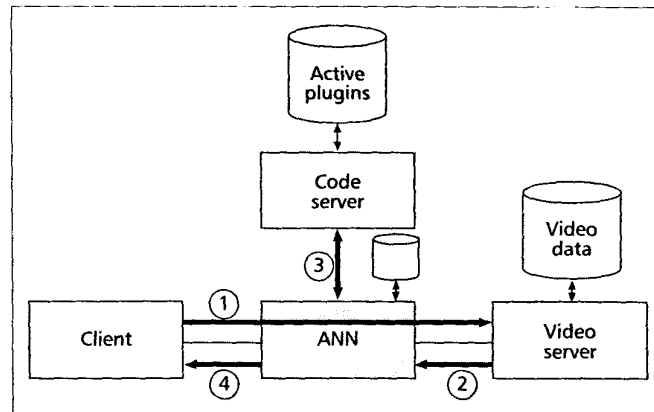

■ Figure 5. *ANN downloading an active plugin.*

in a higher-level language such as C and compiled into object code for the ANN platform. Once the node loads the functions, they are in no way different than the ones compiled into the network subsystem at build time. For example, the functions have as much control over the network subsystem's data structures as any other function in the same context, and they are executed as fast as any other code.

Security Addressed by Use of Well-Known Cryptography Techniques — All active plugins stored on code servers are digitally signed by their developers. Code servers are well-known network nodes that authenticate the active plugin when sending them to ANNs. ANNs load only authenticated, digitally signed active plugins and have the capability to check the plugin's sources and developer before installing and running the plugin locally. The security problem is reduced to the implementation of a simple policy rule on the node which lets it choose the right code server and a database of public keys to check the developer's signature and the code server's authentication.

Minimization of Code Download Time — Downloading of active plugins from code servers to ANNs happens infrequently, since this is necessary only on the first occurrence of a new function identifier. Still, some attention has to be paid to minimization of the download delay. Download time can be minimized by the following three architectural considerations:

• Probe packet: By using one or multiple probe packet(s) sent from the server to the client before sending data packets. The probe packet triggers the downloading of active plugins in parallel on all routers along the packet's path. The total end-to-end code download delay can be reduced approximately to the time a single ANN requires for the download.
• Optimal code server arrangement: Code servers should be as "close" as possible to the ANN. They can be put into a hierarchy similar to DNS servers where the root code servers get their active plugins from the programmers of the plugins.
• Minimizing the distance between the ANN and the code server: The ANN can reach the code server in different ways. One or multiple unicast addresses of code servers can be configured on the ANN (again similar to DNS). The responsibility for finding a suitable code server is up to the administrator selecting the unicast address. Another possibility is the usage of anycast or multicast addresses, which would delegate the responsibility for finding the best code server to the anycast or multicast routing. Last, the data server itself could maintain a database of active plugins, and the ANN could query the data server for the
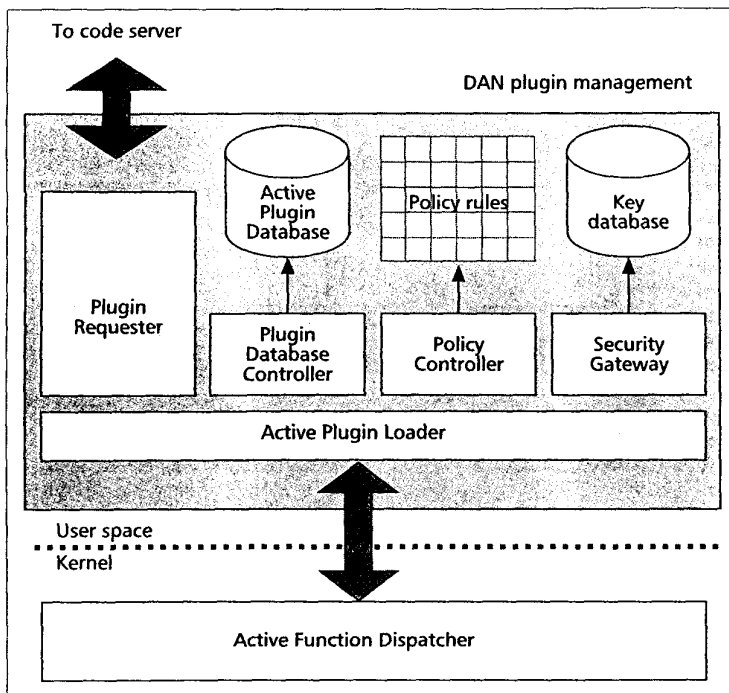
■ Figure 6. *DAN EE.*

plugin. This solution has the advantage that no particular configuration information for code servers must be present on the node, and there is no need for a particular active plugin distribution infrastructure. It seems very natural that the organization providing a data server makes sure that not only end systems (e.g., by offering a plugin for a Web browser) but also all nodes along the data packet's path are able to process the data offered in the best possible way. One disadvantage of this solution is that it allows only one level of authentication (the developer's digital signature). Also, code plugins may come from "nonoptimal" sources in respect to bandwidth and delay since all routers along the packet's path might access the same data server instead of possibly utilizing parallel active plugin downloading through a hierarchical infrastructure.

Policies — We support policies for at least two important system properties: acceptance of specified active plugins and plugin caching behavior.
* Acceptance policies: Policies regarding acceptance of active plugins on nodes are desirable. Even if plugin sources and the plugins themselves are authenticated, network administrators may wish to restrict the set of developers from which they accept active plugins or exclude certain specific active plugins because of undesired behavior.
* Caching policies: Developers are able to set timeouts for active plugins. When a timeout is reached for a plugin on an ANN, the ANN would delete it and refetch it on the next reference in a data packet. This mechanism can be used to deploy prototype versions of new network protocol implementations. Timeouts can be set to infinity for nonexpiring plugins. In addition to these developer-specified timeouts, the administrator of an ANN can set timeouts for an individual plugin, for sets of plugins, or for all plugins in the ANN. These timeouts force a periodic refetch of specified plugins independent of a developer's settings. Such a refetch can be set to happen out-of-band to provide the node with the most recent plugin version independent of references in data packets.

By installing a set of rules on the ANN we will enable both mechanisms that implement these policies to be configured.

Integration with Existing Network Protocols — Our active networking support can be provided in existing protocols by introducing new function identifiers at different layers. We will briefly look into three possibilities:
* Data link layer: Using ATM as a link layer, an application of function identifiers could be to use them in the link layer control (LLC) SNAP field.
* Network layer: IP options, which are defined for both IPv4 and IPv6, might be the most common way to introduce new function identifiers ([3] and [5] also describe a way to use IP option fields for AN). Options are commonly used to specify unusual datagram processing (e.g., source routing). Whereas option usage in IPv4 is very limited because the total option length is 40 bytes, IPv6 introduces a very flexible option concept by allowing very long and unlimited numbers of options wrapped into either hop-by-hop or destination option extension headers. Using IPv6 options has the further advantage that the system can benefit from the option type semantics, which specifies the node's behavior in case it does not recognize the option type (skipping over the option/discarding the packet/sending ICMP message to the source).
For function identifiers in IP options in the context of connection-oriented protocols like TCP, active plugin download can take place on connection setup. When the data server replies with a SYN back to the client requesting the connection, it may include a packet containing the probe function identifier and optional configuration information, and force the nodes along the path to fetch the active plugins. In BSD 4.4, the retransmission delay for the client initiating the SYN is approximately 6 s before the next SYN is sent out, and the client waits 76 s before considering the request failed. Thus, on-demand loading of the corresponding code should be possible with out the need for changing the end node's TCP.
* Transport layer: For functions to be executed on end systems only, function identifiers can occur in addition to or instead of the usual transport layer function identifier for TCP/UDP.

We described the most important properties of distributed code caching in this section. The usage of caching techniques and active plugins in machine code promises to deliver a significant performance improvement over traditional interpretation-based capsules. Next, we describe the implementation of our DAN EE on top of the ANN NodeOS described earlier.

The DAN Execution Environment

The DAN EE is shown in Fig. 6. It consists of the active function dispatcher (AFD) in the kernel and the DAN plugin management (DPMgmt) in user space. The DPMgmt consists of the active plugin loader, policy controller, security gateway, plugin database controller, and plugin requester. Next follows a description of the individual components.

Active Function Dispatcher (AFD) — The AFD scans a data packet for function identifiers and passes the packet to the corresponding active plugins. Although, as shown in the pre-

vious section, it is possible to integrate DAN function identifiers at various places in datagrams, in its current implementation the AFD looks for DAN function identifiers only in IPv6 hop-by-hop options. We embed DAN function identifiers in ANEP [26] packets. ANEP is a packet header defined by the Active Networks Working Group to precede EE-specific packet headers. The IPv6 option processing code passes the packet to the AFD when it finds DAN function identifiers following an ANEP header. While scanning the packet, the AFD calls the functions referenced until they have all been called or one of them has dropped the packet. The AFD keeps track of all known function identifiers and a pointer to their corresponding instances on a per-flow basis. As the instances are called, the AFD chains them together as shown earlier. Note, however, that the AFD calls the instances *individually*. The chaining has no effect on the AFD. It is only considered if the packet contains a selector. In that case, it is passed directly to the first instance of the chain without going through the AFD.

In case of a previously unknown function identifier, the AFD contacts the active plugin loader (APL, described next) in order to request the corresponding active plugin. It temporarily suspends packet processing for the packet causing the call, enqueues the packet in a dedicated queue, and proceeds with the next packet received. The AFD maintains its own queue of active packets with previously unknown function identifiers. On a call from the APL, the AFD resumes processing of the enqueued packet by calling the newly installed active plugin.

Active Plugin Loader (APL) — This component interfaces with the networking subsystem in the kernel through a dedicated socket interface similar to the way *routed* does in BSD UNIX. On the occurrence of an identifier for a previously unknown active plugin, the AFD requests the corresponding active plugin from the APL. The APL talks first to the PC to find out whether the request for the plugin is permitted. If the PC positively acknowledges the request, the APL requests the plugin from the plugin database controller, which maintains the database of local active plugins. If the plugin is locally available, it is immediately loaded into the networking subsystem through the plugin control unit. If not, the APL contacts the plugin requester to send out a request to a code server. On reception of the plugin from a code server, the APL passes it to the security gateway for origin and signature control. If the active plugin's signature is valid and its origin proven, it gets passed down to the plugin control unit for integration into the networking subsystem. Previously suspended packet processing then resumes. Finally, the plugin is passed to the plugin database controller, which includes it in its local database of active plugins.

Policy Controller (PC) — The PC maintains policy rules set up by the node's administrator. As previously described, we implement both acceptance policies as well as caching policies for active plugins.

Security Gateway (SG) — The SG is responsible for checking the integrity and origin of active plugins. Which security checks are required is determined by the configuration of the ANN in question. The SG maintains a database of public keys. We implement full RSA public-key encryption using the RSAREF [27] library as a basis. This library provides both MD5 one-way hashing as well as RSA public key encryption. MD5 one-way hashing will be used to generate a plugin-specific hash key that is then digitally signed with

RSA using the developer's private key. The code server transmits the active plugin together with the signed hash to the ANN. On reception of the plugin, the ANN calculates the plugin's MD5 hash, decrypts the received hash with the developer's public key, and compares both hashes. If they match, the plugin is assumed to be valid. Security extensions [28] to the Domain Name System (DNS) provide support for a general public key distribution service which we use to distribute the developer's and code server's public keys. The stored keys enable ANNs to learn the authenticating keys of code servers in addition to those for which they are initially configured. Keys associated with the code server's and developer site's DNS names can be retrieved to support our system. An ANN can learn the public key of a code server or a developer by either reading it from the DNS or having it statically configured. To reliably learn the public key by reading it from the DNS, the key itself must be signed with a key the ANN trusts. The ANN must be initially configured with at least the public key of one code server and developer. This is typically the network subsystem developer's key. From there, the node can securely read the public keys of other code servers and developers. As an alternative to the DNS security scheme, we will use IP security [29], which is mandatory for IPv6 implementations, for code server authentication. This allows simple and streamlined security for ANNs, which do not check developer signatures but depend on code server authentication only.

Plugin Database Controller (PDC) — The PDC efficiently administers the local database of active plugins. Plugins are indexed by developer codes and function identifiers for fast access. If the ANN offers code server service to other ANNs, the database may contain active plugins for foreign hardware and software architectures. The active plugins are stored together with the developer's digital signature and the originating code server authentication. Typically, the plugins are stored on nonvolatile storage, like disks or flash RAM. This is not required for regular ANNs since they can refetch active plugins from code servers on system startup, thus saving the download time during packet processing. Plugins come with an expiration date that can be set to infinite. Administrators are able to set a global expiration time for unused plugins independent of plugin-specific settings. On expiration of a code plugin, the PDC deletes it from its nonvolatile storage and reloads it on request.

Plugin Requester (PR) — The PR is responsible for requesting active plugins from code servers and replying to such requests. The request is either unicast, multicast, or anycast, depending on the local configuration as described using a lightweight non-connection-oriented protocol (e.g., UDP/IP) to ensure stable operation under heavy load. Since we use a datagram-oriented protocol instead of a connection-oriented one, loss of either the active plugin request or the reply (the active plugin itself) may occur. In this case, the packet causing the request is dropped by the networking subsystem with a possible error message sent to the source. The download of the active plugin is reinitiated the next time the same function reference occurs in a packet.

The Code Server

Code servers feature a database of active plugins for possibly different operating systems and hardware architectures. They get their plugins either manually by configuration through a system administrator or automatically from an upper-level code server in the code server hierarchy.

Code servers are network nodes running a version of the DPMgmt. We are carefully separating ANN NodeOS-dependent layers (e.g., the part that communicates with the AFD) from the rest of the DPMgmt to be able to port the DPMgmt to a wide range of platforms. Since most of today's router hardware lacks large mass storage, end systems similar to database servers are better suited to be configured as code servers. We are exploring the usability of publicly available relational and object-oriented database technologies to efficiently store active plugins.

Plugin Packages

The code for active plugins is stored on code servers and in the local active plugin database on the individual ANNs together with additional data. The code for multiple active functions can be wrapped together into one active plugin package. A download of such a package would install multiple active functions on the ANN. This is useful for strongly correlated active functions such as the four options for IPv6 mobility support [30]. On occurrence of a function identifier in a data packet, not only is the referenced active function implementation downloaded and installed, but also one or more others, since they will very likely be referenced in the future. This mechanism requires only one download cycle and one application of related functions such as security checks for the whole package. A package contains at least:

- The code for one or more active functions
- The developer's digital signature
- The code server's authentication information
- Configuration information for the ANN which will be passed to the plugin after installation and a set of rules regarding storage of the plugin package, like its expiration date

Conclusions and Future Work

We elaborate on three key factors critical to pave the way for active networking in a gigabit environment. First, we elaborate on a gigabit hardware platform that allows high-performance active networking in a scalable fashion combining off-the-shelf and customized hardware components. Second, we described our NodeOS supporting both the active network paradigm as well as the hardware in a highly optimized way. Third, we reviewed a new active networking execution environment called Distributed Code Caching which we believe to be especially well suited to our high-performance hardware and software environment.

We are currently in the process of implementing the system described here and expect to have a prototype up and running by the time this article is published. As a next step, we will start working on a variety of applications.

One of the most promising applications of our environment is automatic protocol deployment. We plan to show automatic upgrading of IPv4 nodes to IPv6 nodes as well as on-the-fly revision of IPv6 implementations. Without active networking, it is extremely hard (if not impossible) to change a protocol once it is deployed. What is needed is a fully automated way to deploy and revise new protocols. This would allow for incremental refinement of specifications and implementations based on real-world experience, which has not been possible so far. Consider, for example, IPv6 options: in IPv6, only a very small set of IP options is specified in the base specification [31]. These options are mainly used to pad data packets to certain sizes in order to align them at word boundaries; however, the protocol supports new options in a modular way. An arbitrary number of IPv6 options can follow the IPv6 header in the form of hop-by-hop or destination

options. It is expected that these new options are "hardwired" into an IPv6 implementation. To support new options such an implementation would require recompiling, which is difficult and time-consuming to do in an operational network. With the system described here, a new active plugin for an IPv6 option is downloaded on demand from a code server the first time the new option is referenced, and the active plugin is stored for later use in the local cache. We will show this feature for options required for IPv6 mobility support and possibly others.

Other applications planned for implementation on the platform include application-specific reliable multicast, congestion control for real-time audio/video, media gateways, and sensor data mixing.

References

[1] D. Tennenhouse et al., "A Survey of Active Network Research," IEEE Commun. Mag., Jan. 1997.
[2] D. Alexander et al., "Active Bridging," Proc. SIGCOMM '97, Sept. 1997.
[3] S. Bhattacharjee et al., "An Architecture for Active Networking," Proc. INFOCOM '97, Apr. 1997.
[4] B. Schwartz et al., "Smart Packets for Active Networks," Jan. 1998; available at http://www.net-tech.bbn.com/smtpkts/smart.ps.gz
[5] D. Wetherall and D. Tennenhouse, "The ACTIVE IP Options," Proc. 7th ACM SIGOPS Euro. Wksp., Sept. 1996.
[6] D. Decasper and B. Plattner, "DAN: Distributed Code Caching for Active Networks," Proc. INFOCOM '98, Apr. 1998.
[7] D. Wetherall et al., "ANTS: A Toolkit for Building and Dynamically Deploying Network Protocols," Proc. IEEE OPENARCH '98, Apr. 1998.
[8] L. Lehman, S. J. Garland, and D. Tennenhouse, "Active Reliable Multicast," Proc. INFOCOM '98, Apr. 1998.
[9] U. Legedza, D. Wetherall, and J. Guttag, "Improving the Performance of Distributed Applications Using Active Networks," Proc. INFOCOM '98, Apr. 1998.
[10] E. Nygren, "The Design and Implementation of a High Performance Active Network Node," MIT Master's thesis, Feb. 1998.
[11] S. Bhattacharjee, K. Calvert, and E. Zegura, "Self-Organizing Wide-Area Network Caches," Proc. INFOCOM '98, Apr. 1998.
[12] E. Zegura, "CANEs: Composable Active Network Elements," Georgia Tech; http://www.cc.gatech.edu/projects/canes/
[13] D. Alexander et al., "The SwitchWare Active Network Architecture," IEEE Network Special Issue on Active and Programmable Networks, May/June 1998.
[14] I. Hadzic and J. Smith, "On-the-fly Programmable Hardware for Networks," Proc. GLOBECOM 1998.
[15] D. Mosberger and D. Peterson, "Making paths explicit in the Scout operating system," Proc. 2nd Symp. Op. Syst. Design and Implementation, Oct. 1996.
[16] J. Hartman et al., "Joust: A Platform for Liquid Software," IEEE Network, July 1998.
[17] Y. Yemini and S. daSilva, "Towards programmable networks," IFIP/IEEE Int'l. Wksp. Dist. Sys.: Operation and Mgmt., Oct. 1996.
[18] G. M. Parulkar, D. C. Schmidt, and J. S. Turner, "aItPm : A Strategy for Integrating IP with ATM," Proc. SIGCOMM '95, Aug. 1995.
[19] T. Chaney et al., "Design of a Gigabit ATM Switch," Proc. INFOCOM '97, Apr. 1997.
[20] Z. Dittia, J. R. Cox, Jr., and G. Parulkar, "Design of the APIC: A High Performance ATM Host-Network Interface Chip," Proc. INFOCOM '95, Apr. 1995.
[21] D. Decasper et al., "Router Plugins — A Software Architecture for Next Generation Routers," Proc. SIGCOMM '98, Sept. 1998.
[22] Active Networks Working Group, "Architectural Framework for Active Networks," Aug. 1998; http://www.cc.gatech.edu/projects/canes/arch/arch-0-9.ps
[23] C. Tschudin and D. Decasper, "Simple Active Packet Format (SAPF)," Experimental RFC, Aug. 1998.
[24] M. Shreedar and G. Varghese, "Efficient Fair Queuing using Deficit Round Robin," Proc. SIGCOMM '95, Aug. 1995.
[25] I. Stoica, H. Zhang, and T. S. E. Ng, "A Hiearchical Fair Service Curve Algorithm for Link-Sharing, Real-Time and Priority Services," Proc. SIGCOMM '97, Sept. 1997.
[26] D. Alexander et al., "Active Network Encapsulation Protocol (ANEP)," RFC DRAFT, July 1997.
[27] RSA Library, ftp://ftp.rsa.com/rsaref
[28] D. E. Eastlake, "Domain Name System Security Extensions," RFC 2065, Jan. 1997.
[29] R. Atkinson, "Security Architecture for the Internet Protocol," RFC 1825, Aug. 1995.
[30] D. Johnson and C. Perkins, "Mobility Support in IPv6," draft-ietf-mobileip-ipv6-07.txt, Nov. 1998.

[31] S. Deering and R. Hinden, "Internet Protocol, Version 6 (IPv6), Specification," RFC 1883, Dec. 1995.

Biographies

DAN S. DECASPER (dan@tik.ee.ethz.ch) (http://www.arl.wustl.edu/~dan) is a research assistant in a Ph.D. program at the Computer Engineering and Network Laboratory (TIK) at ETH, Zurich, Switzerland. He received his diploma from ETH in 1996. His research focuses on new network protocols, modular high-performance router architectures, and active networking. In 1997 and 1998 he was invited by Washington University in St. Louis, Missouri, to spend a total of 12 months at the Applied Research Laboratory (ARL). He contributed significantly to the design and development of a modular gigabit IPv6/ATM platform named Router Plugins. Currently, he is working with ARL on a DARPA-funded active networking project for which he co-authored the project proposal.

GURU M. PARULKAR (guru@arl.wustl.edu) (http://www.arl.wustl.edu/~guru) is a professor of computer science and director of the Applied Research Laboratory (A Network and Multimedia Systems Laboratory) at Washington University, St. Louis. He is also a co-founder of a high-end routing/switching company called Growth Networks Inc. (GNI). He received a Ph.D. in computer science from the University of Delaware in 1987 and was the recipient of the Frank A. Pehrson Graduate Student Achievement Award in Computer and Information Sciences. At Washington University, he has led several projects that emphasize prototyping of high-performance systems. Example systems include a gigabit Active Networking Router, a gigabit ATM desk-area network, an I/O subsystem in NetBSD that can support QoS guarantees, a scalable multimedia-on-demand storage server and service, IP switching with integrated services, reliable multicast protocols, and network monitoring, visualization, and control systems.

SUMI CHOI (syc1@arl.wustl.edu) (http://www.ccrc.wustl.edu/~syc) received a B. S. degree from Yonsei University in 1994 in mathematics. From 1995 to 1997 she worked on the implementation and development of graph drawing at Brown University. She is currently in a Ph.D. program in computer science at Washington University, specializing in computer networks. Her research interests include development and design of active network applications.

JOHN D. DEHART (jdd@arl.wustl.edu) (http://www.arl.wustl.edu/~jdd) received his M.S. in computer science from Washington University in 1984. Since 1989, he has been working in the Applied Research Laboratory at Washington University and is currently a senior research associate. His areas of interest include active networking, multimedia networking, and software control of ATM networks.

TILLMAN WOLF (wolf@arl.wustl.edu) (http://www.ccrc.wustl.edu/~wolf) received his B.S. degree in computer science from the Universität Stuttgart, Germany. He completed his M.S. in 1998 and currently pursues a doctoral degree at Washington University. He works as a graduate research assistant in the Applied Research Laboratory. His research interests are in the fields of high-speed and active networking.

BERNHARD PLATTNER [M] (plattner@tik.ee.ethz.ch) (http://www.tik.ee.ethz.ch/~plattner) is a professor of computer engineering at ETH Zurich, where he leads the communication systems research group. His research currently focuses on applications of communication systems and higher-layer protocols, and on integrated services networks. Specifically, he is involved in the development of multimedia applications for high-speed networks and is exploring new approaches to protocol and network engineering, such as active networks. In his earlier work he directed research in message handling systems and distributed directories. He is a member of the ACM and the Internet Society.