A Scalable, High Performance Active Network Node

Dan Decasper[1], Guru Parulkar[2], Bernhard Plattner[1]
[dan|plattner]@tik.ee.ethz.ch
guru@arl.wustl.edu
[1]Computer Engineering and Networks Laboratory, ETH Zurich, Switzerland
Phone: +41-1-632 7019 Fax: +41-1-632 1035
[2]Applied Research Laboratory, Washington University, St. Louis, USA
Phone: +1-314-935 4586 Fax: +1-314-935 7302

April 1998

Abstract

Active networking in an environment built to support link rates up to several gigabits per second prodices many challenges. One such area is that the total available processing power of a router is limited by the memory bandwidth and individual processing power of the router's microprocessors. In this paper, we identify three key components, which combined promise an innovative active network solution. This solution implements the key features typical to active networking such as automatic protocol deployment and application specific processing and is also suitable for a gigabit environment. First, we make use of a new large scale architecture which we call Distributed Code Caching. Second, we propose the Active Network Node (ANN), a scalable, high performance hardware platform based on off-the-shelf CPUs connected to a gigabit ATM switch backplane. Third, we describe a modular, extensible, highly efficient software framework which implements Distributed Code Caching and load-balancing among the ANN's CPUs, thus maximizing the computational performance of the ANN.

***Key words**: active networks; distributed code caching; gigabit active networking; scalable active network node*

1 Introduction

Active networks [37] are packet-switched networks in which packets can contain code fragments that are executed on the intermediary nodes. The code carried by the packet may extend and modify the network 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 to their favor.

Active networking research concentrates on two commonly separated approaches: "programmable switches" [2, 6, 34] and "capsules" [27, 38]. These two approaches can be viewed as the two extremes in terms of program code injection into network nodes. Programmable switches typically "learn" by implicit, out-of-band injection of code by a network administrator. Research in the area of programmable switches focuses on how to upgrade network devices at run time or 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. End system applications supported are self-learning web caches, congestion control algorithms, online auctions, and sensor data mixing to name just a few. Since the code is injected out-of-band, programmable switches provide no automated, on-the-fly upgrading functionality. Capsules 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 and the result of that computation determines what happens next to the packet. Applications include simple proof-of-concept implementations like packet pingers which ping-pong data packets between network nodes, network diagnostic tools, active multicasting and more. This approach has an enormous potential impact for the future of networking. In the near future, security constrants will cause sever performance problems for capsule-based solutions. They commonly make use of a virtual machine that interprets the capsule's code to safely execute on a node. The virtual machines must restrict the address space a particular capsule might access to ensure security, which restricts the application of capsules. We expect network links to be 10 Gb/s or faster in the near future. With an average packet size of 1KB for IP traffic, a router has to process 1.3 million packets per second on every port which is less than 760 nanoseconds per packet. A 300 MHz Pentium processor can therefore not spend on average more than 231 cycles to receive, process, and forward a packet just to keep up with the link speed. It seems obvious that active network architectures based on virtual machines are not well suited to a multi-gigabit scenario.

This paper proposes the design of a high performance Active Network Node (ANN) that supports network traffic at gigabit rates and provides the flexibility of Active Network technology for automatic, rapid protocol deployment and application specific data processing and forwarding. The project is funded by DARPA and we expect to prototype the various components of this approach over the next couple of years. In section 2, we explain a new architectural approach to active networking and how it fits into related active networking research. Section 3 describes the hardware of the ANN and section 4 the software platform running on top of that hardware. Section 5 outlines different possible applications we plan to implement for the ANN to demonstrate its capabilities. Section 6 relates our approaches to those of others and section 7 concludes our work.

2 A New Approach to Active Networking

Before we present our approach, we review the basic requirement for active networking, which 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 the execution environment 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 endsystems 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 the active networking paradigm introduces.

We believe that a new innovative architecture, which we call “Distributed Code Caching” does just that. This architecture has been described in [12], but we review some of the important properties in this subsection, since it is one of the three corner stones of our approach.

2.1 A New Active Network Architecture: Distributed Code Caching

To overcome the performance related problems which will exist in the near term for capsules, we think that a combination of the programmable switch and the capsule approaches is very appealing. We replace the capsules’ program code by a reference to an active plugin stored on a code server, building a distributed code cache. 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, it runs as fast as any other code. More over, the security issues are addressed by usage of well known cryptology techniques (as explained later in this section), and thus, our scheme does not require slow virtual machines. It introduces some restrictions regarding the authorship and the 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 which may be applied to data packets. These functions are identified by one or more unique identifiers in the packet’s header. When a packet is processed, the functions identified 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 a 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 the packet is received on) 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 the 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 which implements these functions must be available on the node processing the packet. In our proposed system, the node contacts a “code server” for the necessary code in case the node does not already have it 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 which provides a library of possibly unrelated functions for different types of operating systems from various developers.

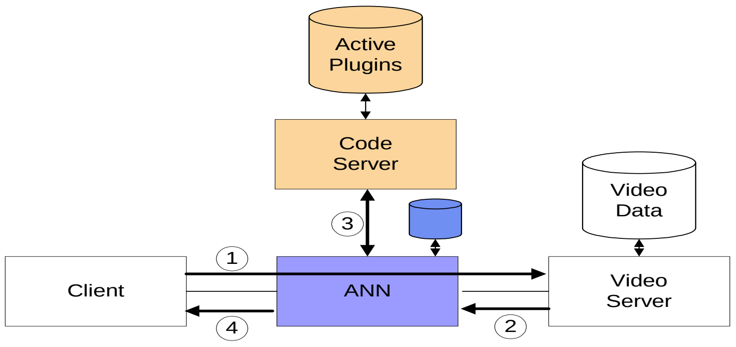

Figure 1: ANN downloading an active plugin

Figure 1 shows an example of a client downloading real-time video through an Active Network Node (ANN) which involves several steps: (1) The ANN receives the connection setup request and forwards it to the video sever; (2) the video server replies with a packet referencing a function for congestion control of the video stream; (3) the ANN does not have the code referenced in it's local cache and therefore contacts a code server for the plugin; (4) 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 preventing other downloads for 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 in a higher level language such as 'C' and compiled into object code for the ANN platform. Once the functions are loaded by the node, 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 usage of well known cryptology techniques**: All active plugins stored on code servers are digitally signed by their developers. Code servers are well known network nodes and authenticate the active plugin when sending them to ANNs. ANNs load only authenticated, digitally signed active plugins and have the possibility to check the plugin's sources and developer before installing and running the plugin locally. The security problem is reduced to the installation of a simple policy rule on the node which let's 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**: Although downloading of active plugins from code servers to ANNs happens very infrequently since this is necessary only on the first occurrence of a new function identifier, some attention has to be paid to the minimization of the download delay. Roughly speaking, the delay is the product of the number of hops from the ANN to the code server and the link bandwidth. Download time can be minimized by the following three architectural considerations:
  (1) "Probe" packet: By using a "probe" packet sent on connection setup from the server to the client which triggers the downloading of active plugins in parallel on all routers along the packets path. The total end-to-end code download delay can be reduced approximately to the time a single ANN requires for the download.
  (2) Optimal code server arrangement. Code servers should be as "close" as possible to the ANN. They might be put into a hierarchy similar to DNS [25] servers where the root code servers get their active plugins from the programmers of the plugins.
  (3) Minimizing the distance between the ANN and the code server: The ANN can reach the code server through 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 anycast/multicast routing. Last, the data server itself could maintain a database of active plugins and the ANN

could query the data server for the 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 plug-in 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 "non optimal" 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 plan to support policies for at least two important system properties: Acceptance/denial of specified active plugins and plugins caching time-out behavior.
  (1) 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 they accept active plugins from or exclude certain specific active plugins because of undesired behavior.
  (2) Caching policies: Developers are able to set time-outs for active plugins. When a time-out 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. Time-outs can be set to infinity for non-expiring plugins. In addition to these developer specified time-outs, the administrator of an ANN can set time-outs for an individual plugin, for sets of plugins or for all plugins in the ANN. These time-outs force a periodic refetch of specified plugins independent of a developers 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 the three possibilities:
  (1) Data link layer: Using ATM as a link layer, an application of function identifiers could be to use them in the LLC SNAP field.
  (2) Network layer: IP options, which are defined for both IPv4 and IPv6, might be the most common way to introduce new function identifiers ([6] and [38] 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 option/discarding packet/sending ICMP message to 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 [35], the retransmission delay for the client initiating the SYN is approximately 6 seconds before the next SYN is sent out and the client waits 76 seconds before considering the request as failed. Thus, for functions which execute only on routers, on-demand loading of the corresponding code should be possible without the need for changing the end node's TCP.

(3) 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 the traditional, interpretation-based capsules. In order to yield the desired result of active networking at gigabit speed, an innovative combination of high-performance hardware components with a carefully designed software framework is also required. In the next two sections, we describe this hardware and software platform.

3 A Scalable High Performance Active Networking Node

We believe that an active networking platform designed for high performance requires:

- Tight coupling between a processing engine and the network as well as between the processing engine and a switch backplane. Since the limiting factors in gigabit networks, especially with active networking support, are processor 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 which are, once determined, common to all packets of a particular flow. Thus, the majority of 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 DMA.
- Scalable processing power to meet the demands of active processing of packets. Processing power must be augmentable to meet the changing requirements of the network load. Computation on active flows must be evenly distributed over the CPUs available.

Over the past few years, we have been prototyping technological components that enable building of an Active Networking Node (ANN). These components meet both of the above requirements extremely well in a cost effective fashion.

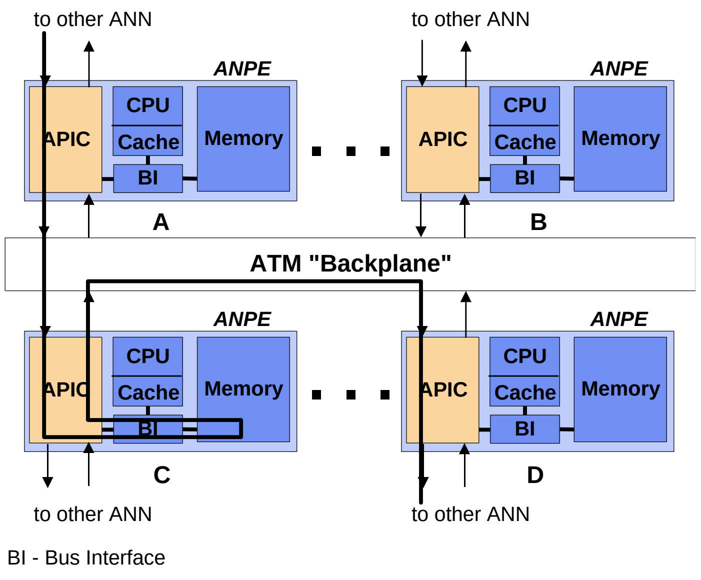

Figure 2: Active Network Node (ANN)

The top level hardware architecture for the Active Network Node (ANN) is shown in Figure 2. It is derived from our high performance IP routing architecture [28] and has been refined and optimized for the purpose of active networks. The node consists of a set of Active Network Processing Elements (ANPE, four in Figure 2) connected to an ATM switch fabric [9]. ANNs are interconnected only through ANPEs. The scalable switch fabric currently supports eight ports with a data rate as high as 2.4 Gb/s on each port. The ANPE comprises a general purpose processor and can implement fast execution of application specific data processing. Each ANPE card has a main data path that passes through our ATM Port Interconnect Controller (APIC) chip. The APIC chip supports zero-copy semantics, so that no copying is required to deliver data from the network to the application. It provides two bidirectional ports, VC switching in hardware and per VC ATM cell pacing. The ANPEs are connected to the backplane via the APIC. Other devices like workstations and servers are connected through a line card directly to the switch fabric. The ATM cell handling is done entirely in hardware and structured so as not to affect the active networking software subsystem. Scalability is guaranteed through (1) the ability to configure any number of ANPEs which can be added to the ANN; (2) a scalable switch backplane; (3) a load sharing algorithm which dynamically distributes active flows over the ANPEs by configuring the corresponding APICs (setting/resetting cut-through switching of selected VCs) in order to move active flows from heavily loaded ANPEs to lesser loaded ones. Figure 2 shows an example data flow coming into the ANN at ANPE A 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). This load-sharing scheme can be further optimized by putting multiple ANPEs (physically on the same card) daisy-chained through the APIC on a single switch port. The load-sharing algorithm can then first distribute the load to the other ANPEs on the same port before diverting to other ANPEs on the switch, thus saving switch bandwidth. ATM virtual circuits can also be configured on the fly to provide streamlined handling of information flows which do not require active processing.

3.1 ANPE Architecture

The top level ANPE architecture has already been described. In this subsection, we focus on the various subsystems and components that are used to implement the ANPE architecture. One central component is the APIC (ATM Port Interconnect Controller). The APIC is an ATM host-network interface device with two UTOPIA (Universal Test and Operations PHY Interface for ATM) compliant ATM ports and a built-in PCI (Peripheral Component Interconnect) bus interface. Both 32-bit and 64-bit wide versions of the PCI bus are supported. Each of the ATM ports can be independently operated at full duplex rates ranging from 155 Mb/s to 1.2 Gb/s.

The default behavior of this device is to forward without modification any cells arriving at one of the two input ATM ports to the other ATM port for output (this is known as the *transit* path). However, if cells arrive on one of a configurable set of receive (Rx) virtual connections (VCs), such cells are passed to the CPU for processing (this is the *receive* path). This device can also take data streams from the CPU and generate corresponding cell streams on a configured set of transmit (Tx) VCs: these cells are then forwarded to the specified ATM output port(s), where they are interleaved with transit cells before being transmitted (this is the *transmit* path). The APIC device incorporates AAL-5 (ATM Adaptation Layer-5) functionality, and is capable of segmentation and reassembly (SAR) processing at the maximum bus rate (1.05 Gb/s peak for PCI-32 and 2.11 Gb/s peak for PCI-64). All packets are segmented and reassembled in *external memory* accessed over the PCI bus — in most cases, this is the ANPE's main memory. The APIC provides a powerful host system interface. Different modes of scatter-gather Direct Memory Access (DMA) provide the flexibility and versatility needed to support true zero-copy protocol processing on the ANPE and allow the implementation of a very high performance I/O subsystem that supports high bandwidth and low latency. All DMA modes employ the same general technique of buffer descriptor chaining, where chains of 16 byte buffer descriptors residing in external memory are used by the APIC to locate the buffers (also in external memory) into which received data should be deposited, or from which data that has to be transmitted can be read. This allows very efficient implementation of vertically integrated active software layers (for example various application specific protocol stacks or slices of protocol stacks).

One off-the-shelf Intel Pentium CPU, the APIC and memory build the core hardware components of a single ANPE. One or multiple ANPEs can be physically placed on one ANPE card (Figure 2 shows only one-CPU/APIC ANPE cards). The main advantage of having multiple CPU/APIC combinations on one card is that the load-sharing algorithm can save switch backplane bandwidth by distributing the active flows to CPUs on the same ANPE card. Figure 3 shows an ANPE card with two ANPEs and three flows of data packets traversing the card. The three flows show the three different modes of operation of an ANPE card. The blue flow is routed through the first APIC into the first CPU. The CPU processes the packets of the flow in software 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 green flow is an example of a flow which is diverted to the second CPU on the same ANPE card. This happens in case the first CPU is heavily loaded. In this case, the load-sharing algorithm picks the second CPU to process the packets and configures both APICs appropriately. Finally, the red flow is cut-through routed through both the APICs without any CPU intervention. This could either be a regular IP flow not requiring active processing or an active flow which is diverted to another ANPE card attached on another switch port. This would happen if both CPUs 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. The usage of these different modes of operation of the individual ANPE cards lead to optimal performance and scalability of the ANN as a whole.

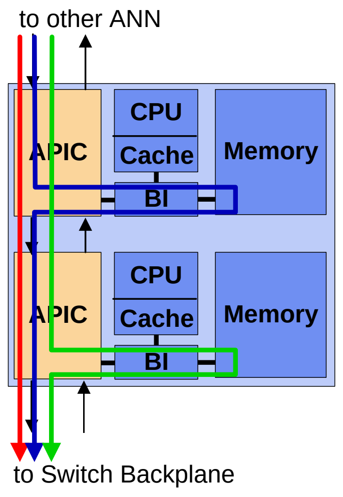

Figure 3: ANPE card

3.2 ATM Switch Architecture

The second key component of the gigabit ANN is the Washington University Gigabit Switch (WUGS). This switch is illustrated in Figure [4]. The system compromises a multistage switching network that implements dynamic routing of cells to evenly balance the load from all inputs over the entire network. The network supports gigabit operation by striping cells across four parallel planes (each cell is divided and transferred in parallel through all four planes simultaneously, minimizing latency). The network also supports an elementary ‘copy-by-two’ operation, which together with a novel cell recycling scheme permits an incoming cell to be copied F times with $\log_2 F$ passes through the network. This architecture is the only one known which provides nonblocking multipoint virtual circuits with optimal scaling properties. The architecture is implemented using a set of three custom integrated circuits, one implementing the *Switch Elements* making up the switching network and the other two implementing the *Port Processors* that interface to the external links and which perform cell routing (using virtual circuit identifiers) and cell buffering. The Input and Output Port Processor chips, like the APIC, implement the Utopia interface, making it possible to directly connect an APIC to a switch port processor. An important feature

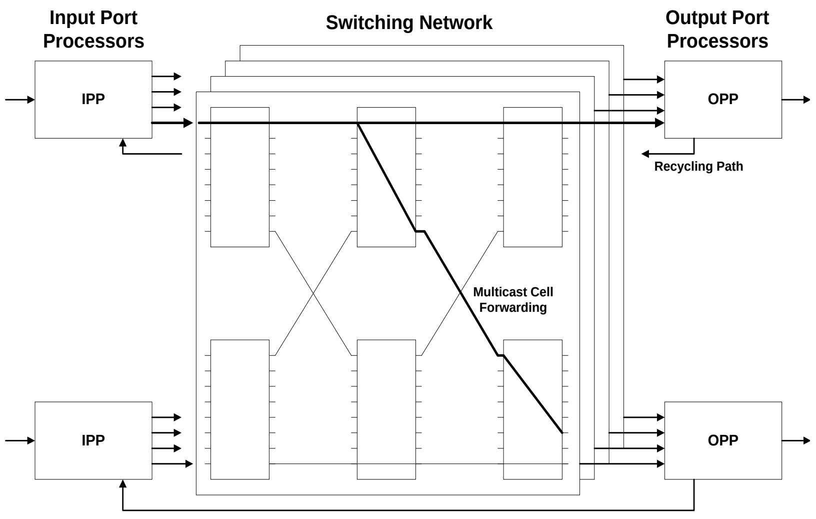

Figure 4: WUGS Switch Architecture

of the ATM switch which we plan to use in the ANN is the fact that it can be configured by sending control cells to it over its ATM links. For those ports which are enabled to receive them, cells with a special VPI/VCI combination are interpreted as switch control cells and the contents of their payloads determine what functions are to be performed and at which of the switch's ports they are to be carried out. When a control cell is received from a link, it is forwarded to the port on which it is to operate and the target port processor carries out the required operation. Most commonly, the requested operation is to read or write an entry in the port's virtual circuit routing table. However this same mechanism can be used to configure certain hardware options or to access cell counters (these are maintained on both a link and per virtual circuit basis) or other status registers. While in typical ATM switch applications, there would be a single processor managing the switch resources, there is no intrinsic reason that these resources cannot be managed in a distributed fashion by a collection of ANPEs that setup and modify virtual circuits quasi-independently. So long as the ANPEs keep each other informed of their resource usage and respond cooperatively when conflicts arise, it should be possible to distribute control in such a way that non-conflict-producing decisions are very fast, allowing virtual circuits to be established in under 100 microseconds.

Washington University has an eight port prototype version of the switch up and running.

4 Software Infrastructure

The software framework running on each ANPE will be embedded into our Router Plugin [11] research platform. The Router Plugin platform provides a flexible framework to investigate services and mechanisms including resource management, packet filtering, and packet scheduling for multimedia/multicast applications. It implements the IPv6 protocol suite in a modular environment, enabling the user to plug in experimental versions of companion protocols for evaluation. It uses a Berkeley NetBSD UNIX kernel and an industry standard PC as its platform and may be configured to serve as an end system or as a router. Although modular, the platform allows packet processing at high speeds easily saturating an OC-3 link running on a P6/200. The experience gained from the development of a modular high-performance router platform will be critical for the development of the active network platform described here, which is inherently modular. Several integral components of the Router Plugin platform, like the Association Identification Unit (AIU) for fast flow detection and packet filtering, will be used and will reduce development time significantly. The software architecture is shown in Figure 5. Most of the complex components not involved in the data path will run as a daemon process in user space and only performance-critical modules directly involved with packet processing in the data path are planned to be integrated in the networking subsystem itself. The software architecture comprises the following (main) components:

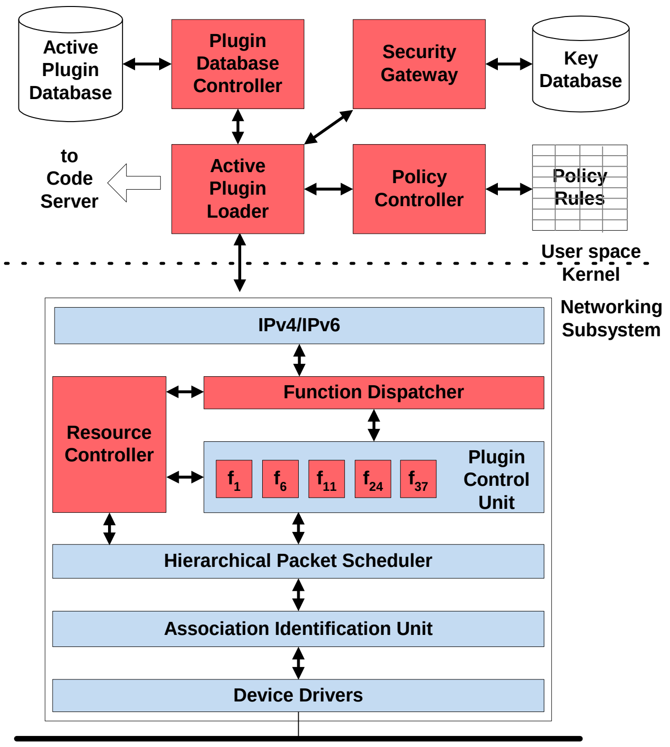

Figure 5: ANPE software architecture

- an **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. We plan to enhance the interface in order to allow fully asynchronous event processing. On the occurrence of an identifier for a previously unknown active plugin, the networking subsystem requests the corresponding active plugin from the APL. After the request is sent to the APL, the networking subsystem enqueues the packet, which caused the request for the active plugin, in a dedicated queue and resumes processing of the next packet in the input queue. All subsequent packets of the flow causing the request are enqueued in the same queue. The APL talks first to the Policy Controller to find out whether the request for the plugin is permitted. If the Policy Controller 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 by the Plugin Control Unit. If not, the APL sends out a request to a code server. The request is either unicasted, multicasted or anycasted depending on the local configuration using a lightweight non-connection oriented protocol (e.g. UDP/IP) to ensure stable operation under heavy load. On reception of the plugin from a code server, the APL passes it to the security gateway for origin and signature control (or either depending on configuration). 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 integrates it in it's local database of active plugins. Since we use a datagram oriented protocol instead of a connection-oriented protocol, both the loss of the active plugin request or the loss of the reply (the active plugin itself) may occur. In this case, the packet causing the request would be 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.
- **Policy Controller (PC)**: the PC maintains a table of policy rules set up by the nodes administrator. Policy rules include which developer's active plugins are allowed on the ANN. It may be desirable for an administrator to restrict the set of possible active plugins based on the active plugin's developer, e.g. to exclude all plugins but the ones implemented by the developer of the networking subsystem itself. Specific undesired plugins can be excluded as well. Further policies specify the set of code servers from which plugins are accepted. In addition to these restrictions, the administrator may specify which local plugins should be loaded into the networking subsystem at boot time of the ANN. If an excluded active plugin is referenced in a data packet, the packet might be dropped or ignored and an error message may be sent to the source of the packet.
- **Security Gateway (SG):** the SG is responsible for checking the integrity and/or 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 plan to implement full RSA public-key encryption [30] using the RSAREF [31] library as a basis (which is free of charge for educational institutions). This library provides both MD5 [29] one-way hashing as well as RSA public key encryption. MD5 one-way hashing will be used to generate a plugin specific hash key which 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 developers public key and compares both hashes. If they match, the plugin is assumed to be valid. The latest proposed security extensions [16] to the Domain Name System (DNS) provide support for a general public key distribution service which we plan to use to distribute the developer's and code server's public keys. The stored keys enable ANNs to learn the authenticating key 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 either by reading it from the DNS or by 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 configured with at least the public key of one code server and/or developer as a starting point. This would typically be the network subsystem developer's key. From there, it can securely read the public keys of other code servers and developers. Alternatively to the DNS related security scheme, we plan to use IP security [5] which is mandatory for IPv6, for code server authentication. This allows simple and streamlined security for ANNs which depend on code server authentication only and which do not want to check developer signatures.

- **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 a non-volatile storage, like disks or flash RAM, but 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. We plan to experiment with various expiration policies for the plugins. Plugins come with an expiration date which can be set to "infinite". Administrators are able to set global expiration time for unused plugins independent of plugin specific settings. On expiration of a code plugin, the PDC deletes it from its non-volatile storage and reloads it on request from the APL.
- **Function Dispatcher (FD)**: the FD is the heart of the kernel resident portion of the system. It operates on every individual packet, therefore it's fast implementation is very critical. It scans a data packet for function identifiers and passes the packet to the corresponding active plugins or to standard protocol implementations for previously installed protocols like IP. In today's networking subsystem, the detection of function identifiers is daisy-chained in a sense that the implementation of a protocol at layer *n* knows where to look for the identifier of the protocol implementation at layer *n+1* and so on. IP, for example, knows that the identifier for the upper layer protocol, usually TCP or UDP, is the byte at offset 9 in the IPv4 header and chooses the function to pass the packet to next accordingly (*udp_input* or *tcp_input* in BSD Unix). Since with active plugins it might not be known to one function where to look for the function identifier of the next (e.g. in case of active plugins as IPv6 options, the next function identifier is totally opaque to the previous function and determined by the IPv6 option framework), we plan to use the FD as a central instance dispatching the corresponding parts of the packet's data to the suitable function implementations. The FD thereby calls all the functions referenced in the packet except for the case that one function decides to either drop the entire packet or forward it to the next node. The FD keeps track of all known function identifiers and their implementation's entry. In case of a previously unknown function identifier, the FD contacts the APL through the socket interface described above 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 in the queue of received packets. It maintains it's own queue of active packets with previously unknown function identifiers. On a call from the APL it resumes the processing of the enqueued packet by calling the newly installed active plugin.
- **Resource Controller** (**RC**): the RC keeps track of the resources consumed by active plugins of different types. First of all, it is responsible for fair priority time sharing which ensures CPU time balance between active functions and the rest of the networking subsystem. Greedy active plugins in terms of CPU cycles will be called less often then plugins which economically consume CPU cycles to achieve their desired result. We plan to design and implement a scheme to ensure fair CPU time balance. The second quantity worth observing is the active plugin's memory consumption. The plugins will be restricted to an upper limit of memory usage depending on the current average utilization of the networking subsystem. We plan to modify the kernels memory management to keep

track of memory usage on a per plugin basis and possibly deny additional memory to greedy plugins.

Both mechanisms for fair CPU time distribution and memory allocation will be configurable by the administrator. The clear goal for both mechanisms is to encourage developers to write effective active plugins.

An important resource management issue has to do with the ANN running out of resources such as CPU and/or memory capacity. We plan to explore both policies and mechanisms which can do (1) effective "admission control" to ensure sufficient resources for admitted active connections (2) 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.

All RCs in an ANN periodically communicate with each other on a reserved VC about their total load. In case of an uneven load distribution in the ANN, packets of selected flows will get diverted for active processing from heavily loaded ANPEs to lesser loaded ones. The load-sharing algorithm will thereby first try to distribute flows to ANPEs connected to the same switch port before diverting to other ANPEs on the switch, thus saving switch bandwidth.

We have already developed operating system support for ensuring QoS guarantees in the context of two other projects [18, 32] funded in part by DARPA's Quorum. We plan to use these operating system mechanisms and possibly expand them to implement the RC.

- Other components shown in Figure 5 are already implemented in the Router Plugin platform and briefly described here: the IPv4/IPv6 stack is a dual-stack IP implementation where the IPv6 part comes from the Institute National de Recherche en Informatique et en Automatique (INRIA, [19]) and is modified to fit into the Router Plugin system. The Association Identification Unit (AIU) implements fast packet filtering and flow detection mapping individual data packets to flows and matching flows against sets of filters. The Plugin Control Unit (PCU) glues dynamically loadable plugins to the rest of the networking subsystem and forwards messages from user-space components to the individual plugins. The Hierarchical Packet Scheduler provides a packet scheduling framework and consists of two dynamically loadable plugins for DRR and Hierarchical Fair Service Curve [HFSC, 36] packet scheduling.

Besides the software framework described, other software components have to be specified and developed in order to build a fully functional system. In the next subsection, we explain the software required to implement a code server. To efficiently administer active plugins and to provide the desired functionality, packaging of the individual active code components is required and this capability is explained in the following subsection as well. Finally, we conclude this section with the motivation for a new active protocol header specification.

4.1 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. For manual code server configuration, we plan to design and implement suitable end user tools. Developers are invited to provide code servers storing their own active plugins. Public root code servers would consult the developer's code servers for an initial set of active plugins.

Code servers are network nodes running at least the Plugin Database Controller, a subset of the Security Gateway (to authenticate active plugins), and a Policy Controller similar to the regular ANPE described above. Since most of today's router hardware lacks large mass storage, end systems similar to database servers are more likely to be configured as code servers. It is important to see that a code server does not have to feature a special, modified networking subsystem as the ANN does. More important is the code server's position relative to the ANNs requesting the active plugins. Code servers should be placed at the center of a network topology. We plan to design our software infrastructure for the code server to be as operating system independent as possible to ensure easy portability and we plan to implement them on at least two operating systems, BSD UNIX and Sun Solaris. We further plan to explore the usability of publicly available relational and object-oriented database technologies to efficiently store active plugins.

4.2 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 makes sense for strongly correlated active functions such as the four options for IPv6 mobility support [21]. On occurrence of a function identifier in a data packet, not only the referenced active function implementation is downloaded and installed but also one or multiple others, since their reference is very likely 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 for possibly different hard/software architectures
- the developers' digital signature
- the code servers' 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.

We plan to design and publish the exact package format in the context of this work.

4.2.1 An Active Protocol Header

Although we think that our architecture extends the current internet architecture in a complementary way by simply introducing dynamic upgrading of network nodes on occurrence of new function identifiers in data packets, a potential problem with our solution could be the size of the function identifier space available. In IP, protocol identifiers are only eight bits long. Roughly 10% of the numbers available are already allocated to protocols. In IPv6, the option type identifier is eight bits long out of which two bits are reserved, leaving 63 possible IPv6 options with four different given semantics. If active networking technologies start being deployed in the future, this might not be sufficient. We therefore plan to design and propose a new active network protocol header which includes the minimal number of fields possible to allow a larger space of function identifiers. We plan two versions of such a header depending on the context: the first version to be used as IPv6 options and the second as an independent layer three or four header. We might also consider a version of our header in the context of the Active Network Encapsulation Protocol (ANEP, [1]) if it turns out to be broadly accepted.

5 Applications

We intend to design and implement a variety of applications on top of our active network architecture. Two applications, discussed first in the following section, will be part of the base effort: automatic network protocol deployment/revision and active reliable multicast. Other applications include congestion control for real-time video and audio streams and mixing of sensor data [37].

5.1 Network Protocol Deployment/Revision

Design, implementation and large scale deployment of new network protocols has proven to be extremely hard with the current network architecture. If one follows the development and deployment efforts of IPv6 as successor of IPv4, one realizes how difficult it is in today's heavily commercial Internet. The design efforts for IPv6 started as early as 1994 [8] and still, even the most optimistic voices do not expect IPv6 to be used widely before 2010. Once a protocol is deployed, it is extremely hard (if not impossible) to change it globally. Therefore the design has to be very careful and every single feature must be discussed in a lot of detail which takes a tremendous amount of time. The discussion on how to use the IPv6 class (former: 'priority') field has been ongoing for months and still no consensus has been reached. Besides technical issues, political arguments more and more influence the design and deployment of such technologies since they became strategic for most companies. What is needed is a fully automated way to deploy and revise new network protocols globally. This would allow for incremental refinement of specifications and implementations based on real-world experience which has not been possible so far. One of the clear foci of the work is therefore on automatic protocol deployment and protocol revision. Incremental addition of new features to existing protocol implementations can easily be done with the our system as well. In IPv6, only a very small set of IP options is specified in the base specification [13]. 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 would be '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 [21] and possibly others. As mentioned before, the described system is not limited to such incremental upgrades of an existing protocol. We will show deployment of new, yet to be designed, experimental protocols in order to demonstrate site wide migration to a new protocol stack in a very smooth and centralized way, where only one server has to be configured. The process of installing a new site wide network protocol implementation would roughly go through the following steps. Assume that we want to install a new layer four protocol for real-time video called VTP, Video Transport Protocol:

- Configure one machine as a code server which initiates the upgrading process and has the active plugins available in it's local database (or a least a pointer to a code server somewhere on the Internet).
- Send a broadcast (or multicast to the "all nodes" group in IPv6) of a packet from the code server to the network nodes in the same subnet containing a function identifier "VTP init". If the subnet is large, the code server could also unicast the packet to one node after another to prevent server overload; that could also be necessary in case the systems have to be upgraded in a certain order, e.g. routers before hosts or to apply different configuration to different nodes.
- On reception of the packets, the clients ask the code server to send the corresponding active plugin since they encountered an unknown function identifier.
- The code server sends the active plugin requested to the clients; the active plugin package may

contain multiple function implementations.

- The clients load the active plugins and incorporate the various functions into the network subsystem and apply configuration information to the "VTP init" function which can be broadcasted from the code server as well.
- After waiting a random amount of time (again to prevent code server overload), the clients which are configured as routers forward the "VTP init" packet on their interfaces. This is done by code in the "VTP init" packet itself.

After going through the process for all the nodes in the network, the whole network has been upgraded to VTP. On reception of function identifiers for VTP functions in packets (regular VTP packets), the packets are passed to the new functions as if they had always been there. Assuming careful timing and a time span large enough (e.g. multiple hours to a few days) such a protocol upgrade could be done for even a very large network. We plan to demonstrate the deployment of IPv6 functionality into an IPv4 network through active plugins. We think that this kind of automatic network protocol migration could help a great deal to simplify protocol deployment in the future and is of great importance.

5.2 Large-Scale Reliable Multicast

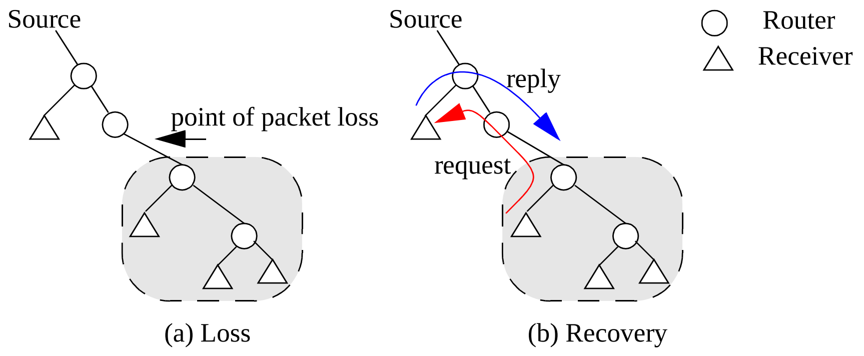

Figure 6: Recovery steps with topology knowledge

The two main problems with retransmission based error control solutions are request implosion and lack of local recovery. While existing schemes have good solutions to request implosion, they offer only approximate solutions for the local recovery problem. There are two trends that motivate a need for reliable multicast as an application of active networks: (1) there is a consensus in the community that reliable multicast solutions should be application specific; (2) most of the difficulty in existing reliable multicast end-to-end approaches stems from not having a good way to implement a hierarchy using only endpoint nodes. As a result, new research in "end-to-end" approaches is exploring the potential improvements to be gained by expanding the services provided by network routers. Washington University developed a new reliable multicast scheme [26] which introduces a small amount of "intelligence" into routers. In a multicast environment, loss of a packet in the network results in a failure to deliver a copy of the packet to all receivers located in the subtree rooted at the branch starting from the point of loss, as shown in Figure 6. A natural solution to recover the packet is the following: (1) the receiver directly below the loss sends a request to the receiver immediately above the loss; (2) the receiver immediately above the loss multicasts the lost packet to the affected branch. By using two new IP multicast options and an additional byte in IGMP reports, the scheme allows routers to provide a refined form of multicasting that enables local recovery. Besides providing good local recovery, the scheme has small recovery latencies (it requires no back-off delays), produces fewer duplicates than other schemes, and isolates group members from details of group topology. As with IPv6 options, the enhancements to the routers would have to be integrated on compile time of the networking subsystem. We plan to implement (and possibly improve) the scheme by usage of active plugins which allow upgrading of the routers at run time.

Another interesting scheme for reliable multicast using active network technologies has been proposed recently (ARM, [22]). Routers in the multicast tree play an active role in loss recovery in a sense that they use soft-state storage within the network to improve performance and scalability. In the upstream direction, routers suppress duplicate NACKs and in the downstream direction, they limit the delivery of repair packets to receivers experiencing loss. To reduce wide-area recovery latency and to distribute the retransmission load, routers cache a certain amount of multicast data. A prototype has been implemented on MIT's ANTS [39] architecture and yields promising results [22]. We plan to implement or port a version of the algorithm for comparison to our system.

Besides these two schemes, we plan to design and implement a set of fully distributed, application specific active reliable multicast protocols which takes full advantage of the network's topology since we believe that optimal performance for very large-scale, reliable multicast is possible therewith.

5.3 Other Applications

We plan to design, implement and experiment with active networking support for real-time audio and video streams and mixing of sensor data (Option 2). As other applications emerge over time however, we might consider their applicability as well if our resources permit. Possible other applications of interest are self-organizing network caches [7], e.g. for web traffic and on-line auctions [22].

5.3.1 Real-Time Video and Audio

- **Congestion control for real-time video:** Congestion control for real time video streams has been shown to be a promising field for active network technologies [6]. An MPEG stream consists of frames of three different types: I, P, and B. Coding dependencies exist between the frames, causing P and B-frames to possibly require I frames in order to be properly decoded. By selectively dropping P or B-frames prior to I-frames in case of congestion, much better results in terms of video quality can be achieved than by using the regular, random drop mechanism implemented in today's routers. The ETH has developed an alternative video codec called WaveVideo [10] based on the wavelet transformation. Although this algorithm achieves much better results when exposed to packet loss then most MPEG codecs (it performs particularly well in mobile computing environments where packet loss occurs very frequently), it can benefit in similar ways from selective packet dropping under congestion. Since we envision a multi-gigabit environment with our architecture, the bandwidth consumed by a typical video-stream might appear insignificant. Nevertheless, we believe that congestion control is an important application of active networks in low-bandwidth network environments (e.g. mobile computing) and plan to design a selective packet drop algorithm for WaveVideo as an active plugin.
- **Congestion control for real-time audio**: Real-time audio is even more sensitive to packet loss than video. Most audio codecs use the Real-Time Transport protocol (RTP) [32] for transmission over a network. There is a large variety of different audio encoding standards [33]. They are either sample-based (each audio signal is represented by a fixed number of bits) or frame-based (a fixed length block of audio is encoded into another block of compressed data, typically also fixed length). Depending on the particular coding algorithm used, selective packet dropping could achieve better results under network congestion. For example, dropping packets belonging to less "important" channels (e.g. the 'surround' channel) and dropping high-rate samples prior to low-rate samples, would each produce better quality audio than random dropping.
- **Media gateways**: We plan to investigate the usage of active network technologies to efficiently implement media gateways [3, 4]. The main purpose of a media gateway is to mitigate bandwidth heterogeneity among receivers of a real-time multicast audio/video session.

5.4 Mixing of Sensor Data

In a situation in which geographically dispersed sensors and receivers are connected over a data network, the sensors are continuously collecting large amounts of information that must be combined for one or more receivers. Instead of passively forwarding each packet of the input stream to each receiver, an active node can use signal fusion to do some of the signal data mixing within the network, as pointed out in [22]. If multiple input signals pass through the same node at approximately the same time, that node can mix the signals. If the mixed signal is smaller than the sum of its constituents, this will reduce the total network traffic. If multicasting is used to transmit the signals from the sender to the receivers, further optimizations are possible using active networking. Think of a situation with three senders S1, S2, S3 and four receivers R1, R2, R3, R4 (Figure 7). S1 multicasts to all of the receivers. S2 multicasts to R1 and R2, S3 to R3 and R4. All multicast signals pass through the same ANN on their way to the receivers. It might be useful to first split the signal from S1 into two multicast messages sent to the multicast groups including (R1, R2) and (R3, R4) and mix these two signals with the signals from S2 and S3 going to the same groups (R1, R2) and (R3, R4). Mixing of data in the network can significantly reduce overall bandwidth consumption. More generally, fission and fusion of sensor data in a network with multiple sensors and multiple users consuming the data allows for customized data delivery to each of the users according to their individual preferences in the most effective way.

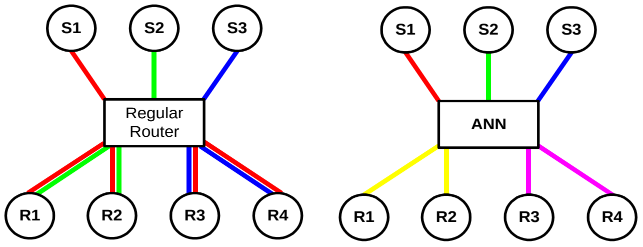

Figure 7: Sensor data mixing

6 Related Work

6.1 MIT

Tennenhouse et al. [36] proposed "capsules", that is datagrams carrying small fragments of code, and an implementation in the form of an IP option [38]. The TCL language and a stripped-down TCL interpreter is used to provide safe execution of the code. Some of the simpler 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. Due to the use of interpretation and virtual machines to overcome security issues, the approach potentially suffers from performance problems.
Recently, this group proposed the ANTS [39] toolkit (downloadable code available). It seems that ANTS' main goal 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 single packet, packets carry pointers to code which is loaded initially from the previous hop along a packet's path. Java is used as a programming language for active code. This approach optimizes the bandwidth usage with the drawback of a considerable initial delay. Our scheme potentially suffers from a similar delay problem but can address it by using a "probe" packet sent on connection setup from the server to the client which triggers the downloading of code modules in parallel on all routers along the packet's path. Although exploited in a different way, this work shows that code caching is an interesting optimization that is worth pursuing.
In another document, the usefulness of active reliable multicast is shown [23] using the ANTS platform (we briefly discussed this approach to multicasting in section 5.2) and results in terms of significant performance gains are measured in [22]. [22] also proposes various applications of active network technologies like sensor data mixing and inspired us to follow similar ways.

6.2 BBN

Smart Packets [27] is another interesting capsule related 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 with the goal to produce code which can be interpreted fast and which is compact enough to fit into an Ethernet packet. The programs are authenticated before interpretation and run-time limited during execution. The system provides a data dictionary which specifies what information a packet's program is allowed to access.

6.3 Georgia Tech

Zegura et al. [6, 7, 41] introduced the 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 work again provides a proof-of-concept, this time for the usefulness of application specific data processing in network nodes.
Recently, this group showed how self-organizing network caches can be built using active network technologies [7]. They used simulation as well as an analytical model to evaluate the performance gains offered by the caching and showed that it might be an application of active networking worth pursuing. The system proposed here builds in part on the theoretical background and terminology introduced in [6], but extends the system's capability to downloading function modules in-band, on-the-fly, and elaborates on the infrastructure needed to do so.

6.4 University of Pennsylvania

The SwitchWare [34, 17] project mainly uses the "programmable switch" approach with three important components: active packets, switchlets, and a secure active router infrastructure. Active packets are similar to MIT's capsules and switchlets are programs that provide specific services on the network nodes and can be dynamically loaded. The secure active router infrastructure ensures security for the other layers. Active packets are programmed in a simple programming language called PLAN (Programming Language for Active Networks). PLAN programs are strongly typed and statically type-checked to provide safety before being injected into the network. Further, PLAN programs are made secure by very greatly 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. Like in our approach, the code fragments are authenticated by the developer but explicitly (and 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) upon which system integrity is dependent 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 by 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 which makes the system as a whole less flexible. Active Bridging [2] is an application of SwitchWare which shows reprogramming of a bridge with switchlets.

6.5 Columbia University

Netscript [40] is middleware for programming functions of intermediate network nodes. Since it is middleware, it is not as comparable with our work as other work described here, but we still consider it very interesting. The Netscript programming language allows script processing of packet streams with 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 admit 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. Netscript code is packaged as mobile agents that can be dispatched to network nodes and executed there dynamically at runtime. 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.

7 Conclusions

We elaborated on three key factors critical to pave the way for active networking in a gigabit environment. First, we described a new active networking architecture called “Distributed Code Caching” which optimizes the capsule approach. We replace the capsules' program code by a reference to an active plugin stored on a code server, thus building a distributed code cache. Second, we elaborated on a gigabit hardware platform which allows high-performance active networking in a scalable fashion combining off-the-shelf and customized hardware components. Third, we described a software infrastructure which optimally supports the hardware components and provides the functionality necessary for highly efficient active networking by employing load-balancing. Finally, we proposed a two state-of-the-art applications for our platform intended to show the capabilities of the system and active networking in general.

The project to build the system described in this paper is funded by DARPA, and we expect to start prototyping the hardware and software components during the summer of 1998.

8 References

[1] Alexander, D., et al., “Active Network Encapsulation Protocol (ANEP)”, *RFC DRAFT*, July 1997

[2] Alexander, D., Shaw, M., Nettles, S., Smith, J., “Active Bridging”, In *Proceedings of SIGCOMM ‘97*, September 1997

[3] Amir, E., McCanne, S., “An Application Level Video Gateway”, In *Proceedings of ACM Multimedia*, November 1995

[4] Amir, E., McCanne, S., Katz, R., “Receiver-driven Bandwidth Adaptation for Light-weight Sessions”, In *Proceedings of ACM Multimedia*, November 1997

[5] Atkinson, R., “Security Architecture for the Internet Protocol”, *RFC 1825*, August 1995.

[6] Bhattacharjee, S., et al., “An Architecture for Active Networking”, In *Proceedings of INFOCOM ‘97*, April 1997

[7] Bhattacharjee, S., Calvert K., Zegura, E., “Self-Organizing Wide-Area Network Caches”, In *Proceedings of INFOCOM ‘98*, April 1998

[8] Bradner, S., Mankin, A., “The Recommendation for the IP Next Generation Protocol”, *RFC 1752*, January 1995

[9] Chaney, T., et al., “Design of a Gigabit ATM Switch”, In *Proceedings of INFOCOM’97*, April 1997.

[10] Dasen, M., Fankhauser, G., Plattner, B., “An Error Tolerant, Scalable Video Stream Encoding an Compression for Mobile Computing”, *ACTS Mobile Summit 96*, November 1996

[11] Decasper, D., Dittia, Z., Parulkar, G., Plattner, B., “Router Plugins - A Software Architecture for Next Generation Routers”, To appear in *Proceedings of SIGCOMM’98*, September 1998

[12] Decasper, D., Plattner, B., “DAN: Distributed Code Caching for Active Networks”, In *Proceedings of INFOCOM’98*, April 1998

[13] Deering, S., Hinden, R., “Internet Protocol, Version 6 (IPv6), Specification”, *RFC 1883*, December 1995

[14] Dittia, Zubin, Jerome R. Cox, Jr. and Guru Parulkar. “Catching Up With the Networks: Host I/O at Gigabit Rates”, *Technical Report WUCS-94-11, Department of Computer Science, Washington University in St. Louis*, 1994.

[15] Dittia, Zubin, Jerome R. Cox, Jr., and Guru Parulkar. “Design of the APIC: A High Performance ATM Host-Network Interface Chip,” In *Proceedings of INFOCOM’95*, April 95.

[16] Eastlake, D.E., “Domain Name System Security Extensions”, *draft-ietf-dnssec-secext2-02.txt*, November 1997

[17] Gunter, C., Nettles, S., Smith, J., “The SwitchWare Active Network Architecture”, *http://www.cis.upenn.edu/~switchware/papers/switchware.ps*, November 1997

[18] Gopalakrishnan, R., Parulkar, G., "Bringing Real-time Scheduling Theory and Practice Closer for Multimedia Computing",In *Proceedings of the 1996 ACM SIGMETRICS*", May 1996

[19] INRIA IPv6 implementation for NetBSD/FreeBSD, *ftp://ftp.inria.fr/networking/ipv6*

[20] Jacobson, V., and McCanne, S., “Visual Audio Tool”, *Lawrence Berkeley Laboratory, Software online*†

[21] Johnson, D., Perkins, C., “Mobility Support in IPv6”, *draft-ietf-mobileip-ipv6-03.txt*, July 1997

[22] Legedza, U., Wetherall D., Guttag, J., “Improving the Performance of Distributed Applications Using Active Networks”, *In Proceedings of INFOCOM’98*, April 1998

[23] Lehman, L., Garland, S.J., and Tennenhouse, D., “Active Reliable Multicast”, In *Proceedings of INFOCOM’98*, April 1998

[24] McCanne, S., Jacobson, V., “vic: A Flexible Framework for Packet Video”, In *Proceedings of ACM Multimedia*, November 1995

[25] Mockapetris, P., “Domain Names - Implementation and Specification”, *RFC 1035*, November 1987

† ftp://ftp.ee.lbl.gov/conferencing/vat

[26] Papadopoulos, C., Parulkar, G., and Varghese, G., “An Error Control Scheme for Large-Scale Multicast Applications”, In *Proceedings of INFOCOM’98*, April 1998

[27] Partridge, C., Jackson, A., “Smart Packets”, *Technical report*, BBN, 1996

[28] Parulkar, G.M., Schmidt, D.C., Turner, J.S., “a^It^Pm : A Strategy for Integrating IP with ATM,” In *Proceedings of SIGCOMM’95*, August 1995.

[29] Rivest, R.L., “The MD5 message Digest Algorithm”, *RFC 1321*, April 1982

[30] Rivest, R.L., Shamir, A., and Adleman, L.M., “On Digital Signatures and Public Key Cryptosystems”, *Technical Report*, *MIT/LCS/TR-212*, January 1979

[31] RSA Library*, ftp://ftp.rsa.com/rsaref*

[32] Schmidt, D., et al., "A High-Performance Architecture for Real-time CORBA", IEEE Communication Magazine, Vol. 14, No. 2, 1997

[33] Schulzrinne, H., “RTP Profile for Audio and Video Conferences with Minimal Control”, *draft-ietf-avt-profile-new-01.ps*, November 1997

[34] Smith, J., Farber, D., et al., “SwitchWare: Accelerating Network Evolution”, *White Paper*, June 1996

[35] Stevens, W., “TCP/IP Illustrated Volume 1 - The protocols”*, Addison-Wesley Professional Computing Series*, 1994

[36] Stoica, I., Zhang, H., and Ng, T.S.E., “A Hiearchical Fair Service Curve Algorithm for Link-Sharing, Reail-Time and Priority Services”, In *Proceedings of SIGCOMM’97*, September 1997.

[37] Tennenhouse, D., et al. “A Survey of Active Network Research”, *IEEE Communications*, January 1997

[38] Wetherall, D., Tennenhouse, D., “The ACTIVE IP Options”, In *Proceedings of the 7th ACM SIGOPS European Workshop*, September 1996

[39] Wetherall, D., et al, “ANTS: A Toolkit for Building and Dynamically Deploying Network Protocols”, submitted to *IEEE OPENARCH’98*

[40] Yemini, Y., daSilva, S., “Towards programmable networks”, In *IFIP/IEEE International Workshop on Distributed Systems: Operation and Management*, October 1996

[41] Zegura, E., “CANEs: Composable Active Network Elements”,
Georgia Institute of Technology,
h*ttp://www.cc.gatech.edu/projects/canes/*