

# DAN: Distributed Code Caching for Active Networks

Dan Decasper<sup>1</sup>, Bernhard Plattner<sup>2</sup>

dan@arl.wustl.edu, plattner@tik.ee.ethz.ch

<sup>1</sup>Applied Research Laboratory, Washington University, St. Louis, MO, USA

<sup>2</sup>Computer Engineering and Network Laboratory, ETH Zurich, Switzerland

## ABSTRACT

*Active networking allows the network infrastructure to be programmable. Recent research focuses on two commonly separated approaches: “capsules” and “programmable switches”. Capsules are typically small programs in packets which flow through the network and are executed in-band on nodes receiving them. Programmable switches are network devices which offer a back door to inject code by a network administrator out-of-band in order to enhance the device’s capabilities. By combining these two approaches, this paper proposes a novel system architecture which allows both application specific data processing in network nodes as well as rapid deployment of new network protocol implementations. Instead of carrying code, data packets carry pointers to digitally signed active modules initially loaded on-the-fly, in-band, from trusted code servers on the network. Packet processing runs at high speed, may access and modify the whole network subsystem and no potentially slow virtual machines are needed.*

**Key words:** active networks; application specific data processing; automatic protocol deployment; distributed code caching

## 1. INTRODUCTION

Active networks [24] 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.

Increased flexibility and customizability implies security problems. Stability is crucial to network devices not only because the network has become so important to people’s daily work, but also because the devices lie in separate administrative domains and are run by different people. Breakdowns are extremely hard to trace and fix.

In this paper, we present an active network architecture which addresses security through policy and simple cryptology instead of potentially slow, restricting and complicated technical measures like virtual machines. We plan to implement our solution on a platform specially designed for active networking through Distributed Code Caching (DAN). We will show examples for both

safe, new network protocol deployment as well as application specific data processing in network nodes using our system.

The next section, section 2, explains the main motivations and problems of active networks and puts the security issue into a different light. Section 3 describes our system and addresses questions and concerns related to it. Section 4 outlines two applications we plan to implement as proof-of-concept for our system. Section 5 provides some insight into the actual implementation environment. Section 6 relates our work to that of others, and Section 7 summarizes our ideas.

**WARNING:** this paper is a design paper. It elaborates on a idea which has not been implemented nor verified yet. We do not provide final results. Several components of the system described may be subject to chance!

## 2. ACTIVE NETWORKS: MOTIVATION AND PROBLEMS

Active networking is an exciting area of research which concentrates on two commonly separated approaches: “programmable switches” [2, 6, 22] and “capsules” [18, 25]. 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 either focuses on how to upgrade network devices at run time or on upgrades which support end system applications (e.g. congestion control for real-time data streams) or on a combination of both.

Capsules, though, are miniature programs that are transmitted in-band and executed at each node along the capsule’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. It looks like this approach has an enormous potential impact for the future of networking. In the near future, capsule-based solutions potentially suffer from performance related problems mainly due to security constraints. They commonly make use of a virtual machine that interprets the capsule’s code to safely execute on a node. This is similar to the way Java applets run in web browsers. The virtual machines must restrict the address space a particular capsules might access to ensure security, which restricts the application of capsules.

One of the main motivations of active networks is to reduce the difficulty of integrating new technologies and standards into a

shared network infrastructure. If one follows the development and deployment efforts in the context of the IPv6 [11] network protocol (successor of IPv4), one realizes how extremely difficult such a project is in today's heavily used Internet. The design efforts for IPv6 started as early as 1994 [7] and still even the most optimistic voices do not expect IPv6 to be used widely before 2005-2010. Since it is so extremely hard (if not impossible) to change IPv6 globally with the common network architecture once it starts getting used, every single feature must be discussed very carefully and in a lot of detail. The discussion on how to use the IPv6 class (former: 'priority') field has been ongoing for months and still no full consensus is 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 allows for incremental refinement of specifications and implementations based on real-world experience which has not been possible so far. The system proposed in this paper shows a possible architecture to achieve that.

Even if protocols like IPv6 and TCP/UDP over IPv6 are spread and used worldwide, they will be the "best" network protocols for a subset of applications only. The emerging need for fast multimedia applications shows that it is often very desirable to run application specific data processing in network nodes. [5] shows that active networks technologies successfully help to handle congestion which occurs when transmitting MPEG video streams over loaded routers by application specific, selective packet dropping. Other examples show how to use the technology for reliable multicast [16, 17], mixing of sensor data [15], web-caching [7], and many more. This is another important application of active networks technologies, which we focus on.

To overcome the performance related problems of capsules, we think that a combination of both the capsule and the programmable switch approach is very appealing. We address security through policy and cryptology. If the node executing an active code fragment has the possibility to reliably check the source of a fragment and the identity of its developer, it is able to decide based on that information whether or not it wants to accept and execute the fragment. This is very similar to the way software is distributed to end systems. Nobody would want to run a word processor in a virtual machine just because he/she can't trust the word processor's developer. By knowing the origin of an application, we implicitly assume that it does not corrupt our system. The code fragments used in our system are digitally signed by the developer and come from trusted code servers. This 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.

### 3. THE DAN ARCHITECTURE

In common networks, network nodes are connected over (possibly) heterogeneous link layer media to each other. They are able to talk together because they use a common network layer protocol (typically IP) which hides both the data link layer and the transport and higher layer details. Every node which is forwarding a

packet is called a "router" and does not have to know about protocols on layers higher than the network layer. A packet usually flows from one end system called a "host" to one or multiple other hosts by jumping from router to router until it reaches the final destination. Most of the communication in the internet is client-server oriented: one node called the "client" sends a request for some sort of "passive"<sup>1</sup> data to another node called "server". The server may or may not, depending on policies established on the server, provide the client with the data. In this paper we call such a server "data server" in order to distinguish it from "code servers" introduced later.

Network packets consist of a finite sequence of identifiers for functions and input parameters for these functions. 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 the packet is received on) and the last function or set of functions is implemented in the application consuming the packet. An Ethernet packet, for example, contains a unique identifier for the upper layering 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. 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 transport layer data or detection of errors). Depending on the type of node the packet is processed on and the packet's content, only a subset of the functions may be called.

This scheme is common to all of the well known types of networks. We introduce a minimal amount of formalism to describe this scheme inspired by [6] which uses similar terminology. A packet can be interpreted as a sequence of function identifier  $f_1 \dots f_N$  with usually distinct sets of parameters  $P_1 \dots P_N$  (Figure 1). The function identifiers and parameters are strictly order in the



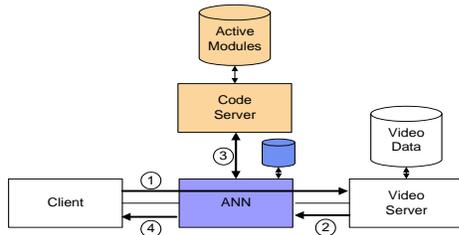
Figure 1: Datagram (schematically)

packet with  $f_n$  followed by  $P_{n-1}$  with the exception of the final parameter  $P_N$  ( $f_2 P_1 \dots P_N$ , ignoring the for the discussion irrelevant detail that the function identifier must not necessarily always be the first byte or word of the  $f_n P_{n-1}$  pair). The first function is not indicated by any function identifier but derived from the context in which packet processing starts (e.g. the packet is received by an Ethernet card, therefore the Ethernet input function is called). Every function identifier  $f_n$  unambiguously points to an implementation of the function  $f_x$  ( $f_x | x=f_n$ ). If the function identified by  $f_{n-1}$  decides to call the function identified by  $f_n$  and  $f_n$ <sup>2</sup> is not known to  $f_{n-1}$ ,  $f_{n-1}$  proceeds into some form of error handling by usually dropping the packet and possibly notifying the node originating the packet.

<sup>1</sup>"Passive" meaning here that the content of the data is irrelevant to the network subsystem of the client, the data server and the intermediate routers

<sup>2</sup>we say " $f_{n-1}$ " and mean "the implementation of the function identified by  $f_{n-1}$ "

Now, we simply add a new alternative for the node's behavior. If the node is unable to locate the function identified by  $f_n$ , it temporarily suspends processing of the packet and calls a "code server" for the implementation of the function  $f_x$ . In contrast to data servers which provide a client with "passive" data, a code server is a well known node in the network which provides a library of (possibly) unrelated functions for (possibly) different types of operating systems from various developers. We call a router or end-system modified to download active modules an Active Network Node (ANN). Figure 2 shows an example of a client downloading



**Figure 2:** ANN fetching active module from code server

real-time video through an 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 real-time video transport; (3) the ANN does not have the code referenced in it's local cache and therefore contacts a code server for the module; (4) the ANN receives the active module, dynamically links it in its networking subsystem, applies the data to the transport function and forwards the packet to the client. Once the module is downloaded, it is permanently stored locally on the ANN preventing other downloads for the same active module in the future. One can think of the code server as a "cache" for the active module code. By putting code servers into a hierarchy (see section 3.1.2), several levels of caching can be established.

The cycle can be described in pseudo-code as follows:

```

execute the first function depending on hardware properties (interface the packet is
received on)
n=2
do {
  x=fn
  if(fx not locally available) {
    enqueue packet
    send request for active module to code server
    resume packet processing with a packet in input queue
  } else {
    execute fx(Pn)
  }
  inc n
}
on reception of active module implementing fx {
  dequeue packet
  goto 11
}

```

Eventually one of the functions  $f_n$  will jump out of the 'do' loop and terminate processing of the packet either by dropping the packet, forwarding the packet to another node or passing it to an application.

It is important to note that the active modules 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, they are executed as fast as any other code.

Another important point in this context is security: by actually loading active modules from well known code servers which authenticate them and give the node the possibility to check the module's sources, and by providing digitally signed modules from well known developers only, the whole security problem advanced active networks solutions usually suffer from, degenerates to the installation of a simple rule on the node which let it choose the right code server (see section 3.1.1 for explanation on how exactly the node chooses the code server) and a database of public keys to check the developers signature. The node uses RSA public key encryption [21] to check signatures and authentication information. It automatically downloads the public keys required through the Domain Name System (DNS) featuring its recently published security extensions [13]. Only one public key (usually the networking subsystem developer's key) has to be installed initially on the node. Alternatively, if a node relies only on code server authentication, it can use IP security [5] which implementation is mandatory in IPv6 and does not require any additional overhead for security.

The combined facts that the active modules may have access to the whole network subsystem and that they run at high speed, we believe that our solution offers a powerful environment for active networks. While our approach is safe and fast, it may appear less flexible than the typical capsule approach. This is true to a certain degree. In order to properly authenticate code modules, some administrative overhead must be introduced. An independent authority must take care of distributing function identifiers (as it is already the case today), developer codes, and network subsystem code in similar ways IP addresses or MAC addresses are distributed to developers and sites. We believe that active modules will be implemented mainly by companies developing network subsystems which are not very large in number. Thus, we don't expect this kind of limitation to be very important in a real-world scenario.

### 3.1 DAN properties

There are (other) problems introduced with on-demand, just-in-time fetching of active modules described above which have to be addressed. This section discusses the most important issues. We elaborate on where the active modules come from and how a node may find a code server. We suggest putting code servers in a hierarchy for best possible distribution of active modules in the large scale. Policies installed locally on nodes regulate storage and acceptance issues. Finally, interaction of the most common network protocols (TCP/UDP/IP) with our system and the problem of connection setup delay due to code downloading is discussed.

### 3.1.1 Binding to a code server

We propose four different ways for a node to find a code server.

First, the node may be configured with a list of unicast code server addresses. The node chooses the first address and tries to fetch the module. In case it gets no answer after some pre-empted time, it tries the next address and so on until it either receives the module or the code fetch fails. In case of failure, it may send a message back to the source. This setup gives the node the possibility to check both the developer's signature in the module as well as the code servers authenticity. The responsibility for finding a suitable code server (in terms of minimal delay and maximal bandwidth) is up to the administrator selecting the unicast address.

Second, code servers could come with an anycast [19] address. All the client has to know is the one and only anycast address for a code server. All code servers listen to the same anycast address and finding the closest one would be up to routing. When the node receives the module, it may check whether the module came from a trusted source by looking at the modules authentication information. This method has the advantage of being the fastest possible way for a node to get the module since it automatically connects to the closest server. Anycast fetch can be combined with the first method in a sense that the node tries anycast first and falls back to unicast in case of failure. Unfortunately, anycast support is currently not deployed widely and therefore this method is probably not of great interest in the near future. Another problem is that the node requires a list of potential code servers in order to check whether it came from a trusted source. Alternatively, it could just rely on the developers digital signature. The module received can potentially be useless if the code server sending the module is not on the node's list.

Third, a node can send a multicast message to the "all code server" group asking for the module. All code servers receiving the message reply with their version of the module. The client may either pick the first module it receives or the most recent version of the function it received after some time. The node can set the Time-To-Live/Hop Count (TTL) field in the request packet header to a small value to prevent the request from going "too far". Again, combination with the other two methods are possible. However, as in the anycast case, if the node receiving the module has to check it's authenticity, it needs a list of potential sources.

Fourth, the node could try to fetch the active module from the data server itself. 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, dedicated, code module distribution infrastructure, as suggested in the next section. 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. The disadvantage of this solution is that it allows only one level of authentication (the developer's digital signature) and that code modules may come from "non optimal" sources in respect to bandwidth and delay.

### 3.1.2 Establishing a code server topology

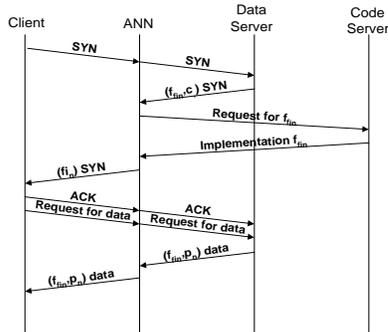
The goal of setting up a topology of code servers in the large scale is to simplify code distribution and to ensure minimal download time for code modules. The setup of an optimal, global scale topology is a complex issue and certainly requires further investigation. To start, it may be good enough for an organization to set up one code server locally in its intranet or to fetch active modules from the code server as discussed. New active modules from developers can be downloaded and installed manually on an intranet code server as required to provide the local intranet with enhanced functionality. This is already a big step towards automatic protocol deployment. We will suggest possible large scale arrangement of code servers later, based on experience we gain from implementing and testing the system described in this paper.

### 3.1.3 Policy issues

Policies regarding acceptance of active modules on nodes may be desired. Even if module sources and the modules themselves are authenticated, network administrators may wish to restrict the set of developers they accept active modules from or exclude certain specific active modules because of undesired behavior. Our system allows for these kinds of restrictions to be configured on nodes. Specific function identifiers can be excluded. In case an active module corresponding to an excluded function identifier found in a packet received is detected, an error message is sent back to the source of the packet.

### 3.1.4 Integration into common network protocols

On-demand downloading of the active modules potentially introduces a serious delay problem. By applying a small set of precautions, the delay problem can be significantly reduced. Basically, as stated earlier, a active module is fetched and loaded as soon as a function identifier for an unknown function is detected in a packet header. For all connection-less protocols like UDP, this means that active modules will simply be fetched on reception of the first packet of a flow on all nodes which do not have the active module already. Data sources which make use of DAN may send out packets containing the function identifiers in question, which are otherwise empty to trigger module loading along the packets path first before actually sending payload data. For connection oriented protocols like TCP, the best moment to load the active modules is on connection setup (Figure 3). When the data server replies with a SYN back to the client requesting the connection, it may include a packet containing the function identifier and optional configuration information and force the nodes along the path to fetch the active modules. Such a packet would be forwarded immediately to the next hop before downloading of the active module and lead to a quasi parallel download of the active module on all routers along the packets path. In BSD 4.4 [23], 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. For active modules which execute only on routers, on-demand loading of the corresponding code should be possible without the need of changing the end node's TCP if the code server in question provides reasonable response time. Note that Figure 3 shows the



**Figure 3:** Code fetch during TCP connection setup

worst possible case regarding delay: the active module has to be loaded from the code server. A local active module database on every node (the small cylinders attached to the ANN in Figure 2) prevent nodes from multiple downloading of the same active module. Once the node fetches the active module over the network, it can easily put a copy into its local (if possible non-volatile) memory. The next time the node boots, it may check its module database and load all necessary modules into the network subsystem. Subsequently received packets containing identifiers pointing to originally (at the network subsystem's build time) unknown but previously referenced functions can be processed at regular speed without the need of going through the packet queuing and active module fetching process again.

### 3.1.5 An active protocol header

Although we think that DAN 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 of our solution could be the size of 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 also eight bits but two bits are reserved, leaving 63 possible IPv6 options with four different 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 it will be used: the first version to be used as IPv6 options and the second as 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.

## 4. SAMPLE APPLICATIONS

We plan to use the system described to implement most of the applications which have been identified recently for active networks like congestion control for real-time video [5] and audio streams, web-caching [7], on-line auctions, mixing of sensor data [15], and high-performance media gateways [3, 4]. In this section,

we briefly look into automatic network protocol upgrading/revision and large-scale reliable multicast since we consider them to be the most important in the context of our work.

### 4.1 Protocol upgrading/revision

New function identifiers may be introduced at different layers. Using ATM as a link layer medium, on application of function identifiers could be to use them in the LLC SNAP field. For functions to be executed on end-systems only, they can occur in addition or instead of the usual transport layer function identifier for TCP/UDP. The most common way to introduce new function identifiers may be as IP options, which are defined for both IPv4 and IPv6 ([5] and [25] also describe a way to use IP option fields for active networks). Options are used to specify unusual datagram processing, e.g. source routing. Whereas option usage in IPv4 is very limited because of a maximal size of 40 bytes, IPv6 introduces a very flexible option concept. Only a very small set of IP options is specified in the base specification [11]. These options are mainly used to pad data packets to certain sizes in order to fit them into word boundaries. However, the protocol supports new options in a modular way. An arbitrary number of IPv6 option can follow the IPv6 header in the form of Hop-by-Hop (processed on every node along a packet's path) or Destination options (processed only on the end node). It is expected that these new options would be 'hardwired' into an IPv6 implementation. To support new options, such an implementation would require recompiling. With the system proposed here, new code modules for IPv6 options can be downloaded on demand from a code server the first time it gets referenced and stored for later use in the local cache. Using IPv6 options has the further advantage that the option type semantics allow for specifying the node's behavior in cases where it does not recognize the option type (skipping over option/discarding packet/sending ICMP message to source). We will show this for the four options required for IPv6 mobility support [14] and others.

Besides such incremental upgrades, we'll show centralized deployment of entirely new layer three and four protocols as well. As a proof-of-concept, we'll set up a code server in an IPv4 Intranet to broadcast special IPv6 packets thereby forcing the all nodes in a subnet to download active IPv6 modules through which they upgrade to IPv6. Provided that careful timing is applied, we believe that even very large Intranets can be upgraded in a centralized, near-automatic way without the need of shutdown/reboot of any of the devices.

### 4.2 Large-scale Reliable Multicast

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 [17] which introduces a small amount of “intelligence” into routers. In a multicast environment, loss of a packet in the network results in failure to deliver a copy of the packet to all receivers located in the subtree rooted at the branch sprigging from the point of loss. 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 multicast 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 modules.

Another interesting scheme for reliable multicast using active network technologies has been proposed recently (ARM, [16]). 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 [26] architecture and yields promising results [15]. We plan to implement or port a version of the algorithm for comparison to the proposed system.

Besides these two schemes and based on the experience gained from development, 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. IMPLEMENTATION ENVIRONMENT

We plan to build a high-performance hardware and software environment optimized for the purpose of DAN. We believe that an active networking node designed for high performance requires: (1) tight coupling between a processing engine and the network as well as between the processing engine and a switch backplane; (2) scalable processing power to meet the demands of active processing of packets. Over the past few years, we have been prototyping technological components that enable building of an Active Networking Node (ANN) that meets both the requirements extremely well in a cost effective fashion.

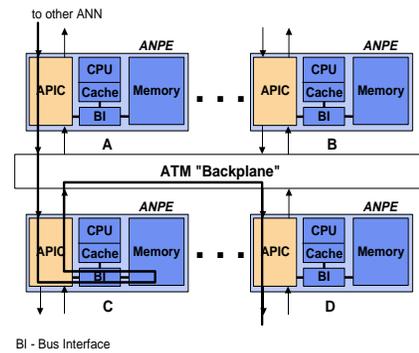


Figure 4: Active Network Node (ANN)

### 5.1 ANN Hardware

The hardware architecture for the proposed Active Network Node (ANN) is shown in Figure 4. It is derived from our high performance IP routing architecture [20] and refined and optimized for the purpose of active networks. The node consists of a set of Active Network Processing Elements (ANPE, four in Figure 4) connected to an ATM switch fabric [9]. The switch fabric 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 central to high speed active networks. Each ANPE card has a main data path that passes through our ATM Port Interconnect Controller (APIC, [12]) 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 letting it switch in hardware between VCs and pacing of ATM cells. The ANPEs are connected to the backplane via the APIC. Basically, ANNs are interconnected only through ANPEs. Other devices like workstations and servers are connected through a line card 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) an arbitrary 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 take away active flows from heavily loaded ANPEs to lesser loaded ones. Figure 4 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 daisy-chained through the APIC on a single switch port. The load-sharing algorithm can then first distribute the load to the other ANPE on the same port before diverting to other ANPEs on the switch 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 thereby removing the load from the active networking software subsystem which ensures optimized performance for the remaining active flows.

## 5.2 Software Infrastructure

The software framework running on each ANPE will be embedded into our Crossbow [10] research platform. The Crossbow project is a joint project between ETH Zurich and Washington University which 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 toolkit 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, Crossbow allows packet processing at high speeds easily saturating an OC-3 link running on a P6/200. We confidently expect to saturate an OC-12 link when running the system on a PII/300. The experience gained from the development of a modular high-performance toolkit will be critical for the development of the active network platform proposed here, which is inherently modular. Several integral components of Crossbow like the Association Identification Unit for fast flow detection and packet filtering will be used for the proposed project and reduce development time significantly. The proposed framework is shown in Figure 5 and introduces the following (main) components: an **Active Module Loader** which loads the active modules authenticated and digitally signed by their developers from well known code servers using a lightweight network protocol (e.g. UDP/IP); a **Policy Controller** which maintains a table of policy rules set up an administrator to restrict the set of supported modules; a **Security Gateway** which allows/denies active modules based their origin and developer by analyzing their digital signatures/authentication information; a **Module Database Controller** which efficiently administers the local database of active modules; a **Function Dispatcher** which identifies references to active modules in data packets and passes these packets to their corresponding function implementations; (5) a **Resource Controller** for fair CPU time sharing among active functions; it negotiates with the load-sharing algorithm for dynamic distribution of active flows to different CPUs.

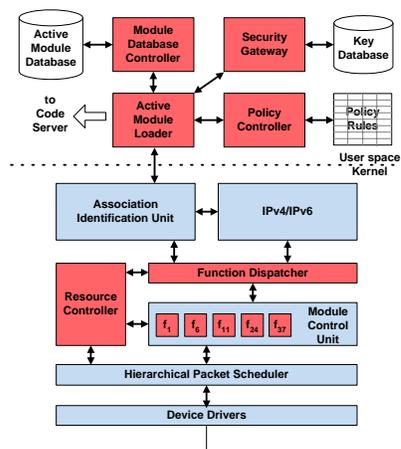


Figure 5: ANPE software architecture

## 5.3 Code server

Code servers are network nodes running at least the Module Database Controller, a subset of the Security Gateway (to authenticate active modules), and a Policy Controller similar to the regular ANPE described above. Since most of today's router hardware lack necessary 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. We plan to design the parts of our software infrastructure required to provide code server functionality as operating system independent as possible to ensure easy portability.

## 6. RELATED WORK

Tennenhouse et al. [24] proposed "capsules", datagrams carrying small fragments of code, and an implementation in the form of an IP option [25]. 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 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 ANTS [26] as an optimization where packets carry pointers to code which is loaded initially from the previous hop along a packet's path. This approach optimizes the consumed overall bandwidth for the drawback of a considerable initial delay. (Our scheme potentially suffers from a similar delay problem but it can be addressed 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 worth pursuing. In another document, the usefulness of active reliable multicast is shown [16] using the ANTS platform and results in terms of significant performance gains are measured in [15]. [15] also proposes various applications of active network technologies like sensor data mixing and inspired us to follow similar ways.

Smart Packets [18] is another interesting capsule related approach. The main focus is on implementation of extended diagnostic functionality in the network. A new compact programming language 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.

Zegura et al. [6,27] presented theoretical analysis on network subsystems and 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 on the example of congestion control for MPEG video streams. The functions referred to in the data packets are loaded out-of-band into the 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 analytic model to evaluate the performance gains offered by the caching and show that it might be an application of active networking worth pursu-

ing. The system proposed here builds in part on the theoretical background and terminology introduced in [5], but extends the system's capability to downloading function modules in-band, on-the-fly, and elaborates on the infrastructure needed to do so.

SwitchWare [22] is another "programmable switch" project which allows out-of-band loading of program modules into network nodes. The "Switchlets" are sent to the input port of the switch to enhance the switch's functionality. Switchlet modules are written in a language (SML/NJ) 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. Active Bridging [2] is an application of SwitchWare which shows reprogramming of a bridge with Switchlets.

## 7. CONCLUSIONS

This paper elaborates on an idea which shows a novel application and combination of known active networks technologies. By introducing a relatively simple modification to the network subsystem which allows it to upgrade and enhance automatically, fast deployment of new network protocols as well as application specific data processing in network nodes can be achieved.

Next, we plan to refine the system's definition and implement the Active Network Node as described. We will further look into the problems of binding to the right code server fast enough to provide reasonable delay bounds for the case of new active modules and finding the optimal large scale arrangement for code servers. For IP, we will propose a new option type which introduces a 24 bit protocol identifier in order to reference a large number of active modules and a "version" field to identify upgrade implementations of a given module. We will demonstrate both application specific data processing as well as centralized protocol migration and deployment and take various measurements, e.g. on initial connection setup delay due to function downloading or on propagation time for the spreading of new protocol feature initiated by a broadcast from a code server throughout an Intranet.

## 8. ACKNOWLEDGMENTS

The authors would like to thank John DeHart and Fred Kuhns for proof-reading this paper as well as Zubin Dittia and Hari Adishesu for valuable input and for inventing the acronym of the system described.

## 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., Calvert, K., Zegura, E., "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*, November 1994
- [9] Chaney, T., et al., "Design of a Gigabit ATM Switch", In *Proceedings of INFOCOM'97*, April 1997.
- [10] Decasper, D., Waldvogel M., Dittia, Z., Adishesu, H., Parulkar, G., Plattner B., "Crossbow - A Toolkit for Integrated Services over Cell Switched IPv6", In *Proceedings of the IEEE ATM'97 workshop*, May 1997
- [11] Deering, S., Hinden, R., "Internet Protocol, Version 6 (IPv6), Specification", *RFC 1883*, December 1995
- [12] 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.
- [13] Eastlake, D.E., "Domain Name System Security Extensions", *draft-ietf-dns-sec-secext2-02.txt*, November 1997
- [14] Johnson, D., Perkins, C., "Mobility Support in IPv6", *draft-ietf-mobileip-ipv6-03.txt*, July 1997
- [15] Legedza, U., Wetherall D., Guttag, J., "Improving the Performance of Distributed Applications Using Active Networks", In *Proceedings of INFOCOM'98*, April 1998
- [16] Lehman, L., Garland, S.J., and Tennenhouse, D., "Active Reliable Multicast", In *Proceedings of INFOCOM'98*, April 1998
- [17] Papadopoulos, C., Parulkar, G., and Varghese, G., "An Error Control Scheme for Large-Scale Multicast Applications", In *Proceedings of INFOCOM'98*, April 1998
- [18] Partridge, C., Jackson, A., "Smart Packets", *Technical report*, BBN, 1996
- [19] Partridge, C., Mendez T., Milliken W., "Host Anycasting Service", *RFC 1546*, November 1993
- [20] Parulkar, G.M., Schmidt, D.C., Turner, J.S., "altPm : A Strategy for Integrating IP with ATM," In *Proceedings of SIGCOMM'95*, August 1995.
- [21] Rivest, R.L., Shamir, A., and Adleman, L.M., "On Digital Signatures and Public Key Cryptosystems", *Technical Report, MIT/LCS/TR-212*, January 1979
- [22] Smith, J., Farber, D., Gunter, C., Nettles, S., Feldmeier, D., Sincoskie, W., "SwitchWare: Accelerating Network Evolution", *White Paper*, June 1996
- [23] Stevens, W., "TCP/IP Illustrated Volume 1 - The protocols", *Addison-Wesley Professional Computing Series*, 1994
- [24] Tennenhouse, D., Smith, J., Sincoskie, W., Wetherall, D., Minden, G. "A Survey of active network Research", *IEEE Communications*, January 1997
- [25] Wetherall, D., Tennenhouse, D., "The ACTIVE IP Options", In *Proceedings of the 7th ACM SIGOPS European Workshop*, September 1996
- [26] Wetherall, D., et al., "ANTS: A Toolkit for Building and Dynamically Deploying Network Protocols", In *Proceedings of IEEE OPENARCH'98*, April 1998
- [27] Zegura, E., "CANes: Composible Active Network Elements", Georgia Institute of Technology, <http://www.cc.gatech.edu/projects/canes/>