Activating Networks

J. M. Smith[*], K. Calvert[†], S. Murphy[‡], H. Orman[§] and L. L. Peterson[¶]

November 13, 1998

Abstract

Research in active networking has resulted in a variety of working systems and experimental results obtained from these systems. More importantly, the active network research community has been identifying abstractions and principles with which first-generation results can evolve into large scale active networks serving many disparate applications. These abstractions are captured in architectures for execution environments, security and a node operating system, all of which we describe in this paper.

1 Introduction

Active networks are networks which are programmable or customizable to application requirements. Thus, systems such as web proxy caches and firewalls can be seen as simplistic active networking technologies, and infrastructures with control plane programmability such as "open signalling" as increasingly flexible and sophisticated approaches to customizing network infrastructures. Research in active networks, however, has taken perhaps the most ambitious stance, that of allowing "on-the-fly" modification of packet-switching infrastructures by packets, either in a switch-like model where active packets are intermixed with other packets, or in the "capsule" model, where all packets are regarded as programs[TSS+97].

The research has now produced a variety of results, including working infrastructures such as ANTS[WGT98], PLAN[HKM+], and Alien[ASNS97], and applications of these infrastructures to problems ranging from reliable multicast[wLGT98] and auctions[LWG98] to more traditional networking tasks such as error control[FMS+98], congestion control and bridging[ASNS97].

It has become clear that a general architecture for active network elements can be defined using a separation of concerns. Applications for the switch model, the capsule model, or both will require an "execution environment" (EE) in which the programmable services are made available. The

[*]University of Pennsylvania. Work at Penn was supported by DARPA under Contract #N66001-96-C-852

[†]University of Kentucky

[‡]Trusted Information Systems

[§]University of Arizona and DARPA

[¶]Princeton University

EE will accept valid programs and packets, perhaps execute them ephemerally or modify state within the EE, and may emit one or more new packets or programs. The EEs can be thought of as the programming environments of active networks, and are often centered around a particular programming language or programming model.

There are likely to be a multiplicity (although probably a small number) of EEs. A single monolithic EE might manage hardware resources itself within the runtime system. However, when node resources are shared, protection between EEs must be enforced for these resources, and the likely overlap in core services motivates the separation of some functions into a "Node Operating System" (NodeOS).

A high-level illustration of a system architecture based on these ideas is given in Figure 1.

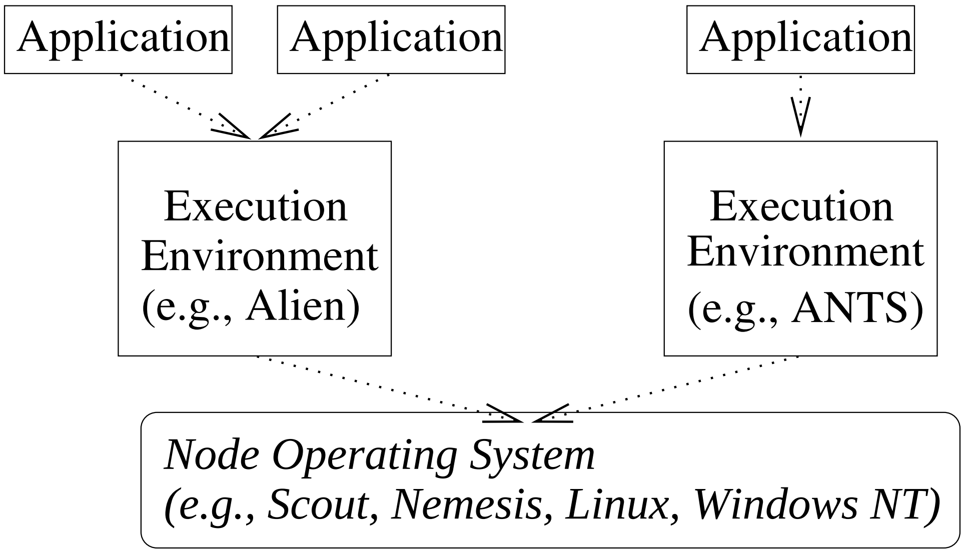

Figure 1: High-Level Active Networking Node Architecture

Finally, as security is a central concern in active networks, we require an architecture in which principals can be defined, identified and authenticated. Further, the principals must have associated rights which are enforced remotely by provision of cryptographically-protected credentials.

The remainder of the paper is organized as follows. Section 2 elaborates on the active network architecture. Section 3 surveys the execution environments, and Section 4 discusses the Node OS. The security architecture is discussed in Section 5. Section 6 briefly discusses some novel applications of active networks, and Section 8 concludes the paper, both with an elaboration on the design space and where we can go next.

2 Active Network Architecture

A network architecture identifies functional components—e.g., end systems and intermediate systems such as routers or switches—and the protocols required for them to cooperate in providing end-to-end services to users of the network. Examples of such services include global addressing and connection-oriented or connectionless data transport service. An active network provides a *platform* on which different network services can be developed and deployed, possibly simultaneously, rather than a fixed service. The functional components of that platform, and the interfaces between them, are defined by the active network architecture.

A primary objective of the architecture is to minimize the amount of global agreement required to deploy new end-to-end services, and allow maximum flexibility for in the nature of those services. These services are assumed to be packet-based (as opposed to, say, circuit-based), and the network itself is assumed to exist for the purpose of sharing transmission resources (and not, say, to support computation *per se*). The separation of concerns mentioned in the previous section follows from the requirement for flexibility: components responsible for implementing end-to-end services (the EEs) are separated from components that provide building-block functions in support of those services. In addition to the nodeOS, the latter components include a security policy database and enforcement engine and a management interface. The functionality of the EEs is limited to what can be implemented using the services provided by these other components via the nodeOS interface, but is not otherwise constrained by the architecture. In particular, the nature of the API exported to the users of the network is completely up to the EE. It might be an extended Java Virtual Machine, or it might resemble the popular "sockets" interface, with added capabilities.

In addition to the requirement for maximum flexibility, the architecture should satisfy some additional requirements. Performance is an important characteristic in networks, and the active network architecture should not preclude fast-path processing optimizations, so that packets needing only "plain old forwarding service" need not be penalized in terms of performance. In addition, the architecture should scale to support very large global active networks.

The basic model of operation is the same for all active nodes: packets arrive on some physical link, and are immediately classified based on their contents (i.e. headers), and placed in logical *channels* according to this classification. Each channel has some associated protocol processing, which may include security checks. A channel delivers a packet either to an EE for interpretation and/or processing, or to an output physical link. EEs may also originate packets and place them in output channels, which provide processing and, ultimately, transmission on a physical link. This model, which applies to both end systems and intermediate systems, allows the traditional forwarding function provided by current networks to be viewed as simply another EE; this provides a backward-compatible deployment avenue.

EEs, channels, and other resources are discussed further in subsequent sections.

3 Execution Environments

3.1 Active Network Transport System - ANTS

3.2 The ALIEN Active Loader

3.3 Netscript

3.4 Packet Language for Active Networks - PLAN

4 Node Operating System

Because multiple execution environments may run on a single active node, as illustrated in Figure 1, the node must run an operating system that manages the available resources—memory regions, CPU cycles, link bandwidth—and multiplexes them among the EEs running on the node. Because we also want to simplify the process of porting EEs to multiple underlying OSs, the active network architecture specifies an EE-to-NodeOS interface, the so-called "NodeOS interface".

While one option for the NodeOS interface is to simply use the Posix interface, and to implement EEs as Unix processes, we chose not to do this for two reasons. The first is that the primary function of an active node is to forward network packets; Unix, and the Posix interface, are not well-suited for that objective. The second is that in order to provide adequate support for quality of service, the NodeOS must allocate resources at a finer level of granularity than per-EE. When an EE requests a service from the NodeOS, the request may be accompanied by an identifier (and possibly a credential) for the entity in whose behalf the request is made. This entity may be the EE itself, or another party—a network packet flow—in whose behalf the EE is acting. The NodeOS is then able to schedule these fine-grain entities more effectively.

The NodeOS interface defines four primary abstractions: *thread pools, memory pools, channels*, and *flows*. The first three encapsulate a system's three types of resources: computation, storage, and communication. The fourth abstraction, the flow, is used to aggregate control and scheduling of the other three abstractions.

4.1 Abstractions

This section introduces the four main abstractions, and explains the relationships among them.

Thread Pool

The *thread pool* is the primary abstraction for computation. Each flow contains a single thread pool that is initialized when the flow is created. Three parameters must been specified when creating a thread pool: (1) the maximum number of threads in the pool, which limits concurrency; (2) a cycle rate, which limits how much of the processor the pool is allowed to consume; and (3) a scheduler object, which specifies how the threads are scheduled for execution.

From the perspective of the NodeOS interface, the thread pool primarily exists for accounting purposes. Whether or not a given NodeOS pre-allocates the specified number of threads is an implementation issue. Moreover, even if the NodeOS does pre-allocate some number of threads, these threads may not be able to handle all computation that takes place on behalf of the thread pool; for example, they may not be allowed to run in supervisor mode. Any thread running on behalf of the thread pool, no matter how it is implemented, will be charged to the pool.

Memory Pool

The *memory pool* is the primary abstraction for memory. Each flow contains a single memory pool that is initialized to contain those pages shared by its parent when the flow is created. A thread running in the flow can then map virtual memory regions into the pool, unmap regions out of the pool, and control various attributes of these virtual memory regions (*i.e.*, set and unset read, write, execute, and shared attributes.)

Because many EEs will not want to directly manage virtual memory, it is expected that the NodeOS will also define a liballoc library that supports the familiar malloc, free and realloc calls.

Channels

Flows create channels to send, receive, and forward packets. Some channels are *anchored* in an EE; anchored channels are used to send packets between the execution environment and the underlying communication substrate. Anchored channels are further characterized as being either incoming (inChan) or outgoing (outChan). Other channels are *cut-through* (cutChan), meaning that they forward packets through the active node—from an input device to an output device—without being intercepted and processed by an EE.

When creating an inChan, the flow must specify two things: (1) which arriving packets are to be delivered on this channel (i.e., it must associate a demultiplexing key with the channel), and (2) a buffer pool that queues packets waiting to be processed by the channel. This buffer pool is created out of the flow's memory pool. When creating an outChan, the flow must specify how much link bandwidth (bits-per-second) the channel is allowed to consume (guaranteed to get). Cut-through channels both receive and transmit packets, so they must specify a demultiplexing key, a buffer pool, and the link bandwidth.

It is helpful to define channels in terms of a more primitive building block—*protocol modules*—which implement some well-defined primitive communication service. For example, ANEP, TCP, UDP, IP, an ethernet driver (ETH), and an ATM driver (ATM) are protocol modules with the obvious meaning. Protocol modules can be combined (stacked) to build more complex communication services. We use a path name to denote the protocol stack that implements a channel. For example, /ANEP/UDP/IP/ETH would identify an anchored channel (it could be either an inChan or an outChan) that includes the ANEP protocol running on top of UDP (as illustrated later in Figure 10 of Section 7.1), and so on. Similarly, ATM/IP/ETH would identify a cutChan that forwards packets from an ATM device, through IP, to an Ethernet device.

Note that a given NodeOS need not implement channels as stacks of composable protocol modules, although a NodeOS can leverage such an implementation to support extensible channels.

How a given NodeOS supports extensible channels, if it does at all, is not exposed to EEs through the NodeOS interface. EE's need to be able to name the protocol stack that implements a channel, but the NodeOS interface does not provide EEs with a means to extend the set of modules available to be stacked.

In the case of inChans and cutChans, the EE must provide enough information to allow the NodeOS to classify incoming packets and demultiplex them to the appropriate channel. Much of this demultiplexing information will be implied by the path name and the addresses passed as attributes when the channel is created. For example, by knowing the path name /TCP/IP and peer addresses that include TCP port numbers, the NodeOS is able to correctly demultiplex incoming packets using whatever packet classifer it chooses. There are two complications, however.

The first is that the EE may want to segregate different EE-visible data streams onto different channels. For example, EE number 47 (where 47 is the ANEP demux identifier that corresponds to this EE) might want to open two /ANEP/UDP/IP channels, each supporting a distinct EE-specific flow of packets. While one can imagine arbitrarily many levels of demultiplexing within an EE—as well as all the other complications that make packet classifiers hard—the NodeOS will support only one additional level of demultiplexing. That is, the EE is allowed to specify a single ⟨offset, length, key⟩ that the NodeOS uses to partition EE-visible packet streams onto separate channels. This 3-tuple says to match a length-byte key at offset bytes in the packet payload (i.e., after the header of the channel's topmost protocol has been stripped).

The second complication is due to cut-through channels, which sometimes want to capture packets with a far richer demultiplexing key than is implied by the protocol stack that implements the channel. For example, cut-through channel ATM/IP/ETH might be forwarding packets that match a much longer /TCP/IP demultiplexing key. Fortunately, cut-through channels that behave like this are always created relative to existing anchored channels, and so they can inherit the existing channel's demultiplexing information. In fact, the cut-through channel *replaces* the anchored channel in such a situation. Thus, cut-through channels can be created in one of two ways: either as a brand new channel (in which case its demultiplexing key is implied by the constituent protocol modules) or as a replacement for an existing anchored channel (in which case its demultiplexing key is the same as that of the existing channel.)

Flow

The flow is the primary abstraction for accounting, admission control, and aggregate scheduling in the system. System resources, such as CPU cycles, memory, and network bandwidth are allocated to particular flows. As shown in Figure 2, a flow typically contains the following resources: an inChan, an outChan, a memory pool and a thread pool. We think of active packets arriving on the inChan, being processed by the EE using threads and memory allocated to the flow, and then transmitted on the outChan.

Note that channels consume not only network bandwidth, but also CPU cycles and memory buffers. The threads that shepherd messages across the flow's channels come from the flow's thread pool and the cycles they consume are charged to that pool. Similarly, the I/O buffers used to queue messages on a flow's channels are allocated from (and charged to) the flow's memory pool.

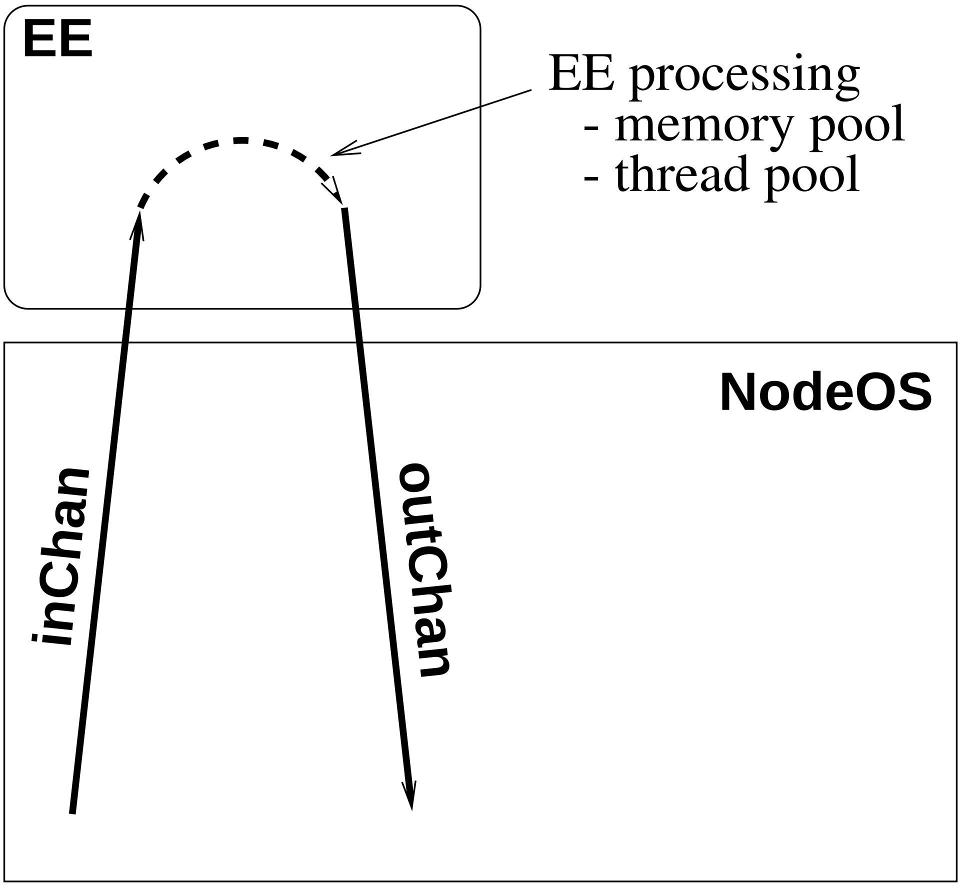

Figure 2: A flow consists of an inChan, outChan, and memory/threads needed for EE-specific processing.

In addition to these standard resources, a flow might also include a cut-through channel, as shown in Figure 3. In such a scenario, one would expect "control" packets to be received on the inChan, processed by the EE, and sent on the outChan, while the cutChan forwards "data" packets. In this case, cycles and memory used by the cut-through channel are also charged to the flow's thread and memory pool.

Flows not only contain memory, threads, and channels, but they can also contain sub-flows. This results in a hierarchy, with interior nodes of the hierarchy corresponding to flows, and the leaf nodes corresponding to channels, memory, and threads. An example flow hierarchy is shown in Figure 4; for simplicity, it shows only channels (no threads or memory). This example illustrates how flows can be either as "broad" as an entire EE (corresponding to a Unix process) or as "narrow" as a single packet stream (corresponding to the traditional use of the term "flow" in networking).

The root flow corresponds to the NodeOS. By convention, the flows at the first level of the hierarchy (Flow1, Flow2, and Flow3) correspond to execution environments. Flows at lower levels of the hierarchy are EE-specific. In this example, the EE implemented in Flow1 has chosen to implement independent packet streams in their own flows (Flow4 through FlowN), while the EEs

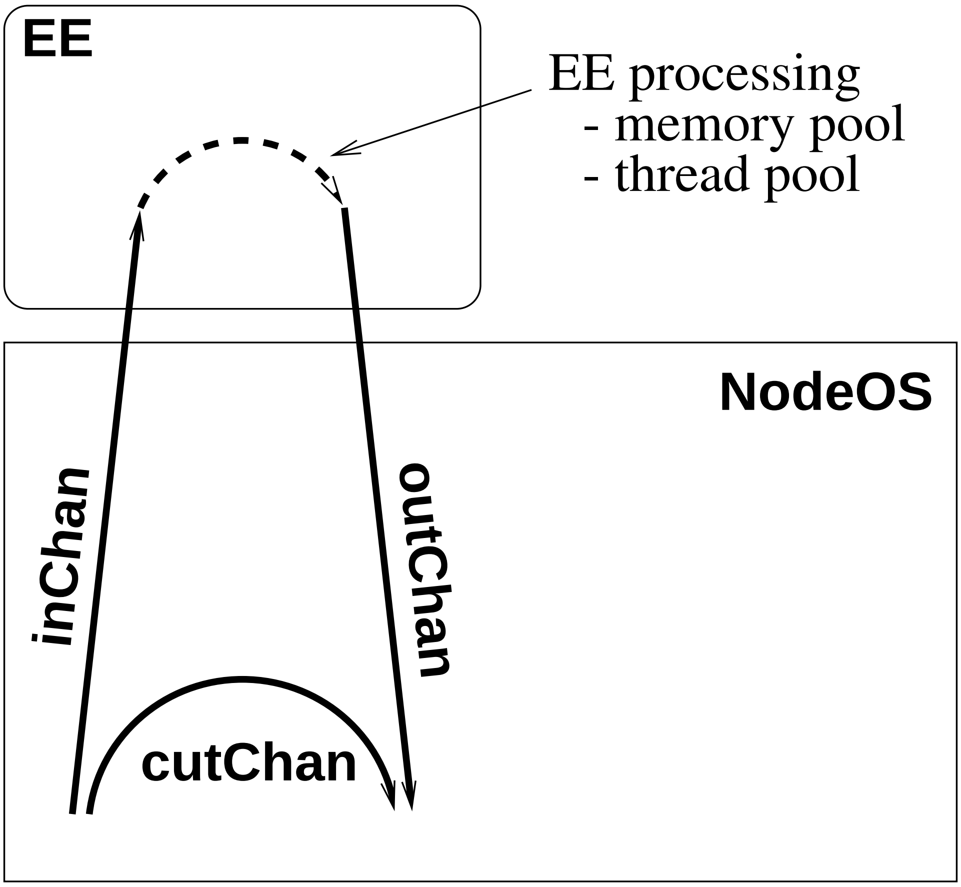

Figure 3: A flow with a cut-through channel.

running in Flows 2 and 3 aggregate all packets on a single set of channel, memory, and thread resources. The advantage to using narrow flows, as in the case of the EE running in Flow1, is that the individual packet streams are visible to the NodeOS, thereby allowing the NodeOS to allocate and schedule resources on a per-stream basis. (Flow1 also has its own inChan and outChan, which might carry EE control packets that belong to no specific sub-flow.)

Although each flow in the hierarchy has its own set of channels, a thread pool, and a memory pool, the total amount of resources used by a parent flow and all of its descendants is bound by the parent flow's resource limits. For example, if Flow1 in Figure 4 is granted 1MB of memory, then the amount of memory used by the subtree rooted at Flow1 can be no more than 1MB.

Channels and threads belong to a single flow, but there is a more complex relationship between memory and various flows in the hierarchy since it is possible to share memory among multiple flows. Specifically, a parent flow has full access, to any memory mapped by any of its descendant flows. However, children flows do not have access to memory that is mapped by their parents unless that memory was explicit mapped "shared" by the parent. Similarly, flows must explicitly set the "shared" attribute in order to share memory with sibling flows.

A flow can be destroyed by the flow itself, by the parent flow that created it, or by the NodeOS

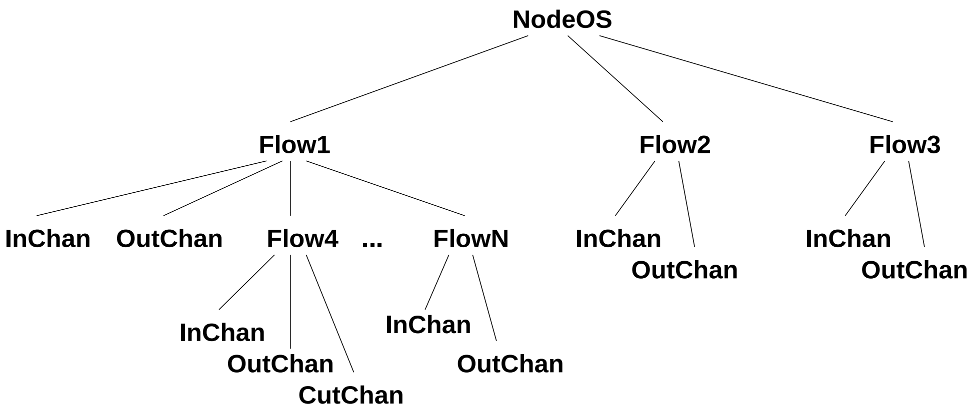

Figure 4: Flow hierarchy.

because the flow has violated some resource usage policy. When a flow is destroyed, all of its children flows are also destroyed. The parent flow is notified, and any unused resources belonging to the destroyed flows are credited back to the parent flow.

5 Security Architecture

An active network retains all of the security concerns from traditional networks. The users of the active network will be concerned with the authenticity, the integrity and the confidentiality of the data communicated through the network. But the movement of arbitrary computation into the infrastructure of the network introduces additional security concerns at each level of the architecture. Whereas the concern in traditional networks is focussed on the possibility of damage to user data and end-nodes, active networks must also be concerned with the possibility of damage to the robustness of the active network as a whole through the agency of the active code in the active packets. The active nodes in the infrastructure could be harmed by active code, either by modification of the node state or denial-of-service attacks through excessive resource usage. The execution environments themselves implement their own services and resources that are subject to damage from the active code.

Consequently, the security protections must follow the movement of computation into the infrastructure, so that the network, each node and each EE is properly protected. The network has no single entity to provide its protection; the protection of the robustness of the network as a whole must be built into the design of the individual nodes and EE's. For example, some existing EE's have defined a network resource that is consumed during an active packet's traversal of a node. This resource prevents a single active packet from placing an infinite demand on the network.

5.1 Node Security

In contrast to the network, protection of an active node or execution environment can be self-enforced. The active nodes protect themselves by ensuring that activity within the node on behalf of the active packet is safe and properly authorized. Safety can be ensured by the use of a variety of techniques, such as the use of type safe languages or software fault isolation. Authorization is expressed in terms of access to portions of the node state, either to protect its confidentiality (e.g., preventing unauthorized exposure of any keying material used to cryptographically protect the node's communications) or its content (e.g., preventing unauthorized modification of the node configuration parameters). Authorization is also expressed in terms of limitations on resource usage, as discussed in previous sections. The fundamental motivation for such limitations is to prevent denial-of-service attacks against the node (and thereby indirectly against the network). Clearly, validating the source authenticity and integrity of the active code are crucial in ensuring proper authorization. If the source can be spoofed, then the node can be deceived into taking actions, based on the apparent source, that will cause harm. If the active code can be changed, then it can be made to perform actions other than those intended by the author, possibly inducing harm as a result.

The execution environments offer services and resources, composed from the node services and resources, to the active code. An execution environment will want to ensure that active code that uses or accesses its resources or services has the proper authorization. It will want to protect the integrity, the authenticity and perhaps even the confidentiality of those services and resources, just as for the node.

5.2 Authorization in an Active Network

Authorization, then, is the fundamental security service that must be provided to protect either the active node or the execution environment. The difficulty is in providing a mechanism for authorization that is suited to the scale of active networks - anything from an enterprise network governed by a single administration with a homogeneous model and policy to a wide area network including a multitude of authorization models and policies. For scalability and manageability, authorization cannot always be expressed in terms of unique entities and single services or resources. Aggregates of entities must be created, identified by some security attribute that policy relates to authorization for access to services and resources. For further abstraction, the services and resources could be similarly aggregated and identified by security attributes, so that authorization is decided using the requesting entity's security attribute and the service's or resource's security attributes. These aggregates could be any of the aggregates that have been widely used in creating secure systems, such as roles, groups, an ordered set of labels, domains or types.

5.3 Policy Enforcement

The NodeOS enforces a policy for authorization to the NodeOS services and resources, e.g., the flows, memory pools, thread pools and channels mentioned in previous sections. Because the flows are the requesters of memory, threads, and channels in this architecture, a principal must be associated with each flow as it is created. It is the authorization of this principal that decides the accesses and usage requested by a flow. It is not necessary that each flow have a separate principal. In keeping with the spirit of active networks, it should be possible to modify the policy dynamically. Dynamic modification of the policy requires a separation of the storage of the policy and the enforcement of the policy into a policy database and an enforcement engine. The enforcement engine in the NodeOS refers to the policy database to decide each controlled access. Changes to the policy database, possibly through active code, can effect a immediate change in policy.

The execution environment has its own services and resources to protect according to its own policy. For example, an execution environment may offer specialized routing services and a routing table to the active code it executes. The execution environment has a policy governing who is allowed to use the routing table, who is allowed to update the routing table, and who is allowed to use the routing services. However, each of these services and resources is composed from services and resources provided by the NodeOS and access to the NodeOS level objects must be decided in accordance with the Node OS policy. The NodeOS policy and the execution environment policy may conflict. There is no guarantee that the NodeOS policy for access to the memory pool that underlies the routing table, for example, meets the execution environment's policy. The NodeOS policy must be enforced but it must be possible for the execution environment to enforce a stricter policy.

There are several ways to divide the responsibility for enforcement of policy between the execution environment and the NodeOS. An execution environment could choose to defer all authorization decisions to the NodeOS. This would require that the execution environment communicate to the NodeOS its policy governing access to those NodeOS services and resources that underly each of its own services and resources. The NodeOS would have to be capable of partitioning the resources it has associated with the execution environment according to the EE's policy.

An easier solution would be for the execution environment to enforce its own policy, with its own policy database and its own enforcement engine. Indeed, the execution environment cannot be prevented from performing authorization checks of its own for a service or resource it provides. If it denys access, then the access or usage is not provided. (As a corollary, a NodeOS policy cannot *guarantee* provision of an new service for a channel that is anchored in an EE. It must trust that the EE will perform the service.) After the execution environment has decided to grant access to a service or resource, it will attempt to access the NodeOS services and resources from which its own services and resources are constructed. The nodeOS can enforce its own policy when deciding access to the underlying NodeOS services and resources.

When making use of the NodeOS services and resources, the execution environment may or may not indicate the principal associated with the active code that induced the request. There is no way to force the execution environment to do so, as the execution environment may make requests unrelated to any active code. If it does not indicate the principal behind the request, then the NodeOS makes the decision based on the principal associated with the execution environment's flow. The execution environment would be accountable for the usage of NodeOS resources. Note, however, that in this case the NodeOS cannot ensure that the execution environment enforces the NodeOS policy. It must trust the execution environment to manage the resources granted to it in accordance with the NodeOS policy. This means that the NodeOS must communicate its policy to the execution environment on startup, as an initial policy database.

5.4 Identification of Principals

The final problem becomes the distribution of the representation of principals and their authorizations. In an enterprise network in a trusted environment, it may be possible to store identifiers and security attributes directly in the packet. In such an environment, the meaning of identifiers and the policy governing such identifiers are known everywhere and the source and integrity of the packets is trusted. For example, NFS packets carry the UNIX user id and user's group id used in making file access checks between the file server and the client. The user id's and group id's are trusted to have the same semantics, represent the same principal, throughout the NFS network. In more wide spread and heterogeneous environments, identification of principals and their authorizations can not presumed to be widely known, uniform or trusted. A system of authorities must be established who can attest to the authorizations granted to a principal.

In an untrusted environment, cryptography is used to create a binding of principal and authorizations and the packet. Normally, the principal is associated with a key that is used to provide some cryptographic service for the principal. An authority binds the keying material and some designation of the authorization of the principal into a *certificate* or *credential*. The credential carries the *digital signature* of the authority, using a key bound to the authority. An infrastructure of authorities is created, with authorities attesting to the binding of keys to other authorities. Finally, the credential must be bound to the packet by the initiator to prevent its use in another packet. The usual mechanism is to employ the key represented in the credential to cryptographically bind the packet and credential with a digital signature.

One active packet can contain the credentials of multiple entities. The authorizations associated with the initiator of the packet are clearly important to security. The authorization of the author of the active code could be important in some node policies. Relevant attributes of the code, such as the proof for proof-carrying code or evaluations of its quality by independent authorities, might be part of some node policies. A significant challenge in active networks will be to create enough uniformity of security requirements that application developers can comply with these requirements and effectively use the security features of the network, while still maintaining the flexibility to tailor security services to the purposes of the applications and individual networks.

Finally, the source authenticity and integrity of the packet must be maintained to derive any assurance from the credentials contained in the packet. When the principal threat is from external intruders, protection of the integrity of the packet between neighboring active nodes can be accomplished with low cost symmetric cryptographic algorithms using shared keys. This ensures that the packet came from the neighboring node and has not been modified in transit. However, when it is not possible to implicitly trust every active node in the network, more assurance than just the immediate source of the packet is needed. In that case, the ultimate source and integrity of the packet must be ensured. The digital signature that binds a credential to a packet also provides this protection if the credential is associated with the source of the packet.

6 Novel Applications of Active Networks

6.1 Active Reliable Multicast

Active Reliable Multicast[wLGT98] (ARM) embeds active routers in a multicast tree. The routers differ from conventional routers in their handling of loss recovery in a reliable multicast setting. The availability of processing and storage can also be used to enhance performance.

The central ideas in the loss recovery improvement scheme are the use of duplicate NACK suppression (to inhibit NACK implosion), the use of "best-effort" caching of multicast data (in case retransmission is required for recovery), and the use of "local" multicasts in the retransmission process (to reduce bandwidth required for data retransmission).

ARM has the desirable architectural properties that not all nodes need be active, and that there are no "necessary" routers in the loss recovery scheme. The result is a robust scalable system for reliable multicast, which shows significant benefits (in recovery latency, scalability, reduced bandwidth required for recovery, *etc.*) under both analytic and simulation studies, particularly when compared to SRM[FJM+95].

6.2 Protocol Boosters

Protocol boosters [FMS+98] are protocol elements which can be inserted into and removed from existing protocols, with the intention of building protocols dynamically. Such dynamic protocol construction may offer advantages in both protocol performance and the rate of evolution of network technology, as the methodology for dynamic construction allows rapid development of specialized protocols. An illustration of "boosting" a subnet is shown in 5.

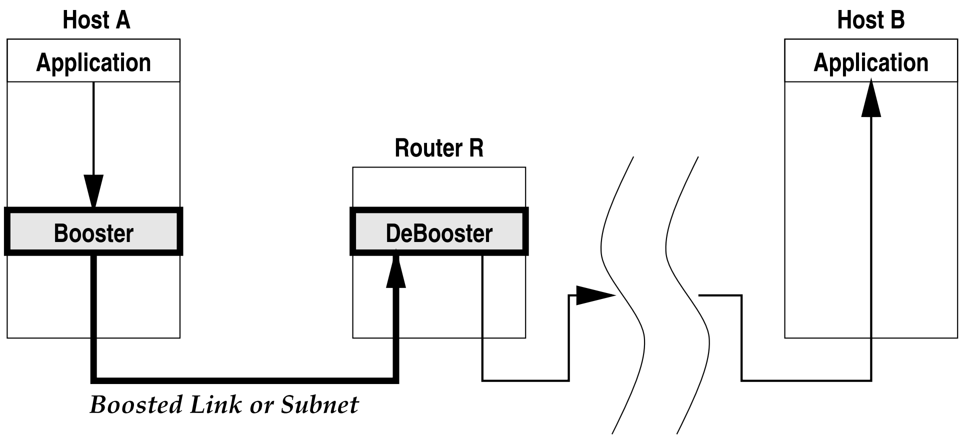

Figure 5: Flexible subnet enhancements with protocol boosters

Protocol boosters have been developed for error control and congestion control. While the protocol boosters architecture preceded the current research in Active Nets, it is clear that Active Networks provide an ideal infrastructure for implementing protocol boosters.

6.3 ACC: Active Congestion Control

Due to its flexibility and presence in the network, active networking offers considerable promise for improving congestion control[BCZ96, ASNS97]. This promise has been explored in the design of Active Congestion Control[Fab98] (ACC). ACC takes advantage of state and programmability to improve feedback congestion control by reducing the delay with which congestion is signalled to sending systems. ACC packets contain small (*e.g.,* 4–8 byte) characterizations of the state of the endpoint's congestion feedback scheme (perhaps a congestion window size at the sender). When congestion occurs at a router, in particular, when a packet is dropped, the router (1) determines what congestion window size will result presuming the packet is dropped, (2) deletes packets that would not be sent with this new window size, and (3) informs the sender of the new window size. As noted by Faber[Fab98], nodes beyond the congestion point see traffic which looks as if the sender had reacted instantly. Thus, ACC is particularly powerful in network contexts which have large $bandwidth * delay$ products.

ACC was evaluated by applying the basic model in the context of TCP/IP congestion control (thus ACC was presumed to be embedded in IP routers). The simulation studies showed that ACC can achieve up to an 18% improvement in throughput when traffic is bursty.

6.4 Improvement of Distributed Systems: Auctions

On-line auction servers collect bids from clients for items and provides the current price of an item on request. Given the time-scales with which such servers can operate (short) and the possible delays in packet arrival in an internet (long), outdated (*i.e.,*, lower than the current price) bids can arrive. Depending on the scale of the auction, these bids, which have become irrelevant, can generate considerable server load.

The active protocol suggested by Legedza, *et al.*[LWG98] deletes low bids with active elements embedded in the network connecting client and server. When the server becomes busy, it enables these elements and periodically updates their notion of the current price. In dropping low bids, they thus serve as proxies for the server in its attempts to focus on relevant bids.

6.5 ARC and Detour

One possibility for rapid deployment of active technology to enhance the existing Internet is *Active Router Control*[SAM+98], where active elements serve as an enhanced programmable management and measurement system for the IP Internet. Such a configuration is shown in Figure 6, using a set of routers as a logical "router in a room".

The basic active "blob" of Figure 6 can be replicated throughout an Internetwork, with the active elements using the managed Internet routes as link layers. A set of active nodes can be grafted into a larger collection of routers and forwarders to create an active Internet.

Researchers at the University of Washington have proposed a similar idea in their Detour[SAA+98] project, but using a high-performance workstation cluster to provide a different blend of computing performance.

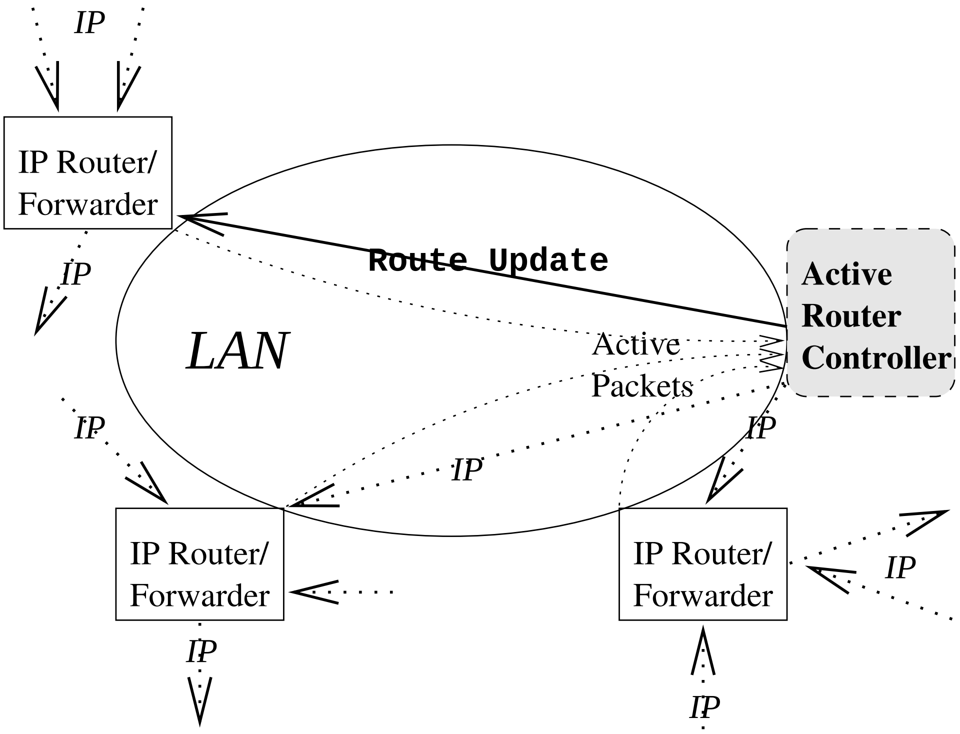

Figure 6: An Active Router Controller (ARC) managing a set of forwarder/routers

7 Interoperability

As discussed in Section 3, there is a variety of approaches being explored in the design space of execution environments. This variety is needed both to meet independent research goals, and perhaps more importantly, to cover the design space. Further, the applications discussed in Section 6 rely on different EEs and exploit their various features and domain-specific functions (such as the link layer support in Alien).

To support this interoperation, an Active Network Encapsulation Protocol (ANEP)[ABG+] was designed. The main goals of ANEP, as discussed below in Section 7.1, are to accelerate interoperability experiments and to use existing infrastructures, *e.g.,* to create active tunnels.

7.1 The Active Network Encapsulation Protocol - ANEP

The near-term solution devised is the use of an encapsulation protocol for active packets. In ANEP terminology[ABG+], a *packet* consists of an *ANEP Header* and a *payload*. An *Active Node* is a network element which can evaluate active packets. The packet header format is shown in Figure 7. Much of the information in the header is organized as <Type, Length, Value> triples called TLVs. The *Basic Header* is the first two words (8 octets) of the ANEP Header.

The packet header is aligned on 32-bit boundaries. At present, Version is 1, and the packet is discarded if the value is unknown. For this version of ANEP, only the most significant byte of

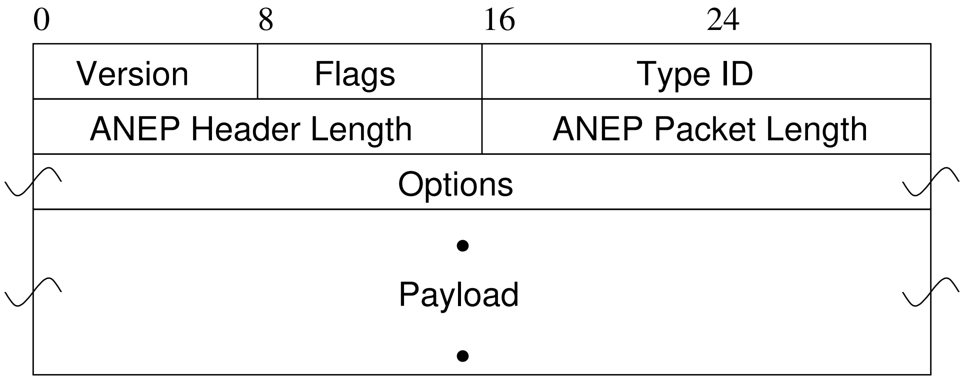

Figure 7: Active Network Encapsulation Protocol Header

Flags is used; a value of 0 indicates that forwarding should be attempted by the default means in any case, while a value of 1 indicates that the ANEP packet should be discarded if the TypeID is not recognized. The ANEP Header Length is a 32-bit number (in units of 32-bit words), and can be used in combination with the 32-bit ANEP Packet Length byte count to calculate the size of the options. The TypeID is used to indicate an EE for the message; values are currently assigned by ANANA, which is currently Bob Braden of ISI.

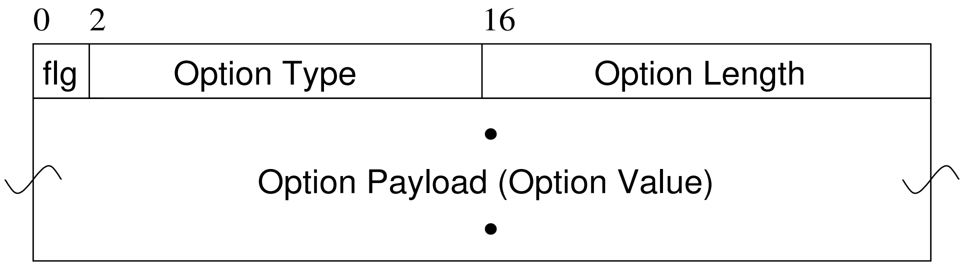

Figure 8: Structure of ANEP Options

The options consist of zero or more TLV constructs following the Basic Header, as illustrated in Figure 8. Option Type is a 14-bit number used to interpret the Option Payload; values are set by ANANA. Several values (1 for Source ID, 2 for Destination ID, 3 for Integrity Checksum and 4 for Non-Negotiated Authentication) are reserved. The Option Length field is the positive-valued TLV length in 32-bit words.

The format for the Source ID, Destination ID and Non-negotiated Authentication types is shown in Figure 9. For Source and Destination IDs (which uniquely identify the sender and receiver) the 32-bit Scheme Identifier identifies the addressing scheme used to interpret the variable-sized Option Payload. Reserved values are 1 for 32-bit IPv4 addresses, 2 for 128-bit IPv6 addresses and 3 for 48-bit 802.3 addresses (the last two octets are 0). The non-negotiated authentication type provides 1-way authentication. The Option Payload is a 32-bit authentication scheme, followed by the data required for the scheme. Reserved values are 1 for SPKI self-signed certificates and 2 for X.509 self-signed certificates.

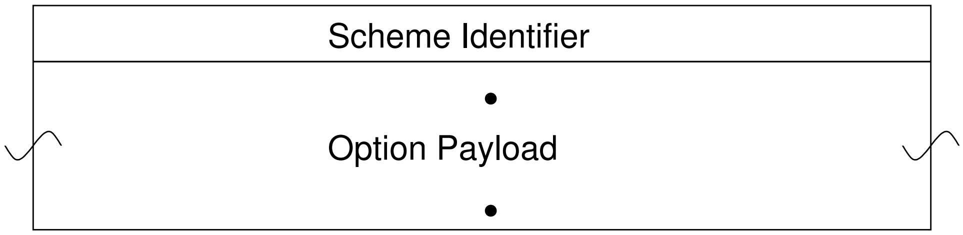

Figure 9: Illustration of Option Type Values

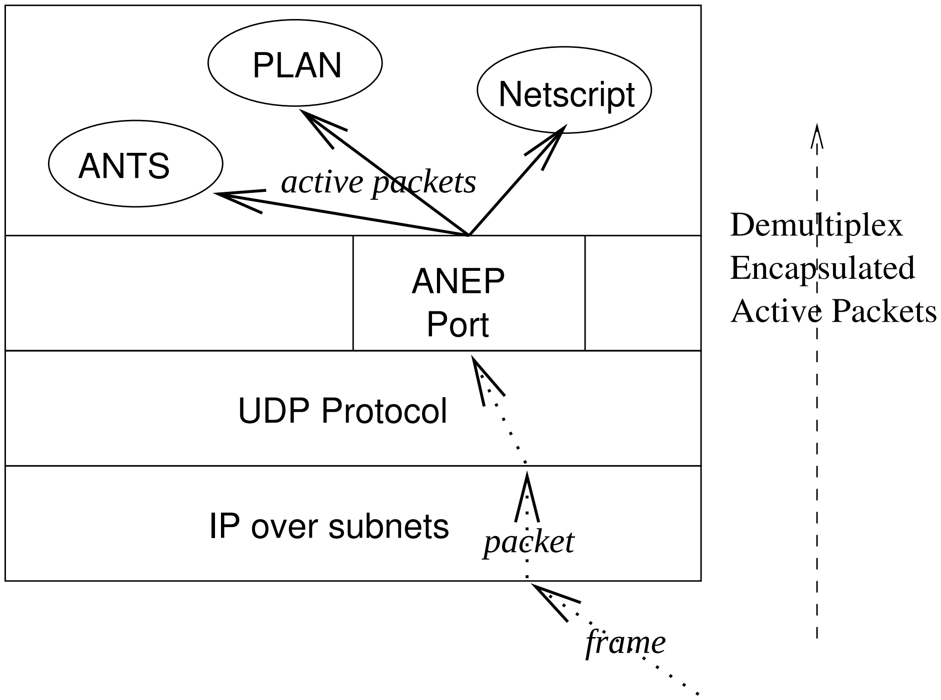

Figure 10: The PLAN ANEP demultiplexing architecture

Figure 10 illustrates how the active packets would be demultiplexed to the appropriate EE.

The active network community devised ANEP as a way to bootstrap itself with existing infrastructure and we do not view it as an end-product of the research program. Nonetheless, it is proving useful in the role we have set for it, especially in developing a prototype "ABONE".

7.2 The ABONE

The ABONE[Bra98] is used for testing EEs and deploying Active Applications (AAs) which use the EEs. The goal is to have a 1000+ node ABONE. ABONE nodes are currently UNIX machines operating the anetd program which performs active network node management, such as initiating EEs. At present, the ABONE is flat, although Braden's proposal[Bra98] suggests that a hierarchy based on locally-administered Ahosts at edges and intermediate Arouters at the core may accelerate progress towards the goal of a 1000+ node ABONE. The ABONE supports ANTS, Netscript and PLAN/Alien at end-1998.

8 Conclusions and Next Steps

Ad-hoc "active" networking is already present in a variety of forms in the Internet, either as overlays or as specialized router enhancements. Examples include WWW proxies[CDN+96], RSVP[CSZ92], MBONE and firewalls[CB94]. Each is a customization of the network infrastructure (whether overlaid or embedded) used to provide enhanced services, enhanced performance, or enhanced security. In the spirit of minimizing complexity, finding a unified general solution will serve to accelerate network evolution. Active Networks represent such a solution. The major challenges to their deployment will be the tradeoffs achieved between usability, flexibility, security and performance.

Cost and performance issues will dictate where and when active network technology will be deployed. With current processing and transmission technologies, operating in the control plane will be the short-term solution. For the long term, there are potentially interesting schemes for limiting the model of "computing"; for example, programmable hardware such as the P4 [HS97] organizes field-programmable gate arrays (FPGAs) in a pipeline, allowing processing at 155 Mb/s line rates. Other methods of gaining performance include restricting computing operations to a locus within the active packet (for example by examining only headers rather than entire packets [CJRS89]), resulting in per-packet processing costs that are not per-byte processing costs. Over the long term, we believe that the cost and performance tradeoffs will tilt in favor of active networks when a total cost of operation model is used.

In the near term, we are deploying EEs on the ABONE, which is now at about 100 ANEP/IP-interconnected machines. A large-scale challenge problem will be attempted in the next year to accelerate progress towards a truly interoperable network programming environment. Over the next several years we will be deploying many other applications over the ABONE to develop an understanding of robustness and scalability issues, provide tests of security and usability, and to demonstrate the advantages of the idea. We expect that during this period, we will winnow the EEs down to a small number, in which development of new features and services will proceed.

References

[ABG+] D. Scott Alexander, B. Braden, C. Gunter, A. Jackson, A. Keromytis, G. Minden, and D. Wetherall. ANEP: Active Network Encapsulation Protocol. Active Networks Group RFC #1, `www.cis.upenn.edu/ switchware/ANEP`.

[ASNS97] D. Scott Alexander, Marianne Shaw, Scott M. Nettles, and Jonathan M. Smith. Active Bridging. In *Proceedings, 1997 SIGCOMM Conference*. ACM, 1997.

[BCZ96] Samrat Bhattacharjee, Ken Calvert, and Ellen W. Zegura. On Active Networking and Congestion. Technical report, Georgia Institute of Technology, 1996. Technical Report GIT-CC-96-02.

[Bra98] Bob Braden. Another Modest Proposal for the ABone, November 1998.

[CB94] B. Cheswick and S. Bellovin. *Firewalls and Internet Security: Repelling the Wily Hacker*. Addison-Wesley, 1994. ISBN 0-201-63357-4.

[CDN+96] Anawat Chankhunthod, Peter B. Danzig, Chuck Neerdaels, Michael F. Schwartz, and Kurt J. Worrell. A Hierarchical Internet Object Cache. In *Proceedings, Winter USENIX Technical Conference*, pages 153–163, January 1996.

[CJRS89] David D. Clark, Van Jacobson, John Romkey, and Howard Salwen. An Analysis of TCP Processing Overhead. *IEEE Communications Magazine*, 27(6):23–29, June 1989.

[CSZ92] D. Clark, Scott Shenker, and L. Zhang. Supporting real-time applications in an integrated service packet network: Architecture and mechanism. In *Proceedings, 1992 SIGCOMM Conference*, pages 14–26, August 1992.

[Fab98] T. Faber. ACC: Using Active Networking to Enhance Feedback Congestion Control Mechanisms. *IEEE Network*, 12(3):61–65, May/June 1998.

[FJM+95] Sally Floyd, Van Jacobson, Steven McCanne, Ching-Gung Liu, and Lixia Zhang. A Reliable Multicast Framework for Light-Weight Sessions and Application-Level Framing. In *ACM SIGCOMM '95*, pages 342–356, October 1995.

[FMS+98] D. C. Feldmeier, A. J. McAuley, J. M. Smith, D. S. Bakin, W. S. Marcus, and T. M. Raleigh. Protocol Boosters. *IEEE Journal on Selected Areas in Communications, Special Issue on Protocol Architectures for the 21st Century*, 16(3):437–444, April 1998.

[HKM+] Michael Hicks, Pankaj Kakkar, Jonathan T. Moore, Carl A. Gunter, and Scott Nettles. PLAN: A Programming Language for Active Networks. Submitted to PLDI'98. Available in `www.cis.upenn.edu/~switchware/papers/plan.ps`.

[HS97] Ilija Hadžić and Jonathan M. Smith. P4: A platform for FPGA implementation of Protocol Boosters. In *Field Programmable Logic 1997*, number 1304 in Lecture Notes in Computer Science, pages 438–447. Springer-Verlag, 1997.

[LWG98] Ulana Legedza, David J. Wetherall, and John Guttag. Improving the Performance of Distributed Applications Using Active Networks. In *Proceedings, IEEE INFOCOM 1998*, pages 590–599, April 1998.

[SAA+98] Stefan Savage, Tom Anderson, Amin Aggarwal, David Becker, Neal Cardwell, Andy Collins, Eric Hoffman, John Snell, Amin Vahdat, Geoff Voelker, and John Zahorjan.

Detour: a Case for Informed Internet Routing and Transport. Technical Report UW-CSE-98-10-05, University of Washington, 1998.

[SAM+98] J. Smith, D. S. Alexander, W. Marcus, M. Segal, and W. D. Sincoskie. Towards an Active Internet, 1998.

[TSS+97] D. L. Tennenhouse, J. M. Smith, W. D. Sincoskie, D. J. Wetherall, and G. J. Minden. A Survey of Active Network Research. *IEEE Communications*, 35(1):80–86, January 1997.

[WGT98] David J. Wetherall, John Guttag, and David L. Tennenhouse. ANTS: A toolkit for building and dynamically deploying network protocols. In *Proceedings, IEEE OpenArch 98*, pages 117–129, 1998.

[wLGT98] Li wei Lehman, Stephen J. Garland, and David L. Tennenhouse. Active Reliable Multicast. In *Proceedings, IEEE INFOCOM 1998*, pages 581–589, April 1998.