Directions in Active Networks*

Kenneth L. Calvert
Department of Computer Science
University of Kentucky
Lexington, KY
calvert@cs.uky.edu

Samrat Bhattacharjee Ellen Zegura
Networking and Telecommunications Group
College of Computing
Georgia Tech, Atlanta, GA
{bobby,ewz}@cc.gatech.edu

James Sterbenz
GTE Laboratories
Waltham, MA
jpgs@acm.org

Abstract

Active networks represent a significant step in the evolution of packet-switched networks, from traditional packet-forwarding engines to more general functionality supporting dynamic control and modification of the network behavior. However, the phrase "active network" means different things to different people. This article introduces a model and nomenclature for talking about active networks, describes some possible approaches in terms of that nomenclature, and presents various aspects of the architecture being developed in the DARPA-funded active networks program. Potential applications of active networks are highlighted, along with some of the challenges that must be overcome to make them a reality.

1 Introduction

Packet-switched networks enable the sharing of transmission facilities so that packets may be efficiently moved among connected systems. Traditional packet networks perform only the processing necessary to forward packets toward their destination. Over time, as computing power becomes cheaper, more and more functionality is being deployed *inside* the network, in an effort to provide better service to users. Examples of such functionality include admission control (to guarantee delay and other performance characteristics for certain classes of users), explicit congestion notification (to enhance the congestion-adaptation of certain applications), packet filtering (to protect end systems from attempts to exploit security holes), and TCP "ack spoofing" (to improve reliability over lossy links).

Active networks represent a "quantum step" in this evolution: by providing a *programmable* interface in network nodes, they expose the resources, mechanisms, and policies underlying this increased functionality, and provide mechanisms for constructing or refining new services from those elements. In short, active networks support dynamic modification of the network's behavior as seen by the user. Such dynamic control is potentially useful on multiple levels:

*Work supported by DARPA. Opinions and views expressed here are those of the authors and are not necessarily representative of their organizations or sponsors. This article was written while Kenneth L. Calvert was with the College of Computing, Georgia Tech.

- From the point of view of a network service provider, active networks have the potential to reduce the time required to develop and deploy new network services. The shared infrastructure of the network presently evolves at a much slower rate than other computing technology. One consequence of being able to change the behavior of network nodes on the fly is that service providers would be able to deploy new services quickly, without going through a lengthy standardization process.

- At a finer level of granularity, active networks might enable users or third parties to create and tailor services to their particular applications and even to current network conditions. Though it seems likely that most end-users would not write programs for the network —after all, individual users can, in principle, program their PC's, but how many do?— it is easy to imagine individuals customizing services by choosing options in code provided by third parties. Indeed, this prospect should appeal to network providers as well, because it enables them to charge more for such value-added services.

- Networks are expensive to deploy and administer. For researchers, a dynamically-programmable network offers a platform for experimenting with new network services and features on a realistic scale without disrupting regular network service.

In this paper, we first introduce a vocabulary for talking about active networks, and describe a range of approaches to building an active network. We then present a snapshot of the architecture being developed in the DARPA active networks program,and discuss some ways in which new services might be composed from basic building blocks. Finally, we briefly highlight some of the research into ways that active networks might be used.

2 Basic Concepts and Nomenclature

An active network is a kind of store-and-forward network. A store-and-forward network consists of a set of *nodes* interconnected by transmission *links*. The purpose of the network is to support the sharing of these transmission facilities. The basic unit of multiplexing of transmission facilities is the *packet*. Nodes receive packets from users and other nodes, perform a computation based on their internal state and the control information (header) carried in the packet, and as a result of that computation may forward one or more packets toward other nodes or to users. The nature of the *network service* is defined by the behavior of the individual nodes of the network, and how users can control that behavior through coded information placed in their packets.

In today's Internet, for example, routers examine the destination address field of the Internet Protocol header along with internal routing tables to determine to which neighbor they should forward the IP packet. The extent of user control over the network's behavior is limited to the range of values that can be placed in that field (and a few others) in the IP header; other services can, however, be envisioned. One example, which has been proposed for the Internetis a "premium" service in which certain nodes classify packets based on information contained in *all* of their headers (TCP or UDP port numbers as well as source and destination IP addresses), and then route and schedule them for transmission on the basis of that classification.

In this discussion we identify the term *user* (or end user) with the originators and recipients of the packets actually carried by the network. Users are, in general, different from *node/network administrations*, which control the configuration and interconnection of network nodes. Often the relationship between user and network administration is that of service provider and subscriber.

What we call the *network application programming interface* (network API) is made up of those aspects of behavior that are visible to the end user, viz., the per-node packet processing behavior and the code through which users control that behavior via the packets they send. We can think of the network API as defining a *virtual machine* that interprets a specific language. The network API for the Internet Protocol comprises the language defined by the syntax and semantics of the IP header and its effect on the routers of the network. In traditional networks the virtual machine is fixed, and the expressive power of the language is very limited.

Active Networks and Programming Interfaces

One way to think about active networks is that they provide a *programmable* network API. If we think of the IP header in the traditional network as the *input data* to a virtual machine, we can think of packets in the active network containing *programs* as well as input data. In the context of this model, a variety of active networking approaches can be characterized by the following attributes:

- **Language Expressive Power.** The degree of programmability of the network API may range from a simple list of fixed-size parameters that select from predefined sets of choices, to a Turing-complete language capable of describing any effective computation (such mechanisms are discussed further in Section 4). The advantage of a less powerful language is that it constrains the possible node behaviors and so simplifies correctness analysis. It is also more likely to admit fast-path optimization, e.g. through special-purpose hardware.

  Many active networking projects, however, have opted for more powerful languages that use typing and other mechanisms to help ensure correctness.[1] Most of these languages feature some form of restriction on their expressive power, order to guarantee that the effect of any packet sent into the network is bounded. For example, a language may admit only straight-line programs, without loops or branches.

- **Statefulness.** Another important characteristic of the network API is the ability to install *state* in the interior nodes of the network, and to refer to state installed by other packets. Some active network APIs provide this capability, while others do not. Where it is present, the API must include control mechanisms to protect users' state from unauthorized access.

- **Granularity of Control.** By this we refer to the scope of node behavior that can be modified by a received packet. One possibility is that a single packet can modify the node behavior seen by *all* packets arriving at the node, and this change persists until it is overridden. At the other extreme, a single packet modifies the behavior seen only by that one packet. Between these extremes, modifications might apply to a *flow*, which we define to be a set of packets sharing some common characteristic, such as temporal locality and/or a particular source and destination address in the headers. In general, the active network API must include security mechanisms that ensure that packets affecting the node behavior have localized effect and/or come from authorized users.

The possibility of programming the network API introduces a new role, namely that of *service developer*: a third party who provides code that can be loaded into the active network to enhance

[1] The ability to express powerful computations is, in itself, not particularly interesting. The utility comes from being able to access node resources (e.g. output queues) from inside a computation. Thus, a key aspect of the programming language is the set of abstractions of node resources it provides.

or customize the service seen by users. Such code might be deployed by users themselves, or by network service providers.

Examples

We briefly describe some active networking approaches in terms of the model and attributes introduced above.

The **ANTS** toolkit [1], developed at MIT, features the notion of a *capsule*, which is a packet containing a byte-coded Java program along with a payload of user data. The ANTS Network API consists of the Java Virtual Machine augmented with the ANTS class, which implements methods permitting capsules to be decoded and interpreted. ANTS supports stateful computation: capsules capsules can install state and invoke classes installed by other capsules. The granularity of control is at the flow/packet level.

Because the standard JVM does not support access to transmission resources at a sufficiently low level, implementations of ANTS (or any other active network API based on Java) on standard platforms cannot support quality-of-service capabilities, and are limited to the basic network capabilities provided by Java. However, work at the University of Arizona [2] is implementing a version of the JVM called Joust, which supports lower-level abstractions for real-time scheduling.

The **SwitchWare** project, at the University of Pennsylvania [3] uses a language called PLAN (Programming Language for Active Networks) as the network API. PLAN is a scripting language that supports some basic primitives, sequential composition, and the invocation of *switchlets*. PLAN does not permit packets to install state in network nodes. Switchlets are API components (programs) that are installed via a separate different network API. The granularity of control of PLAN is per-packet for PLAN; installation of a switchlet, however, makes it globally available at the node.

The **Smart Packets** project at BBN [4] is applying active network technology to assist with the growing problem of managing networks. Two programming languages have been developed: Sprocket, which is a high-level language with built-in features to support network management (e.g. built-in types for accessing management data); and Spanner, which is a CISC assembly language into which Sprocket is compiled. Thus the Network API is implemented by the Spanner virtual machine implemented by a daemon running in an active node. A design goal of the project is to be able to encode meaningful and useful network management programs in less than a kilobyte; thus Spanner provides very compact representations that fit into single packets.

A group of researchers at Columbia University and elsewhere are investigating architectures for **Open Signaling**, focusing on connection management and support for quality of service in ATM networks. The *xbind* implementation [5] is based on a node programming interface comprising a high-level language such as C, plus a set of hardware control primitives provided by the ATM switch manufacturer. Xbind exports an object-oriented, CORBA-based network API, which enables users to modify switch state. The granularity of control is the ATM virtual circuit.

3 Architecture Overview

In this section we present an overview of the architecture under development in the DARPA active networks program. (The reader is cautioned that this is a snapshot, and some aspects of the architecture may still change.) The active *network* architecture deals with global matters like addressing and end-to-end services, which are intended to be programmable (not fixed) in an active

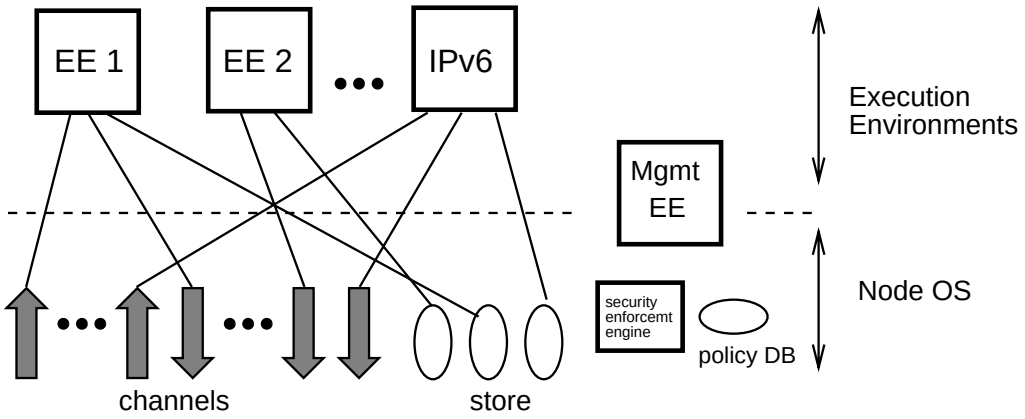

Figure 1: Components

network. The general approach has therefore been to specify a *node* architecture that defines a common base functionality, including how packets are processed, what resources are available at the node, and how they are accessed. Thus, the architecture defines the basic functionality of the active node programming interface, although it does not specify any particular language or encoding for that interface. This approach has the pleasant effect of minimizing the amount of global agreement and standardization required to implement an active network.

The node architecture is explicitly designed to support multiple network APIs simultaneously. Several factors motivate this requirement. First, current active network prototypes occupy disparate points in the taxonomy described earlier, and given our lack of experience it seems desirable to let them be used and compared side-by-side to enhance our understanding. Second, this approach supports the goal of fast-path processing for those packets that just want "plain old forwarding service". A third and related factor is that it provides a built-in evolutionary path, not only for new and existing APIs, but also for backward-compatibility: IPv4 or IPv6 functionality can be provided as simply another network API.

The functionality of the active network node is divided between the *Execution Environments* (EEs) and the *Node Operating System* (NodeOS). The general organization of these components is shown in Figure 1. In terms of the discussion in Section 1, the EE is responsible for implementing the network API, while the NodeOS manages access to local node resources by EEs.

Execution Environments

Each execution environment is analogous to a "shell" program in a general-purpose computing system, providing an interface through which end-to-end network services are provided to users. As noted above, the architecture allows for multiple different EEs to be present on a single active node.[2] All user access to node resources (including transmission bandwidth) is provided through an EE. An EE may provide a very simple service that can be statelessly controlled through user-provided parameters, or it may implement an interpreter for a powerful, stateful programming language, or something in between.

[2]Development of an EE is a nontrivial activity; therefore the total number of EEs is not expected to be large—probably no larger than the number of different shell programs in UNIX.

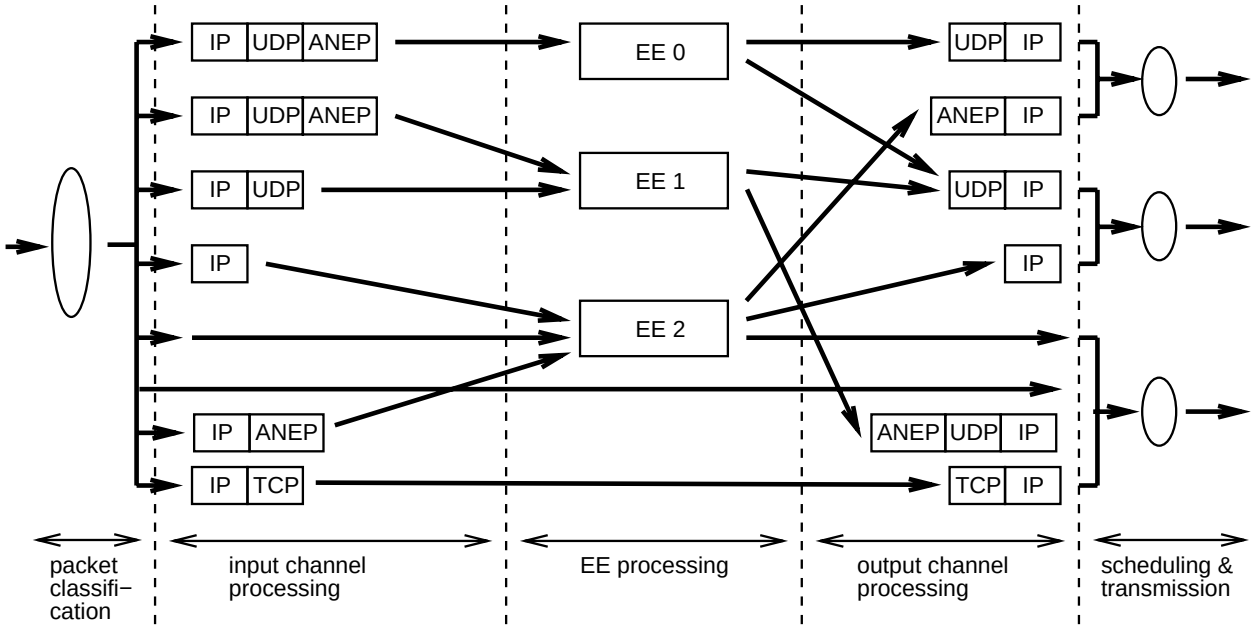

Figure 2: Packet flow through an active node

Node Operating System

The NodeOS provides the basic functions from which EEs build the abstractions that make up net APIs. It manages the resources of the active node and mediates the demand for those resources, including transmission, computing and storage. The NodeOS isolates EEs from details of resource management and the existence of other EEs. The EEs, in turn, hide most (but not all) of the details of interaction with the end user from the NodeOS.

Users and other entities in the network are represented by an abstraction called the *principal*. Security policies are defined in terms of principals; the NodeOS is responsible for enforcement of such policies. When an EE requests a service from the NodeOS, the request is accompanied by an identifier (and possibly a credential) for the principal in whose behalf the request is made. This principal may be the EE itself, or another party (e.g., a user) in whose behalf the EE is acting. The NodeOS presents this information to an *enforcement engine*, which verifies its authenticity and checks that the node's security policy database (see Figure 1) authorizes the principal to receive the requested service or perform the requested operation. EEs may implement their own policies to augment those of the node, but they may not override the NodeOS policies.

The NodeOS implements communication *channels*, over which EEs send and receive packets. These channels consist of physical transmission links (e.g., Ethernet, ATM), plus the protocol processing associated with higher level protocols (e.g., TCP, UDP, IP). The logical flow of packets through an active node is shown in Figure 2. When an active node receives a packet over a physical link, it classifies the packet based on the packet's contents (i.e. headers); each packet is either assigned to an existing channel or discarded.

The mapping of incoming packets to channels is controlled by a pattern specified by the EE when it creates the channel. In the typical case, an EE requests creation of a channel for packets

matching a certain pattern of headers, e.g. a certain Ethernet type or combination of IP protocol and TCP port numbers. It is the responsibility of the security engine to ensure that a given principal is allowed to create a channel with a particular pattern.

To provide for quality of service, the NodeOS has scheduling mechanisms that control access to the computation and transmission resources of the node. These mechanisms isolate user traffic to some degree from the effects of other users' traffic, so that each appears to have its own virtual machine and/or virtual link. When channels are created, the requesting EE specifies the desired treatment by the scheduler(s). Such treatment may include reservation of a specific amount of bandwidth for traffic on the channel, or isolation from other traffic and "fair sharing" of available bandwidth with other channels. Input channels are scheduled only for computation, while output channels must be scheduled for both computation and transmission.

Active Network Encapsulation Protocol

So far we have not specified how users can have their packets routed to a particular EE at a node. The Active Network Encapsulation Protocol [6] provides this capability. The ANEP header includes a "Type Identifier" field; well-known Type IDs are assigned to specific execution environments. (Presently this assignment is handled by the Active Network Assigned Number Authority.) If a particular EE is present at a node, packets containing a valid ANEP header with the appropriate Type ID (encapsulated in a supported protocol stack) will be routed to the appropriate EE.

A packet need not contain an ANEP header for it to be processed by an EE. EEs may also process "legacy" traffic —originated by end systems that are not active-aware— by setting up the appropriate channels. An example of this kind of functionality would be a TCP performance-enhancement service implemented at the border between two regions of the network with different bandwidth/error characteristics.

Interfaces and Standardization

The primary interfaces in this architecture are the user-EE interface (the network API) and the EE-NodeOS interface. The architecture is indifferent to the form of the net API defined by any EE, and thus it may change at any time. (It does, however, need to be carefully designed and specified.)

The EE-NodeOS interface need not be identical from node to node; all that is required is for each node to provide a standard set of basic services to EEs.[3] The definition of this basic set of services is analogous to standardization of the UNIX operating system calls. Work is ongoing in the active network community to gain the experience needed to develop a specification.

Beyond the NodeOS-EE interface, there are only a few facets of the architecture that require standardization; these mainly involve encodings that must be understood by both the end user and the NodeOS. Examples include ANEP, the syntax and semantics of principal identifiers and security credentials, and the units of measure for resource allocation.

[3] It is not difficult to imagine node vendors competing to offer enhanced NodeOS capabilities, and EE-implementors porting their EEs to different node platforms.

4 Composite Network Services

Ultimately, the goal of active networking is to ease the deployment of new network services. This implies that an active network should do more than simply make it possible to install new services. Rather, explicit support should be provided for the process of service creation. An important support feature of a network API is the ability to *compose* services from building blocks. In what follows, we refer to the building blocks for network services as *components*. A network API contains a *composition mechanism* used to create a *composite service* from components. Composition of network services has the usual positive properties of modular design: services need not be built from scratch each time and robust components can be developed incrementally. Further, a composition mechanism may also be used to constrain the set of composite services that can be created, possibly making it easier to reason about the correctness of the overall service and interactions between individual components.

Composite services can take on a variety of forms. A service may execute in its entirety at a single active network node, or it may perform a distributed computation across a set of active nodes. The form of the network API directly affects the sophistication achievable through service composition. For example, if the network API supports only selection of a specific service from a fixed set of choices, then these constitute all of the available "composite" services. At the other extreme, if the network API is a Turing-complete language, an essentially infinite set of composite services can be formed from an available set of service components, using the sequence control constructs of the programming language. In the following section, we outline with examples a set of possible composition mechanisms for active networks.

Composition Mechanisms

A composition mechanism provides a means to specify a composite service constructed from components. Possible approaches to service specification include:

- **Choice from a set of options**

  In this case, the network API supports specification of a scalar argument that selects a pre-defined computation at the network node. This idea can be generalized to a fixed number of scalar arguments, each of which selects a particular pre-defined computation, executed in a pre-defined order. This scheme can be efficiently implemented, and proving the correctness of the composite service is not more difficult than proving the correctness for each of the components. However, in terms of service composition, scalar selection does not provide much flexibility to the end-users. Examples of this scheme are IPv4 and IPv6 in which the user interface to the network is limited to the fields in the IP headers. Correspondingly, the flexibility afforded to users is limited.

- **Turing-complete programming language**

  At the other extreme in expressive power, a Turing-complete programming language forms a generic composition mechanism for statements of the language. The structure of the composition depends entirely upon the statements in the program, and thus the constraints on structure are extremely weak. This is the approach advocated by the ANTS project [1], in which components can be installed in the active node as Java subroutines, and the composite service is a Java program that calls components. Correctness and properties like termination of the composite are typically difficult to prove since the interface is Turing complete.

- **Special-purpose language for composition**

  A restricted language specifically designed for service creation can be used to compose network services. These languages can be designed such that all the composite services created have certain desirable properties, e.g. termination and preservation of the active node's safety. This approach is taken by the Switchware [3] and the Netscript [7] projects, with the languages PLAN and Netscript, respectively. In case of PLAN, the structure of the language guarantees termination of all composite services provided that each of the components terminate as well; Netscript features a dataflow computation model.

- **Event-based framework**

  Dynamic behavior can be incorporated into composition by structuring the composition as an event-driven computation and "binding" code modules to specific events. This approach has been used in other protocol composition frameworks, in particular to compose *micro-protocols* in the *x*-kernel [8]. The Language-Independent Active Networking Environment (LIANE) composition model (described next) is an example of such event-based composition.

Service Composition in LIANE

We provide an example of an approach to service composition that falls into the event-driven framework category. Composition in LIANE has two parts. First, the user selects an *underlying program* from amongst those offered by an active node. There will typically be a small number of underlying programs, and these are installed by the node provider (possibly using a priviledged network API). The underlying program provides a basic service (e.g., forwarding) and includes a set of *processing slots* to be used for customization (e.g., to replace the default forwarding table with a customized forwarding table). Each processing slot is associated with a specific execution point in the underlying program.

In the second part of composition, the user selects or provides a set of *injected programs* used to customize the underlying program. The injected programs can either be supplied by the user, or provided by component developers. Each injected program is "bound" to one or more processing slots. The injected program is "eligible" for execution when the appropriate slot is reached ("raised"). More than one injected program may be bound to the same slot; the order of execution of instructions belonging to different injected programs bound to the same slot is non-deterministic. This style of composition has advantages with respect to proving properties about the composite service based on properties of the underlying program and preservation of properties by the injection process. A similar approach to service composition is being developed in the Active Reservation Protocol project at USC/ISI.

Figure 3 illustrates a basic forwarding service that supports forwarding to a small number of destination addresses [4]. This service has four parameters. Two are required: a source address S and a list of destination addresses A. Two are optional: a forwarding table identifier R and a selection function M used to match addresses with forwarding table entries. If the optional parameters are not supplied, defaults will be used. The service also has several slots where the user may supply policies. Each slot has a default policy, indicated in the square brackets and used if the user does not supply an alternative. For example, Slot 2 is reached if the list of output interfaces turns out to be null. By default, no error message is generated; the user may choose, for example, to send an error message to the source. The other slots provide opportunities to control action taken upon

[4] The service is simplified for exposition. An actual forwarding service would necessarily be more complex.

Parse packet to obtain S, A, R, M
$\langle$**Slot 0:[null]**$\rangle$
Outputlist := ()
For each address a in A:
    Let interface i := **Lookup**(a, R, M)
    $\langle$**Slot 1:[Add** i **to Outputlist]**$\rangle$
if Outputlist = () then $\langle$**Slot 2:[null]**$\rangle$
$\langle$**Slot 3:[null]**$\rangle$
For each unique i in Outputlist:
    Create a copy D of the packet,
        with $A' = \{a : (i, a) \in$ Outputlist$\}$
    if i is congested
        then $\langle$**Slot 4:[discard]**$\rangle$
    else $\langle$**Slot 5:[null]**$\rangle$
        enqueue D for i.

Figure 3: Basic forwarding service

packet receipt and further processing (e.g., sending notification, incoming and outgoing topology information to the source, application of per-interface policy, flow-specific congestion control).

5 Applications

As mentioned in the Introduction, the dynamic control enabled by active networks allows services that are tailored to current network conditions. These services have the potential to improve performance seen by applications, as compared to end-system-only solutions. In this section we provide examples of active networking to improve performance of applications. We then provide more detail on one application, namely, multicast video distribution.

5.1 Examples

Efforts to improve performance using active networking span a wide variety of areas. Amongst the areas that appear to be most promising are:

- **Multicast**

  The "traditional" IP multicast service hides from its users the details of the routing topology and the number and location of receivers. For unreliable multicast, this approach makes sense and allows scaling to larger applications; however, there are inherent problems in using this model when it is desired to deliver data to all receivers *reliably*. The difficult arises in recovering from losses, which typically affect all receivers downstream in the multicast tree from the point of a loss. A common approach is to spread the responsibility for multicast retransmissions among the receivers, to avoid over-burdening the sender. For good performance, this requires that receivers be aware of nearby receivers, that are above the loss point, to provide retransmission.

By including state and processing in the network, retransmissions can be directed to nearby receivers [9] or can come from caches at network nodes [10]. Both approaches reduce the delay and transmission resources required for retransmission.

- **Quality of service**

  Network conditions such as the presence of congestion or a lossy link can significantly degrade the quality of application streams. Schemes that require the sender to adapt to network conditions have well-known limitations, including the time for the sender to detect the condition, react, and transmit the adapted data to the receiver. During the adaptation interval, the receiver will experience uncontrolled losses (when conditions worsen) or less-than-optimal performance (when conditions improve). By moving information about how to adapt to network conditions into network nodes, the appropriate type of adaptation can occur when and where it is needed.

  Efforts in this area have included transparent in-line protocol "boosters" to adapt to network conditions (e.g., by adding forward error correction over error-prone links) [11] and intelligent discard strategies for preserving the quality of MPEG video in the face of network congestion [12]. An interesting combination of multicast and MPEG video distribution is examined later in this section.

- **Caching**

  A substantial fraction of all network traffic today comes from applications in which clients retrieve objects from servers (e.g. the World-Wide Web). The *caching* of objects in locations "close" to clients is an important technique for reducing both network traffic and response time for such applications. Caching schemes require decisions about where to locate objects and how to forward requests between caches. Current wide-area caches are manually configured into a static hierarchy, thus incurring an administrative burden and limiting the ability to react to dynamic conditions.

  Efforts to use active networking for caching include using network mechanisms to route cache requests to pre-configured cache locations [10], and combining small caches with information about the contents of nearby caches, at each network node [13]. The Adaptive Web Caching project is developing an architecture for an adaptive, scalable web caching system, with mechanisms that could be realized using active networking or application-layer protocols [14].

- **Network management**

  The conventional approach to network management is to poll managed devices from a management station, requesting the values of variables and checking for anomalies. This approach concentrates the intelligence in the management stations, resulting in processing and communication bottlenecks. Further, the poll-and-check approach severely limits the ability to track problems in a timely and efficient manner.

  Several projects are considering the use of active networking to improve network management. The Smart Packets project [4] at BBN is developing architecture, languages, and protocols for making managed nodes programmable. The Netscript project [7] at Columbia is developing techniques to automatically create systematic management instrumentation and respective MIBs, using the structure of active elements.

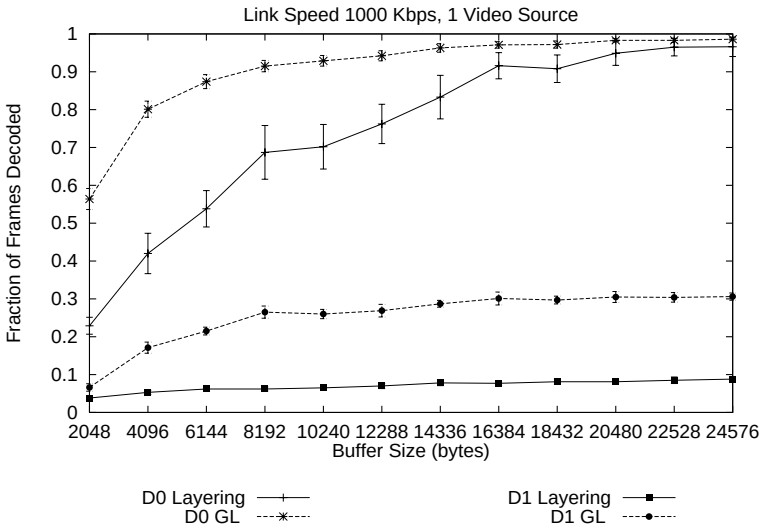

Figure 4: Single video source, varying buffer size, 1000 Kbps Link

5.2 Multicast Video Distribution

We conclude this section by providing a snapshot of the performance improvement offered by active networking for multicast video distribution. Multicast video distribution is particularly challenging due to heterogeneity in the paths from the sender to the receivers and the problems of scaling for large numbers of receivers. The classic approach to dealing with heterogeneity and scale is the use of receiver-based adaptation schemes. Under these schemes, the sender transmits multiple streams suitable for a range of path characteristics. Each receiver joins the stream (or streams) that can be supported by the individual path from the sender.

Active networking allows another option for the location of the adaptation, namely: in the network. The basic operation of network-based adaptation is as follows: the sender transmits a full rate stream on a single multicast group, to which all receivers subscribe. The routers in the multicast tree have information (established *a priori* by out-of-band mechanisms and/or carried in-band as part of data packets) about how to intelligently reduce the rate of the stream in the face of congestion on an outgoing link. For routers in the tree that do copying, the rate adaptation occurs after copying the multicast packet. Thus each outgoing link is treated separately and each receiver obtains performance based on the path from the sender, not influenced by other receivers or parts of the multicast tree. The adaptation occurs when dictated by congested conditions, thus an increase in available bandwidth can immediately be utilized and a decrease in available bandwidth immediately triggers controlled reduction of the rate.

We compare a network-based adaptation strategy with a receiver-based layered strategy for distribution of MPEG video. We use a topology that has two receivers: one relatively uncongested (D0) and one heavily congested (D1). The details of operation of the two approaches and the experimental configuration can be found in [15].

Figure 4 shows the fraction of frames decoded at each receiver (D0 and D1) for network- and receiver-based, as the amount of buffering at the routers varies. A frame is considered decoded only if the receiver gets all packets in the frame and all packets in all *dependent* frames, i.e., those frames used by MPEG for the relative encoding. The network-based adaptation curves are labeled

"GL" for GOP-level discard; the receiver-based adaptation curves are labeled "Layering".

For the congested receiver (D1), the network-based adaptation consistently delivers 2-3 times more decodable frames than layering. For the uncongested receiver (D0), the network-based scheme has far better performance at smaller buffer sizes. This is because the receiver-based adaptation "over-reacts" to transient congestion and drops one of the streams, which accounts for 60% of all the frames. As the buffer sizes are increased, the network is able to absorb the transient bursts, and the performance of receiver D0 under receiver-based adaptation approaches 100% goodput. However, increasing buffers do not help in case of long-lived congestion as is seen in the case of the congested receiver (D1).

6 Future Directions

We have presented an overview of approaches to active networking, and discussed some potential applications. We have omitted many details, and the reader should bear in mind that many problems must be addressed to make this vision of active networks a reality. Among the interesting issues and challenges are:

- What form of access to low-level node resources is compatible with maintenance of a "fast path" for packet processing.
- Development of scheduling mechanisms and policies to deal with combined computation (protocol processing) and transmission bandwidth requirements on output channels.
- Definition of common mechanisms to protect network resources —transmission and computation bandwidth, and storage— from waste due to misbehaving network components. An example of such a mechanism is the "time to live" field in IP, which ensures that packets are forwarded a bounded number of times. In an active network, where the relationship between received and transmitted packets at a node is less straightforward, a more sophisticated mechanism may be required; the challenge is to keep it scalable.
- Approaches to state installation that enable users to place functionality at the "right" place in the network —for example, branch points in multicast distribution trees— while hiding most details of the topology.

Acknowledgments

The architecture described here is the result of contributions by the researchers in the DARPA active networks community, including (but not limited to) Scott Alexander, Bob Braden, Carl Gunter, John Guttag, Gary Minden, Sandy Murphy, Scott Nettles, Hilarie Orman, Larry Peterson, Jonathan Smith, and David Wetherall. Errors, inconsistencies, and omissions are the responsibility of the authors.

References

[1] D. Wetherall, J. Guttag, and D. L. Tennenhouse. ANTS: A toolkit for building and dynamically deploying network protocols. In *IEEE OPENARCH'98*, San Francisco, CA, April 1998.

[2] Larry Peterson et al. Scout Joust. `http://www.cs.arizona.edu/scout/joust.html`.

[3] Jonathan Smith, David Farber, Carl A. Gunter, Scott Nettles, Mark Segal, William D. Sincoskie, David Feldmeier, and Scott Alexander. SwitchWare: Towards a 21st century network infrastructure. Whitepaper.

[4] Beverly Schwartz, Wenyi Zhou, Alden Jackson, W. Timothy Strayer, Dennis Rockwell, and Craig Partridge. Smart Packets for Active Networks. BBN Technologies, http://www.nettech.bbn.com/smtpkts/smart.ps.gz, 1998.

[5] Ivan Ming-Chit, Weiguo Wang, and Aurel Lazar. A comparative study of connection setup on a concurrent connection management platform. In *Proceedings IEEE OpenArch '98*, April 1998.

[6] D. Scott Alexander et. al. Active Network Encapsulation Protocol (ANEP). http://www.cis.upenn.edu/ switchware/ANEP/docs/ANEP.txt, 1997.

[7] Y. Yemini and S. da Silva. Towards programmable networks. In *IFIP/IEEE International Workshop on Distributed Systems: Operations and Management*, L'Aquila, Italy, October 1996.

[8] Nina T. Bhatti and Richard D. Schlicting. A system for constructing configurable high-level protocols. *Proceedings of SIGCOMM '95*, 25(4):138–150, October 1995.

[9] G. Parulkar C. Papadopoulos and G. Varghese. An error control scheme for large-scale multicast applications. In *Infocom'98*, 1998.

[10] Ulana Legedza, David J. Wetherall, and John Guttag. Improving the performance of distributed applications using active networks. IEEE Infocom'98, 1998.

[11] D. Bakin, W. Marcus, A. McAuley, and T. Raleigh. An FEC Booster for UDP Application over Terrestrial and Satellite Wireless Networks. In *Int'l Mobile Satellite Conference (IMSC 97)*, 1997.

[12] S. Bhattacharjee, K. L. Calvert, and E. W. Zegura. An Architecture for Active Networking. In *Proceedings of High Performance Networking 97*, 1997.

[13] S. Bhattacharjee, K. L. Calvert, and E. W. Zegura. Self-organizing wide-area network caches. In *IEEE Infocom'98*, 1998.

[14] L. Zhang, S. Michel, K. Nguyen, A. Rosenstein, S. Floyd, and V. Jacobson. Adaptive web caching: Towards a new global caching architecture. In *3rd Int'l WWW Caching Workshop*, 1998.

[15] S. Bhattacharjee, K. L. Calvert, and E. W. Zegura. Network support for multicast video distribution. Technical Report GIT-CC-98-16, College of Computing, Georgia Institute of Technology, 1998.