A Survey of Active Networks

Adel A. Youssef
adel@cs.umd.edu
Department of Computer Science
University of Maryland
College Park, MD 20742
CS-TR-4422
May 1999

ABSTRACT **2**

1. INTRODUCTION **2**

2. A GENERAL ARCHITECTURE FRAMEWORK **5**

2.1 EXECUTION ENVIRONMENTS 7
2.2 NODE OPERATING SYSTEM (NODEOS) 7
2.3 ACTIVE NETWORK ENCAPSULATION PROTOCOL (ANEP) 9
2.4 DESIGN OBJECTIVES 9

3. CODE DISTRIBUTION **10**

3.1. A DISCRETE (OUT-OF-BAND) APPROACH 10
3.2. CAPSULES - AN INTEGRATED (IN-BAND) APPROACH 11

4. COMPOSITE NETWORK SERVICES **11**

5. CURRENT RESEARCH **13**

5.1 ANTS (MIT) 13
5.2 SWITCHWARE (UNIVERSITY OF PENNSYLVANIA) 16
5.3 NETSCRIPT (COLUMBIA UNIVERSITY) 17
5.4 CANES & LIANE (GEORGIA TECH) 18
5.5 SMART PACKETS (BBN TECHNOLOGIES) 20
5.6 LIQUID SOFTWARE (UNIVERSITY OF ARIZONA) 21
5.7 PROTOCOL BOOSTERS (BELLCORE) 22
5.8 ARCHITECTURAL FEATURES SUMMARY 23

6. CHALLENGES FOR DEPLOYMENT **24**

6.1 SECURITY 24
6.2 PERFORMANCE 26
6.3 INTEROPERABILITY 27
6.4 BACKWARDS COMPATIBILITY 28

7. APPLICATIONS OF ACTIVE NETWORKS **29**

7.1 RELIABLE MULTICAST 30
7.2 NETWORK CONGESTION 31
7.3 WIRELESS SERVICES 32

8. CONCLUSIONS **33**

REFERENCES **35**

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 network behavior. However, the phrase "active network" means different things to different people. This survey 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. Also, a snapshot of the current research issues and activities of different institutions is provided. Potential applications of active networks are highlighted, along with some of the challenges that must be overcome to make them reality.

1. Introduction

Traditional data networks passively transport bits from one end system to another. A packet in a passive network carries only data and this data is passed opaquely without examination or modification from node to node. The action that a router takes on a packet is specified independently from the end application that generated the packet. The role of computation within such networks is extremely limited, e.g., header processing in packet-switched networks, signaling in connection-oriented networks, and simple Quality-of-Service (*QoS*) schemes (i.e., priority schemes via packet marking) with packet processing independent of packet contents. Because of this extremely limited computation over a

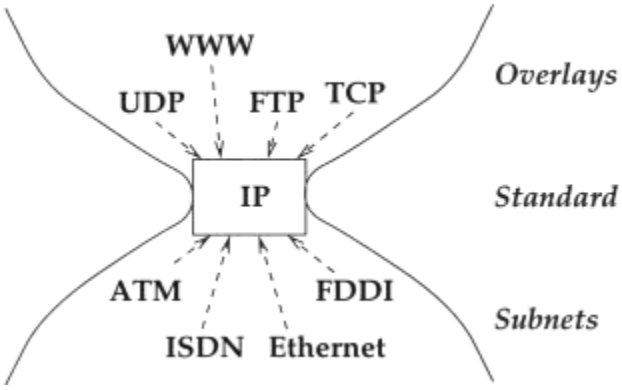

Figure 1 – Hour glass model of internetworking

packet's contents and the decoupling of user control over network behavior, passive networks are often referred to as *store-and-forward networks*.

New applications can benefit from new network services that are tailored to the characteristics of an end host to achieve optimal performance, e.g., customizing a video feed to the display, processing, and connectivity characteristics of different end-user devices such as Personal Digital Assistants (*PDA*s) and laptop computers. However, wide-scale deployment of new services is too slow due to the long standardization process and the backwards compatibility required for the existing network infrastructure. *RSVP* (September 1997) [1], Mobile IP (October 1996) [2], *IPv6* (December 1995) [3], and IP multicast (August 1989) [4] are perfect examples of slow service deployment-none is in common use today.

The fundamental design goal for new Internet services is interoperability. As shown in Figure 1, the idea is to have a wide variety of high-level services and low-level networking technology that interoperate by funneling them through the common IP interface. IP defines a standard packet format and virtual source and destination addressing mechanism that enables interoperation of different networking systems. Its success can be seen by its worldwide acceptance and penetration in the marketplace and its enabling of other services, such as the World Wide Web. However, when new functionality is needed but can not be added either above or below the IP layer, then this layer must be modified. The task of incorporating new functionality, such as support for *QoS* in the Internet, is subject to a lengthy standardization process which includes determining the effects on the existing infrastructure. It is this need to standardize on the IP interoperability layer which makes network evolution so slow.

Research into mechanisms to provide new services includes proxies, firewalls, and transport gateways. These solutions are usually ad hoc and tailored to specific users and applications. Ideally, the goal is to replace the ad hoc approaches to network-based computation with a flexible, generic capability that enables uncoordinated deployment of new services and protocols.

In 1994, the Defense Advanced Research Projects Agency (*DARPA*) research community introduced the concept of active networking. Active Networks [5-9] are different from traditional passive networks in that they allow the network to perform

customized computations on the user data. All nodes in an active network support equivalent computational models, which enables users to effect different computations on different packets. For example, a user of an active network could send a customized compression program to a node within the network (e.g., a router) and request that the node executes that program when processing their packets. Hence, active networks are referred to as *store-compute-and-forward* networks. Figure 2 is a simple depiction of the difference between the passive and active networking paradigms.

These networks are "active" in two ways:

- Routers and switches within the network actively process, i.e., perform computations on, the user data flowing through them.
- Individual users and/or administrators can inject customized programs into the network, thereby tailoring the node processing to be user and/or application specific.

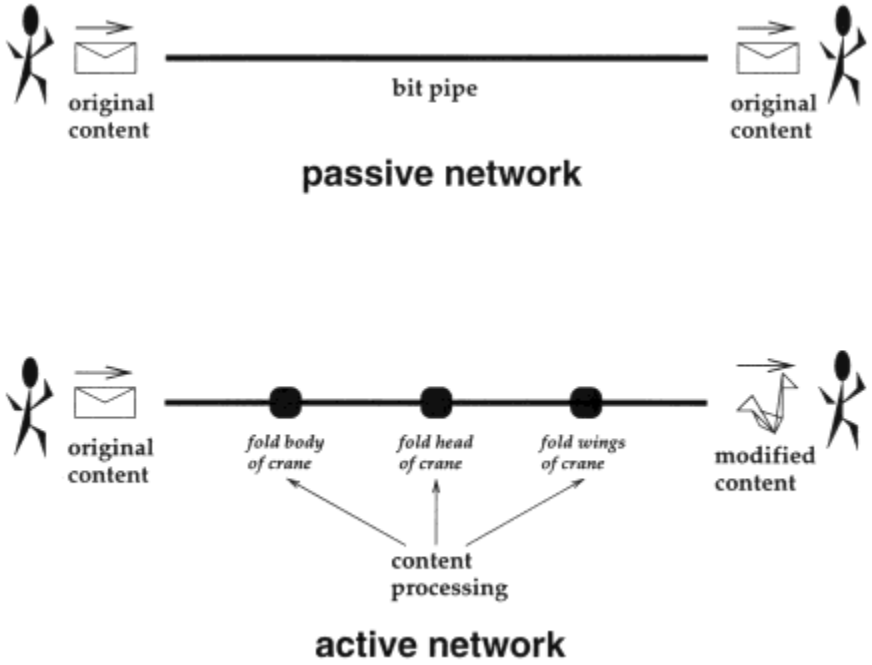

Figure 2 – Passive Vs. Active networks

So instead of standardizing on the low-level packet formats and transmission protocols as in passive networks, active networks present an abstract network programmable interface (or a network *API*) that allows low-level details to be programmed and customized. An

active network, essentially, provides a user-programmable interface to its nodes, enabling modification of network behavior as seen by the user.

Research in active networks is motivated by both technology "*push*" and user "*pull*". The technology "*push*" is the emergence of "active technologies", compiled and interpreted, supporting the encapsulation, transfer, interposition, and safe and efficient execution of program fragments. Today, active technologies are applied within individual end systems and above the end-to-end network layer; for example, to allow Web servers and clients to exchange program fragments (e.g. Java applets). The objective is to leverage and extend these technologies for use within the network - in ways that will fundamentally change today's model of what is in the network.

The "*pull*" comes from the ad hoc collection of firewalls, Web proxies, multicast routers, mobile proxies, video gateways, etc. that perform user-driven computation at nodes within the network. Despite architectural injunctions against them, these nodes are flourishing, suggesting user and management demand for their services.

The remainder of this paper is organized as follows. Section 2 describes *DARPA*'s architectural framework for active networks. The code in "smart" packets is distributed through an active network either in-band or out-of-band. These two approaches will be discussed in Section 2. Then in section 3, various approaches to programming packets in an active network are discussed. Following this, section 4 presents overviews on the various approaches for service composition and code distribution in an active network. Next, in section 5, a window into current research efforts is presented by reviewing the work of various institutions. This review sets the stage for section 4, which presents the numerous issues and challenges that make the realization of active networking a non-trivial task. Section 5 provides a brief overview of enabling technologies which may contribute to active network development. Section 6 describes some applications to demonstrate the flexibility and usefulness of active networks over passive networks. Finally, the paper is concluded in section 7.

2. A General Architecture Framework

In this section we present an overview of the architecture developed in the *DARPA* active networks program [7]. 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 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 *DARPA* architectural framework serves as a reference model for the research efforts. The intent of this model is:

- To lay out the guidelines and objectives for defining the major components and interfaces which make up an active node, and
- To allow various possible solutions/approaches towards the construction of an active node by describing a generic architecture that contains "functional slots" to be filled by the research groups.

DARPA defines an active network as a set of active nodes connected by a variety of network technologies. Each active node runs an operating system (*NodeOS*), a security enforcement engine, and one or more Execution Environments (*EE*). The composition of an active node is depicted in Figure 3.

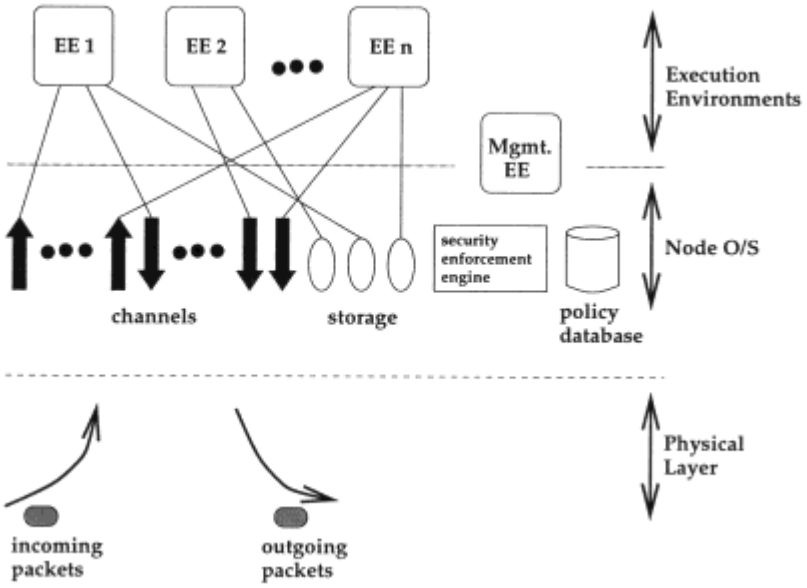

Figure 3 – Components of *DARPA*'s Active Node

2.1 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. All user access to node resources (including transmission bandwidth) is provided through an *EE*. An *EE* provides user access to node resources by exporting an *API* through which users can program. Examples of an *API* may be an extended *Java Virtual Machine* (*JVM*) or an enhanced sockets interface or a secured module-loading interface for adding extensions. Effectively, an *EE* resembles a virtual machine that users can control by sending it the appropriate coded instructions in packets. Interpretation of these packets generally causes the state of the active node to be updated to reflect the network behavior desired by an end-user.

Multiple *EE*s can exist on an active node[1]. The node architecture is explicitly designed to support multiple network *API*s 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 *API*s, but also for backward compatibility: IPv4 or IPv6 functionality can be provided as simply another network *API*. Any methods of dynamically downloading and installing a new EE should only be accessible to node administrators via a management *EE* as depicted in Figure 3.

2.2 Node Operating System (NodeOS)

The *NodeOS* provides the basic functions from which *EE*s build the abstractions that make up net *API*s. It manages the resources of the active node and mediates the demand for those resources, including transmission, computing and storage. The *NodeOS* isolates *EE*s from details of resource management and the existence of other *EE*s. The *EE*s, in

[1] It is expected that the number of different *EE*s supported by a single node at any one time will be small due to the nontrivial task of developing an *EE*.

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 3) authorizes the principal to receive the requested service or perform the requested operation. *EE*s 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 *EE*s 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). 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.

2.3 Active Network Encapsulation Protocol (ANEP)

So far it has not been specified how users can have their packets routed to a particular *EE* at a node. The Active Network Encapsulation Protocol [10] 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*. *EE*s 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.

2.4 Design Objectives

The architecture meets five main objectives [8]:

- Minimize the standardized protocols required to develop and implement end-to-end services. By reducing the amount of global agreement needed, this objective serves both research and commercial interests. Designers can experiment with a "five" network because the network is flexible enough to accommodate their needs during its operation. On the standard foundation, a variety of communication services can be established within each *EE*.
- Maximize flexibility in services supported. An element of this is to support different services simultaneously. The network-element resources the *NodeOS* controls are those universally needed by *EE*s; in this sense the architecture resembles a kernel architecture, in which *EE*s are multiprocessed. Because multiple *EE*s can run concurrently in each network node, research into different programming models can proceed in parallel. This is important for two reasons. First, although the programming community is gaining more insight into various models, they have yet

to select a winner. Second, designers can migrate to new versions of *EE*s while continuing to support the old ones.

- Let networks operated by different administrations be interconnected. Part of this is recognizing that trust relationships vary across administrations. It also means that security must be a fundamental consideration. In the general architecture, the *NodeOS* provides security services to *EE*s.
- Support scaling in both size and speed. This means considering fast-path processing, for packets that may not need active capabilities.
- Encompass current protocols, particularly the Internet Protocol (IP), as instances. In the general architecture, an IP stack can be viewed as an *EE*, albeit with a very simple *API* and role.

3. Code Distribution

User customization of network behavior requires that the user's program be distributed to the active nodes in a network. Several approaches to code distribution have been identified. In this section, we distinguish two approaches: discrete and integrated, depending on whether programs and data are carried discretely (i.e. within separate messages) or in an integrated fashion [5,6].

3.1. A Discrete (Out-of-band) Approach

The processing of messages may be architecturally separated from the business of injecting programs into the node, with a separate, auxiliary, out-of-band mechanism. Users send their packets through such a node much the way they do today. When a packet arrives, its header is examined and a program is dispatched to operate on its contents. The program actively processes the packet, possibly changing its contents. A degree of customized computation is possible because the header of the message identifies which program should be run - so it is possible to arrange for different programs to be executed for different users or applications.

This approach is preferable when the programs are large compared to the packet size. It also maintains modularization between user data and program, enabling better control by network administrators over what programs are loaded into the nodes. For example,

program loading could be restricted to a router's operator who is furnished with a "back door" through which they can dynamically load code. This "back door" would at minimum authenticate the operator and might also perform extensive checks on the code that is being loaded. Note that allowing operators to dynamically load code into their routers would be useful for router extensibility purposes, even if the programs do not perform application- or user-specific computations.

3.2. Capsules - An Integrated (In-band) Approach

A more extreme view of active networks is one in which every message is a program "*Capsule*". A *capsule* is a miniature program (of at least one instruction) that may include embedded data and is executed at each router/switch the message traverses. When a capsule arrives at an active node, its contents are evaluated, in much the same way that a PostScript printer interprets the contents of each file that is sent to it.

Bits arriving to the active node, on incoming links, are processed by a mechanism that identifies capsule boundaries, possibly using the framing mechanisms provided by traditional link layer protocols. The capsule's contents are dispatched to the selected execution environment where they can safely be executed. The execution of a capsule may result in the scheduling of zero or more capsules for transmission on the outgoing links and/or changes to the non-transient state of the node.

Of course, one can employ an approach which is a hybrid of the two approaches described above. For example, it is plausible to have an out-of-band loading of programs into a node and also have packets contain code fragments which might supplement program execution.

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. Some approaches to service composition are as follows:

- **Choice from a set of options**

  In this case, the network *API* supports specification of a scalar argument that selects a predefined 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 discussed in the next section, 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* and the *Netscript* projects discussed in the next section, with the languages *PLAN* and *Netscript*, respectively.

- **Event-based framework**

  In this approach, a service is constructed by binding code modules to specific events. A user selects an underlying program from those that are offered by a node. The program offers a basic-level service such as forwarding but it also contains "slots" into which users can inject customized code. Each "slot" is associated with a specific execution point in the underlying program. The Language-Independent Active Network Environment (*LIANE*) composition model, described in next section, is an example of such event-based composition.

5. Current Research

Work on active networks is underway at a number of sites which are independently studying: capsule and programmable switch architectures; enabling technologies; specification techniques; end system issues; and applications, including network, mobility, and congestion management. In this section, a window into the current research efforts is presented by reviewing the work of various institutions.

5.1 ANTS (MIT)

ANTS [11-13] is a protocol architecture that defines a communication model from which new protocols can be developed independently. Each node in an *ANTS*-based network

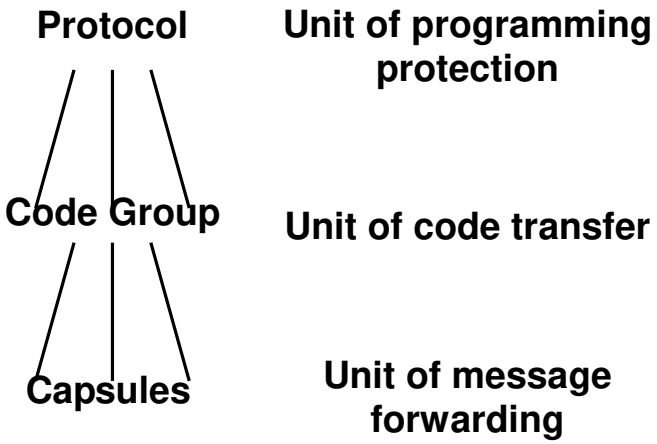

Figure 4 – Capsule composition hierarchy

executes the *ANTS* runtime environment based on Java. There are three key components to the *ANTS* architecture:

1. **Capsules:** A capsule is the unit of message forwarding. It plays the role of traditional packets. Each capsule contains the identifier of a forwarding routine to use at an active node. All forwarding routines belonging to related capsule types form a *code group* which is the unit of code transfer from node to node. In turn, related code groups form a *protocol* which is the unit of protection seen by an active node, i.e., capsules of one protocol may not access information and state of other protocols, nor can these capsules create new capsules belonging to other protocols. The relationships between these entities is illustrated in Figure 5.

   Capsules within a protocol can communicate with each other through state that is shared at active nodes. For example, one capsule type can set up location information

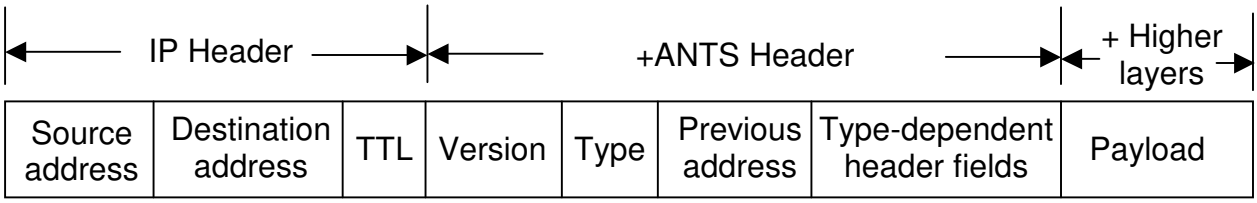

Figure 5 – Capsule format

at active nodes that other capsule types will use to reach a mobile host. They can also spawn other capsules that belong to the same protocol. Figure 6 shows the contents of each capsule. The capsule carries the regular IP header fields plus extension fields specific to *ANTS*. The field "*Type*" is an identifier that tells the associated protocol and forwarding routine. The rest of the capsule contains a shared header with fields common to all capsules, including version information and addresses used by the code

distribution system. Next are header fields that are used directly by the forwarding routines, and vary depending on the capsule type. Finally, the payload contains higher layer information that is carried across the network and exchanged between applications.

2. **Active Nodes:** The active nodes execute capsule-processing routines and maintain protocol state. Each node provides a set of primitives that are used by applications to construct forwarding routines. The choice of primitives is important because it determines the kinds of routines that applications can deploy. Ten primitives have been specified. These primitives roughly fall into three categories: environment calls (return information about the local node environment), storage calls, and control operations (used to route capsules toward other nodes or deliver them to local applications).

   The forwarding routines for a capsule are set at the sender's local node and remain fixed as the capsule traverses the network. Each routine is expected to run to completion locally within a short time, and their memory and bandwidth consumption are bounded by a scheme similar to IP's TTL (Time To Live). Additionally, capsule processing relies on Java sandboxing and bytecode verification to execute untrusted routines efficiently in a contained manner.

3. **Code Distribution Mechanism:** The code distribution system sends forwarding routines to nodes where they must be run. First, applications provide forwarding routines to their local node before sending the corresponding capsules. These routines, however, need to be distributed from the local node to the nodes which a capsule will visit. Code distribution is accomplished via an incremental, demand-loading mechanism which couples the transfer of code with the transfer of data as an in-band function. This approach limits the distribution of code to where it is needed. When a capsule arrives at a node and the required protocol code is not in the node's cache, a request message is sent by the node to the previous node visited by the capsule. The previous node replies with the needed routines, which are then integrated into the current node's cache and used to process the capsule. Effectively, code is propagated along the path that a capsule travels from sender to receiver. Eventually, as more and more capsules are processed, a region in the network develops where the same

processing is invoked repeatedly, thus, code transfer is no longer necessary. This mechanism is also adaptable to routing changes that may be caused by network congestion.

Currently, the *ANTS* architecture has been implemented as a prototype in Java to evaluate the suitability of this design for ease and flexibility in developing new services. The architecture has been used to study an auction service, a reliable multicast protocol, a web cashing service, and a multicast routing [11,13].

5.2 SwitchWare (University of Pennsylvania)

The *SwitchWare* architecture [14, 15] tries to achieve a balance between the flexibility of a programmable network infrastructure against the safety and security requirements inherent in sharing that infrastructure. The architecture consists of three layers, ordered from the highest, most limited in functionality, and least trusted to the lowest, and most trusted: active packets (*capsules*), active extensions, and secure active router infrastructure.

Active packets carry code and data from node to node. Packets are programmed in *PLAN* (Programming Language for Active Networks) [16-20], a simple strongly-typed, script-like language that supports very simple data and control structures. This simplicity makes programs easy to compile or interpret. *PLAN* programs are resource-bounded and can be statically type-checked before they are injected into the network, to ensure program correctness. A packet's *PLAN* code cannot modify or leave state at the router. The limited functionality in *PLAN* is intended so that no authentication is required to achieve lightweight execution. This decision is deliberate to prevent implementation of arbitrary protocols.

The active extensions layer extends the capabilities of active packets by providing services to access node facilities such as cache storage and routing tables. Extensions are loaded onto routers and execute entirely on a particular node. Loading can be done statically (as base functionality in a node) or dynamically (on-demand when an active packet requests a service). Because extensions can access the resources in a node, the loading process is subject to heavier-weight security checks such as cryptographic

signatures or proof carrying code (PCC). To study the active extensions layer, a prototype, *Active Bridge* [21], has been constructed.

The lowest layer in the *SwitchWare* architecture is the secure active router infrastructure. The goal of this layer is to provide a secure foundation upon which the other two layers build. This infrastructure is embodied in *SANE* (Secure Active Network Environment) [22].

SANE performs two core functions:

1. Ensure that an active network element is brought into its operational state via the *AEGIS* secure bootstrap architecture. The boot process is layered, starting from the lowest layer (initialization firmware) up to the node operating-system layer. Each successive layer is checked for integrity with a digital signature before control is passed to it.
2. Ensure that security is maintained when the active network element is operational. This involves node-to-node authentication and provision of a restricted environment for the execution of programs.

5.3 NetScript (Columbia University)

The *NetScript* architecture [23] uses an overlay to achieve both network programmability and interoperability. The network is viewed as a collection of Virtual Network Engines (VNE) interconnected by Virtual Links (VL), which, in its entirety, forms the *NetScript* Virtual Network (NVN). Each VL is composed of a collection of nodes and links with one or more VNEs running at a node. Mobile agents are used to transfer programs to the intermediate nodes. *NetScript* programs can use a library of primitives (provided by the VNE) to access node resources, in addition to scheduling and transmission of packets over VLs.

A *NetScript* program operates on one or more streams of packets. It is based on a dataflow computational model where the computation is organized into a collection of concurrent threads that are distributed at the VNEs. One can imagine the computation as a pipeline or assembly line where each "stage" (representing a thread running at a VNE) performs its assigned computation on a packet before delivering this packet to the next "stage". Each "stage", called a box in the *NetScript* language, is connected through its

input and output ports to other boxes. The boxes form a reactive system in which the arrival of data triggers computation within each box. Otherwise, the box sleeps until data arrives to trigger it.

Dynamic composition of protocols is achieved by dispatching boxes to nodes in the NVN and connecting them to the boxes already residing there. This feature enables a user or network administrator to extend the network with various functions.

The *NetScript* language [24] is based on the "box" notion and as such, is the central construct. The language provides primitives for declaring a box, its input and output ports, and a means to create compound boxes by connecting ports of simpler boxes together.

5.4 CANES & LIANE (Georgia Tech)

The *CANES* project [25 - 27] focuses on applying the active network paradigm to solving the problem of network congestion. The motivation for this work lies in the fact that congestion is an intranetwork event and is potentially far removed from the application. Since active networks enable the programming of intermediate nodes between two endpoints, congestion is a good candidate to study the benefits of network programmability.

The *CANES* architecture defines a finite set of functions that can be executed at each node. Because only specific network-supported functions can be invoked, the researchers claim that security is not an issue. A function is selected via an Active Processing Function Identifier (APFI) that is carried in packet control information. This control information also includes a set of labels called an Association Descriptor (AD) that select the state information to use (and possibly update) for the packet computation. The AD effectively groups packets into a protocol, since packets matching the same set of labels are subjected to the same treatment.

The node architecture consists of a routing core that connects node inputs, outputs, and active processing elements. Data entering the node is either sent directly to the node output if no congestion if; detected or to the appropriate active process otherwise. Due to the finite set of operations in CANES, it can be heavily optimized by implementing the

active processing elements in specialized hardware as opposed to general purpose processors.

The *CANES* project applies the active paradigm to network congestion in order to detect congestion at the points or nodes where it occurs in the network. In turn, these nodes can react promptly to handle the congestion rather than incurring delay by waiting for the endpoints to notice the phenomenon.

LIANE (The Language-Independent Active Networking Environment) is an example of event-based service composition. *LIANE* attempts to construct dynamic, trustworthy services from reliable base services. It is not tied to a particular language, although its prototype implementation is in *C++*, but relies on a reduced programming model that gives a predecided amount of flexibility to the dynamic environment. The advantage is that designers can limit the required security analysis.

Service 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 privileged 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.

5.5 Smart Packets (BBN Technologies)

The Smart Packets project [28] focuses on applying active networks to network management and monitoring. Like an SNMP packet, each smart packet can request and retrieve data from a Management Information Base (MIB). However, unlike SNMP, each smart packet can perform complex operations on this MIB data at the source site, as opposed to only at the requesting host. This capability has the advantages of tailoring the data to the interests of the management center as well as shortening the monitoring and control loop of information exchange.

Each node (including end hosts) in a Smart Packets active network contains an Active Network Encapsulation Protocol (ANEP) daemon process that is responsible for the injection/reception of smart packets and the operation of a virtual machine to execute programs. Computations may not leave state at the routers since state persistence is expensive and can lead to consistency problems. Additionally, packet fragmentation is not allowed, i.e., all packets are self-contained, and programs must be under 1KB in length. For security purposes, node resources are not accessible, instead, implementors must rely on the MIB to provide this information.

A smart packet contains either a program, data resulting from a computation, or informational/error messages. A context field in the smart-packet header identifies the originator of the packet, and a sequence number is used to differentiate the messages belonging to the same flow. Interoperability with the existing IP infrastructure is achieved by encapsulating each smart packet in ANEP, which in turn is encapsulated in IPv4, IPv6, or UDP. However, current IP forwarding semantics do not have a notion of a datagram whose contents is processed at intermediate nodes. To enable IP routers to look at payload, the "Router Alert" IP option is used to tell routers to examine the contents of a datagrarn.

Smart packet programs are written in one of two languages:

1. ***Sprocket*** is a C-like language but with security-threatening constructs, such as pointers, removed. The language also has built-in types for MIB access.

2. ***Spanner*** is a CISC assembly language designed to yield very small encoded programs. Sprocket programs are compiled into Spanner programs which are, in turn, assembled into a compact, machine-independent binary encoding.

Since the Smart Packets architecture is targeted towards network management and monitoring, programs are not expected to exceed 1KB due to the claim that network management functions do not take up a lot of program space.

Security at a node is achieved by having each packet carry an authenticator that identifies the entity which originated the packet. The authenticator is used to first check data integrity via a cryptographic hash over the packet's non-mutable fields, i.e., the packet header. If the verification fails, an error message packet is sent back to the originator, otherwise, authorization is performed by checking against an Access Control List (ACL) to determine what limits a packet program can have, such as access to MIB "set" functions or forwarding packets along a non-default path. If authorization fails, the packet is directed towards a restricted, resource-limited environment for execution.

5.6 Liquid software (University of Arizona)

Liquid software [29] is an entire infrastructure for dynamically moving functionality around in a network. The name indicates that the software easily flows from machine to machine. *Liquid* uses Java as its API. Java's machine-independent bytecode can be safely executed on different machines. However, the important problem that remains is how to execute this mobile code efficiently. The current practice is for each machine to interpret the bytecode, but this is not an efficient approach. Supporting mobile code on network nodes requires that the bytecode be compiled, but this puts the compilation on the critical path. To solve this problem, *Liquid* software contains a set of *Java* tools that offset the speed problem in two ways: First, it uses *Java-to-C* translators in conjunction with *C* compilers, thus avoiding the need to interpret byte code at runtime. Second, it uses compilers that themselves execute quickly and can be run at the point of execution (gigabit compilers). This approach maintains the usability and flexibility of *Java* while improving performance.

5.7 Protocol Boosters (Bellcore)

Protocol Boosters [30, 31] represent a design methodology for network protocols which is centered on the use of transparent, composable protocol functions that are injected into protocol stacks at end hosts and intermediate nodes. The basis for this work lies in the x-Kernel [31], which uses protocol graphs to represent the interactions between protocol elements that carry out functions for a protocol. These graphs are implemented as executable modules that cooperate via messages and/or shared state. Protocol boosters are inserted into the execution path that is followed by a group of packets handled by a protocol. This feature enables boosters to adapt a protocol to a specific application requirement or network environment.

A protocol booster is both parasitic and transparent. The parasitic property means that a booster uses whatever functionality and information is available from other protocols or boosters, but, by itself, it serves no useful purpose. The transparent property describes the booster's ability to add, delete, or delay messages of a protocol without originating, terminating, or converting that protocol's syntax and semantics. Transparency also refers to the user's/network administrator's ability to dynamically add and delete boosters anywhere in the network. Elimination of a booster does not terminate end-to-end communication but the end user might observe degradation in network performance as a result.

Protocol boosters are implemented as kernel-level modules. Policies associated with boosters determine the conditions under which booster functions are invoked. These policies may be based on, but certainly not limited to, observed network behavior, packet source and destination addresses, or time of day.

Two practical examples of protocol boosters are an encryption booster and a compression booster. The encryption booster can transparently increase the security of the network services provided in the case of sensitive data travelling across an insecure subnet. By modifying the protocol stack at the boundary routers, packets can be encrypted (or boosted) upon entrance into the subnet and decrypted (or deboosted) upon leaving. Similarly, the compression booster can transparently reduce the amount of bandwidth required without any added user-level complexity. Policies can be

programmed with the booster to detect the proper conditions under which it should be invoked, e.g., network congestion or transmission to a subnet with limited bandwidth.

Currently, the project is implementing protocol-booster support in the IP layer of FreeBSD. To identify a packet for boosting, the Type of Service (TOS) field in the IP header is used to store a booster id. Multiple boosters are supported by using a demultiplexing algorithm that examines a packet's IP address and based on a table lookup, either invokes a booster (if a match is found) or reinserts the packet into the normal execution path. While the current implementation only supports boosting at the IP layer, the ultimate goal is to provide a general environment to allow booster placement.

5.8 Architectural Features Summary

Table 1 summarizes features in each of the active network architectures surveyed. Entries with the value "-" means that the particular feature is not mentioned in the corresponding literature.

Project	Architecture	Network API	Security Mechanism	State Persistence	Code Distribution Mechanism	Service Composition
ANTS	Per-packet execution	Java	TTL-like schem; coded function id; Java sandbox model	Yes	in-band, on demand	Turing
SwitchWare	Layered	PLAN	Programming language centric; authenticated services	Yes	in-band using active packets; out-of-band for active extensions	Special purpose
NetScript	Overlay	NetScript	Coded function id	Yes	in-band	Special purpose

CANES	Per packet	–	–	Yes	Out-of-band	Predefined set of components
Smart Packets	Per-packet execution	Sprocket (C-like) and Spanner (CISC assembly)	Per-packet authenticator; Access Control list	No	In-band	Turing (though limited to network management and monitoring
Protocol Boosters	Per-packet execution	–	–	–	Out-of-band	Event-based

Table 1 – Feature summary of active-network architecture

6. Challenges For Deployment

As DARPA's active networks program moves into its mid-life, the work by various research groups will be shaped by the challenges that are inherent in active networks. Two key challenges are security and performance. While it is obvious that security mechanisms are needed to prevent invalid use of or malicious attacks on a node's resources, there is a tradeoff between how much security one can provide and the efficiency needed for packet processing. These two challenges are described in addition to others that researchers must tackle towards the realization of active networks.

6.1 Security

Like traditional networks, active network are concerned with the authenticity, integrity, and confidentiality of the data going through the network. However, traditional networks are concerned only with possible damage to user data and end nodes. Active networks share these concerns but must also consider possible damage as the active packet moves into each node and EE. Active nodes could be harmed by active code, either because the

code modifies the node's state or because it drains resources (essentially launching a denial-of-service attack). Thus, enforcing protections at end nodes is not sufficient for active networks. Securing an active network means that protection mechanisms must move into each node and each EE. Protecting the network as a whole is only possible by building a common protection mechanism into the design of individual nodes and EEs. This could be achieved by the following sequence of tasks:

1. **Validation**. Ensuring the program is indeed the correct program. This is commonly achieved by encoding the contents of a packet using a cryptographic hashing algorithm such as the MD5 message digest [32], and carrying this encoding in the packet header. Any packet that fails the validation check is subjected to either dropping or some default forwarding behavior.
2. **Authorization.** Ensuring the program comes from an authorized user. Once a program is validated, an Access Control List (ACL) is consulted. Packets that fail the authorization are handled either by some node-specific or user-specified default processing.
3. **Execution.** Based on information from the authorization phase, the run-time environment enforces the resource usage and access limits on the program's execution. These limits include the maximum amount of time a program can spend running at a node and the amount of memory that can be accessed.
4. **Fault Detection.** Any program faults that may occur during execution are seen as attacks, and must be caught and handled efficiently to prevent harm to the correct operation of the node. Handling of the fault/attack should not disrupt services to existing flows.

A strong security architecture should address all the concerns listed above, but still be as lightweight as possible. However, providing an optimal solution is a very complex problem.

With active networks, security protections travel with the packet so that appropriate protections can be chosen dynamically at nodes to suit the environment through which the packet passes. Different users and organizations have different security requirements and as a result, security systems need to support dynamic interoperable security policies

to enforce proper security measures and access control for packets. This necessity gives rise to several challenges.

- How can these security policies be defined for the different uses of an active network, i.e., different permission levels at a node?
- The ability to negotiate a common set of security services between two or more administrative domains is required. How can differing policies be reconciled?
- How can the security architecture scale to handle a growing population of users with different interests?
- Not only must protection be guaranteed within a node, but protection must also exist on a per-user basis, i.e., it must be possible to protect one user's packets from those of another user.

The requirements on a security mechanism are also dependent on the set of functions a node exports and the expressiveness of the language used to create active network programs. Functions that enable access to node resources and modification to node state are extremely dangerous, requiring tight control over their use. Furthermore, if an expressive language, such as Java, is used to program packets, a node must be able to detect program logic that would behave in a malicious way (e.g., an infinite loop) and prevent the program's execution as early as possible.

The ideal solution to security should be both lightweight and scalable. Most of the proposed solutions offer strong security measures but are costly in time, computation, or the number of messages needed to retrieve keys, especially if a public key infrastructure is in place. Even if the proposed security services are lightweight, this often comes at a cost of less flexibility in what a packet can do at a node. The point is that there is no one perfect, generic solution to the security problem; each solution requires the tradeoffs in one or more areas.

6.2 Performance

Recent developments in network technology seek to implement packet switching at Gigabit per second rates. This is motivated by the need to increase the throughput and speed of networks as the numbers of users and applications continue to grow. However, this trend may suffer a setback with the introduction of active networks. The idea of

moving computation into the nodes suggests that active networks may in fact reduce network performance. Computation and the required security services drastically increase the per-packet processing time. This extra burden complicates the techniques developed to minimize packet processing along an end-to-end path. The challenge that researchers face is how to translate the fast switching technology of passive networks into the active paradigm. What exactly would the translation entail in terms of program complexity, compiler/interpreter technology, execution environments, restrictions on node resources, and hardware?

The scalability problem also affects performance since, potentially, there may be thousands of user processes active at the same time at a node. Worse, the processes may need to access the same set of resources, requiring resource management to prioritize processing needs of applications. The determination of computational requirements for end-to-end services is difficult due to the differences in the underlying hardware architecture of nodes, especially when the nodes lie in different administrative domains.

Traditional network performance measures, such as throughput and round-trip time, are aimed at evaluating the performance of the network rather than the performance of the applications using it. However, network performance is not necessarily related to application performance. While it is true that packet processing in active networks will incur longer delays in packet transmission, applications may actually experience an improvement in performance resulting from active operations that delegate service-code processing and/or congestion handling to intermediate nodes. As a result, performance should be evaluated in terms of application-specific metrics such as the number of client requests serviced per second. Determining which performance metrics to use is dependent on the application being evaluated.

6.3 Interoperability

Current research in active networks offers a good variety of programming approaches in the implementation of router services (service decomposition). This leads to differences in the programming languages and the packet formats, allowing a user the flexibility to select and use the various features of each approach. However, this flexibility requires some means to reconcile their differences in order to provide compatibility. If two or

more administrative domains take different approaches, how will the routers handle the various kinds of packets passing through them? Does this mean that participating domains must agree to run each other's execution environments to enable compatibility? If so, how will this scale when there is no strict control on what approaches can be implemented at a router at any one time? This problem is further exacerbated by the discrete and integrated models presented in section 3. The difficulty here also lies in bridging these two models. Packets in the discrete model have their programs loaded out-of-band, as opposed to the integrated model where packets carry the programs. When a packet leaves a network implementing the discrete model and enters a network implementing the integrated model, how will the required code, pre-loaded at the routers, be moved into the packet? What is not clear at this moment is whether DARPA eventually plans to standardize an active packet format, programming language, and computational model, or to simply leave interoperability as an exercise for domain administrators. In addition to interoperability among programming approaches, there is also a need to define a common network API.

6.4 Backwards Compatibility

Because the Internet connects millions of nodes, this infrastructure will be a good deployment mechanism for active networks in the future. From a practical point of view, it is unreasonable to expect that every network domain will embrace the active paradigm of network communication. In light of this, backwards compatibility is essential to enable active packets to travel between passive and active domains and to be processed accordingly.

There are two approaches to achieving backwards compatibility [33]:

- **Encapsulation.** Active-packet programs are encapsulated into an ANEP packet that is, in turn, placed into an IP packet. This approach is simple and effective, especially when one or more passive IP networks separate two active networks. The difficulty that arises with encapsulation, however, is determining how to inform an active node to process the IP packet payload. Proposed solutions to the problem include modifying IP options in the header or using the Type Of Service (TOS) field (which would then lead into discussions on how this affects Differentiated Services).

- **Gateways.** Gateways sit at network boundaries and convert one protocol to another. The translation process can get quite complex if IP packets do not map well to the packet format required by an active network. There are two challenges that need to be addressed:
  1. Packets moving from a passive to an active network. If the packet is simply a passive IP packet, the gateway must translate it into a "non-functional" active packet. This could be achieved by placing an invalid function id in the active packet header so that each node can treat it with a default forwarding behavior. If the packet originated in an active network, the gateway must "reactivate" the packet after its "deactivation" in the passive network. This might be trivial if the originating and receiving active networks both share the same programming and security architectures (i.e., they are both under the same administration). However, if this is not the case, resolving the differences between the two active networks further complicates the translation process.
  2. Packets moving from an active to a passive network. This case is easier to deal with than its converse since IP encapsulation can simply be used to tunnel the active packet through the passive network.

Both approaches must also deal with fragmentation (when an active program is too large to fit into an IP packet) and the interoperability issues described earlier. For fragmentation, discrepancies between the Maximum Transmission Unit (MTU) sizes at different layers of a protocol must be resolved to effect proper delivery of active programs. Allowing fragmentation in an active network architecture introduces additional performance overhead, since a node must wait for the complete program before it can be executed. An efficient node might be able to execute the portion of code it receives while awaiting the rest, but then the question of program correctness and security resurfaces.

7. Applications of active networks

The task for active networking researchers is to justify the eventual migration towards the active paradigm. To achieve this, active networks must provide some "immediate" benefits over existing solutions to passive network applications. Functionality for active networking will not be added to end systems unless there is some benefit in doing so, and

switch manufacturers and network operators will not upgrade their switches to support active networking unless customer demand exists. In light of this, various research groups have proposed some applications to demonstrate the merits of active networks for controlling the behavior of packets inside the network.

7.1 Reliable Multicast

Multicasting provides an efficient way of disseminating data from a sender to a group of receivers. Instead of sending a separate copy of the data to each individual receiver, the sender just sends a single copy to all receivers, thus, potentially reducing communication costs. A multicast tree is used to determine the delivery paths from the sender (root of the tree) to the receivers (the leaves). Data generated by the sender flows downstream in the tree, traversing each tree edge exactly once. Receivers will either send an ACK upstream towards the sender if data reception is successful, or a NACK if no data is received within a timeout period. However, this reliability in the multicast protocol has some problems:

- The number of receivers could be high, which causes the ACK and NACK implosion problem where the number of ACKS/NACKS sent upstream is too large for the sender to handle.
- Heterogeneity in receivers means that each one will have different loss rates based on differing network connections and processing capabilities. Retransmissions are costly in bandwidth consumption, since packets are resent to all receivers in the group.
- Dynamic membership changes make it hard to designate some router to serve as a proxy in order to reduce consumption of transmission bandwidth. What this means is that rather than having the sender be responsible for all retransmissions, routers at key locations (or interior nodes) in the multicast tree can cache sender packets and retransmit them to the receivers that fall under each router's jurisdiction. However, dynamic membership changes may cause the multicast tree to be restructured so that a router selected as a proxy at one time may no longer be effective at another (e.g., the router becomes the leaf of the multicast tree). These changes must be

continuously detected by the multicast protocol so that proxy designations will also change with the memberships.

Existing work on reliable multicast protocols [34 - 36] offers only partial solutions to the above problems.

Active-network technology has some potential to address the difficulties in deploying multicast [37, 38]. Intermediate, active routers in the multicast tree can merge ACKS or NACKS travelling upstream to prevent implosion at the sender. Packet caching at active nodes enable retransmission to only the subset of receivers that experience loss, and not to the entire multicast group. These nodes can be placed strategically so they are local to different groups of receivers and/or placed where bandwidth becomes scarce, such as at the boundary between wired and wireless links. This implies that not all nodes need to cache packets, but for those that do, each packet has an associated TTL to help a node determine which parts of cache storage it can recover. Retransmissions by intermediate active routers can significantly reduce recovery latency for topologically distant receivers. This helps to distribute load for retransmission to the routers in a multicast tree, which, in turn, protects the sender and bottleneck links from retransmission requests and repair traffic. The fusion of ACKS and NACKS at active nodes also contributes to lowered bandwidth consumption.

7.2 Network Congestion

Network congestion is a problem that is unlikely to disappear. Current congestion-control mechanisms are employed at endpoints, and use network feedback to control transmission rate and to invoke loss-handling routines. While this has worked well for the most part, there are still some well-known challenges:

1. The time interval required for the sender to detect congestion, adapt to bring packet losses under control, and have the controlled-loss data propagate to the receiver can be long. During this interval, the receiver experiences uncontrolled packet loss, resulting in a reduction in quality of service. This problem is further exacerbated if the end-to-end delay and network bandwidth increase, since the longer the delay, the longer it will take the source to detect congestion, and the larger the bandwidth,

the larger the amount of data that will be sent into the network before congestion control is applied.

2. Sender detection of an increase in available bandwidth. Currently, packet loss is the only mechanism for determining available bandwidth. For example, in continuous media applications, a sender adapts to congestion by changing to a lossier encoding. The sender must also detect the easing of congestion by periodically reducing compression and waiting for feedback from the receiver. However, this would cause the receiver to experience periods of uncontrolled loss (from the reduced compression) in the case of long-lived congestion.

To enhance mechanisms for reducing network congestion, active networks [26, 39] can be employed to extend congestion detection and congestion response into the intermediate nodes. This reduces the feedback delay since congestion is detected where it occurs, and changes in congestion state are propagated to the endpoints which, in turn, take further action to alleviate the congestion. As a result, congestion is relieved sooner. Common techniques for controlling loss in the face of congestion can be transplanted into the active paradigm as node programs. These techniques include buffering and rate control, selective dropping of packets, and media transformation.

7.3 Wireless Services

Wireless networks are characterized by low bandwidth and lossy links subject to interference. Transmission over a wireless link requires the data to be limited to a rate that matches the bandwidth of the link. Otherwise, packet loss can occur as a result of queue overflow. Additionally, wireless links have changing bit error rates (BER) and are prone to sporadic connectivity breakdowns.

Active networking is applied in both [30] and [40] to address the above problems. Adaptive Forward Error Correction (FEC) is utilized to demonstrate how active networks can reduce the effective packet loss rate on wireless links. Active nodes at the wired-wireless boundary continuously monitor the BER of their links. Based on the current BER of a link, an appropriate number of FEC bits are added to the packet before transmitting it to the wireless side. To counter intermittent connectivity, packet caching is performed at the active nodes. For timing-sensitive packets, a maximum allowable delay

is maintained and checked against the current time. If the delay time is exceeded, the packet is discarded. Filtering is also applied in [40] to tackle the bandwidth problem.

8. Conclusions

Active networks present an opportunity to change the current network infrastructure from a closed transport system to an open computational environment. By allowing the injection of user programs into network nodes, an active network offers the ability to test and deploy new services quickly. This ability will lead to a user-driven innovation process in which the availability of new services will be dependent on their acceptance in the marketplace, and not be delayed by long standardization activities. However, this is not a universal truth; wrongly applied, active networking could lead to chaotic non-interoperability and severe performance degradation of the network. With various research groups each developing their own approaches to such areas as code distribution, execution environment, node security, and programming language, how will all the differences between the work be resolved? As it stands now, the DARPA architecture seems too general to provide a solid foundation for global agreement and implementation of active networks. There is no clear strategy to develop and deploy the active paradigm into the existing infrastructure. Perhaps this is due to development still largely being in its infant stage, resulting in a lack of clarity on what the role of active networks is envisioned to be in the future. Ultimately, a fundamental and tightly-defined architecture of the network needs to be agreed upon by all segments of the industry. Hence, the long and painful IETF standardization process that active networking seeks to bypass will continue to be necessary until a consensus can be reached.

The flexibility and programmability in active networks is attractive since the current infrastructure is riddled with a diverse range of applications that require different services. However, when one stands back and assesses the potential of an active network, it can be difficult to swallow the idea of nodes running arbitrary user programs that provide the desired services. This over-glorified picture offered by some researchers is misleading and presents many opportunities for nay-sayers to question the feasibility and practicality of active networks, particularly, in the areas of security and performance. From a practical point of view, active networks can provide the flexibility that is needed

in the current infrastructure, but researchers must understand which classes of applications are best suited for the active paradigm and what restrictions are needed in programming environments to balance security and performance. Computation at nodes will no doubt prevent active networks from achieving the network performance of traditional networks, but one must think of gains in application performance when arguing for the side of active networking. Furthermore, the question of who is responsible for programming the nodes requires clarification. Will new services be deployed by the users themselves or by third-party service developers? It seems more practical to employ third-party provisioning of services, since they can be envisioned as having contracts with network domains, as opposed to per-user programming that would require tight control of activity over a potentially large user population. Consequently, the active-networking paradigm can be realized as a restricted programming environment (to balance security and performance) with a modular set of functions for service composition that is developed by third-party programmers.

The challenge for active networking in the next few years is to provide useful and feasible solutions to the many problems inherent in this new paradigm. These include security, performance, interoperability, and robustness. Active-network research must demonstrate the merits of this new paradigm to justify the continuation of research activities. Further to this, a more difficult task is convincing service providers of the advantages of moving to an active network. How can one market the active approach to substantiate a potential growth in the revenue and customer base of a service provider? Perhaps by offering better solutions to new and existing application areas such as multicast and network congestion, these benefits will help shape the future direction.

References

1. E. Braden, L. Zhang, S. Berson, S. Herzog, and S. Jamin, "Resource ReSerVation Protocol (RSVP) - Version 1 Functional Specification", RFC 2205, *Internet Engineering Task Force*, September 1997.
   http://info.internet.isi.edu:80/in-notes/rfc/fies/rfc2205.txt
2. C. Perkins, "IP Mobility Support", RFC 2002, Internet Engineering Task Force, October 1996.
   http://www.faqs.org/rfcs/rfc2002.html
3. S. Deering and R. Hinden, "Internet Protocol, Version 6 (IPv6) Specification", RFC 1883, Internet Engineering Task Force, December 1995.
   http://www.faqs.org/rfcs/rfcl883.html
4. S. Deering, "Host Extensions for IP Multicasting", RFC 1112, *Internet Engineering Task Force*, August 1989.
   http://www.faqs.org/rfcs/rfc1112.html
5. David L. Tennenhouse, Jonathan M. Smith, W. David Sincoskie, David J. Wetherall, and Gary J. Minden, "A Survey of Active Network Research", IEEE Communications, Vol. 35, pp. 80-86, January 1997.
   http://www.tns.ics.mit.edu/publications/ieeecomms97.html
6. David L. Tennenhouse and, David J. Wetherall, "Towards an Active Network Architecture", *International Conference on Multimedia Computing and Networking*, San Jose, CA, January 1996.
   http://www.tns.ics.mit.edu/publications/icmcn96.html
7. Active Networks Working Group, "Architectural framework for Active Networks", Version 0.9, *Technical Report*, DARPA-ITO, August 1998.
   http://www.cc.gatech.edu/projects/canes/arch/arch-0-9.ps
8. Jonathan M. Smith, Kenneth L. Calvert, Sandra L. Murphy, Hilarie K. Orman and Larry L. Peterson, "Activating Networks: A Progress Report", IEEE Computer, Vol. 3, pp. 32-41, April 1999.

9. Kenneth L. Calvert, Samrat Bhattacharjee, Ellen Zegura, and James Sterbenz, "Directions in Active networks", *IEEE Communications Magazine*, Vol. 36(10), pp. 72-78, October 1998.

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

11. David Wetherall, John V. Guttag, and David L. Tennenhouse, "ANTS: A Toolkit for Building and Dynamically Deploying Network Protocols", IEEE OPENARCH '98, April 1998, San Francisco, USA. http://www.Ans.Ics.mit.edu/publications/openarch98.html

12. David Wetherall, John V. Guttag, and David L. Tennenhouse, "ANTS: Network Services Without The Red Tape", IEEE Computer, Vol. 4, pp. 42-48, April 1999.

13. David Wetherall, Ulana Legedza, and John Guttag, "Introducing New Internet Services: Why and How", IEEE Network, Vol. 12(3), pp. 12-19, May 1998. http://www.sds.Ics.mit.edu/publications/network98.html

14. D. Scott Alexander, William A. Arbaugh, Michael W. Hicks, Pankaj Kakkar, Angelos D. Keromytis, Jonathan T. Moore, Carl A. Gunter, Scott M. Nettles, and Jonathan M. Smith, "The SwitchWare Active Network Architecture", *IEEE Network,* Vol. 12(3), pp. 29-36, May 1998. http://www.cis.upenn.edu/~switchware/papers/switchware.ps

15. D. Scott Alexander, Michael W. Hicks, Pankaj Kakkar, Angelos D. Keromytis, Marianne Shaw, Jonathan T. Moore, Carl A. Gunter, Trevor Jim, Scott M. Nettles, and Jonathan M. Smith, "The SwitchWare Active Network Implementation", *International Conference on Functional Programming (ICFP) '98*, 1998, Baltimore, USA. http://www.cis.upenn.edu/~switchware/papers/ml.ps

16. Michael Hicks, Jonathan T. Moore, Pankaj Kakkar, Carl A. Gunter, and Scott M. Nettles, "PLAN: A Packet Language for Active Networks", *International conference on Functional Programming Languages (ICFP) '98*, pp.86-93, 1998, Baltimore, USA.

http://www.cis.upenn.edu/~switchware/papers/plan.ps

17. Michael Hicks, Jonathan T. Moore, Pankaj Kakkar, Carl A. Gunter, and Scott M. Nettles, "Network Programming Using PLAN", *IPL workshop'98*.
http://www.cis.upenn.edu/~switchware/papers/progplan.ps
18. Michael Hicks, Jonathan T. Moore, and Scott M. Nettles, "Chunks in PLAN: Language Support for Programs As Packets", submitted to *IWAN'99*.
http://www.cis.upenn.edu/~switchware/papers/planchunks.ps
19. Michael Hicks, Jonathan T. Moore, D. Scott Alexander, Carl A. Gunter, and Scott M. Nettles, "PLANET: An Active Internetwork, to appear in *IEEE INFOCOM*, New York, 1999.
http://www.cis.upenn.edu/~switchware/papers/planet.ps
20. Michael Hicks, and Angelos D. Keromytis, "A Secure PLAN", submitted to *IWAN'99*.
http://www.cis.upenn.edu/~switchware/papers/secureplan.ps
21. D. Scott Alexander, Marianne Shaw, Scott M. Nettles and Jonathan M. Smith, "Active Bridging", Proceedings of the ACM SIGCOMM'97 Conference, Cannes, France, September 1997.
http://www.cis.upenn.edu/~switchware/papers/sigcom97.ps
22. D. Scott Alexander, William A. Arbaugh, Angelos D. Keromytis, and Jonathan M. Smith, "A Secure Active Network Environment Architecture: Realization in SwitchWare", *IEEE Network*, Vol. 12(3), pp. 37-45, May 1998.
http://www.cis.upenn.edu/~angelos/Papers/sane.ps.gz
23. Yechiam Yernini and Sushil da Silva, "Towards Programmable Networks", IFIP/IEEE International Workshop on Distributed Systems: Operations and Management, October 1996, L'Aquila, Italy.
http://www.cs.columbia.edu/~dasilva/content/netscript/pubs/dsom96.ps
24. Yechiam Yernini, Danilo Florissi, and Sushil da Silva, "Composing Active Services in NetScript", *DARPA Active Networks Workshop*, March, 1998.
http://www.cs.columbia.edu/~dasilva/pubs/ns-composition.pdf

25. Samrat Bhattacharjee, Kenneth L. Calvert, and Ellen W. Zegura, "Active Networking and The End-to-End Argument", *International Conference on Network Protocols '97*, October 1997, Atlanta, Georgia.
ftp://ftp.cc.gatech.edu/pub/people/bobby/an/publications/icnp97.ps.gz

26. Samrat Bhattacharjee, Kenneth L. Calvert, and Ellen W. Zegura, "On Active Networking and Congestion", *Technical Report*, Georgia Institute of Technology, February 1996.
http://www.cc.gatech.edu/projects/canes/papers/anandc.ps.gz

27. Samrat Bhattacharjee, Kenneth L. Calvert, and Ellen W. Zegura, "An Architecture for Active Networking", *International Conference on High Performance Networking*, April 1997, White Plains, NY, USA.
ftp://ftp.cc.gatech.edu/pub/people/bobby/an/publications/hpn97.ps.gz

28. Beverly Schwartz, Wenyi Zhou, Alden W. Jackson, W. Timothy Strayer, Dennis Rockwell, and Craig Partridge, "Smart Packets for Active Networks", Technical Report, BBN Technologies, January 1998.
http://www.net-tech.bbn.com/smtpkts/smart.ps.gz

29. John Hartman, Udi Manber, Larry Peterson, and Todd Proebsting, "Liquid Software: A New Paradigm for Networked Systems", *Technical Report*, University of Arizona, June 1996.
ftp://ftp.cs.arizona.edu/xkernel/Papers/tr96-11.ps

30. William S. Marcus, Ilija Hadzic, Anthony J. McAuley, and Jonathan M. Smith, "Protocol Boosters: Applying Programmability to Network Infrastructure", *IEEE Communications*, Vol. 36(10), pp. 79-83, October 1998.

31. A. Mallet, J.D. Chung, and J.M. Smith, "Operating System Support for Protocol Boosters", *Technical Report*, University of Pennsylvania.
http://carin.belicore.com:8000/boosters/ims.ps

32. R. Rivest and RSA Data Security, "The MD5 Message-Digest Algorithm", RFC 1321, Internet Engineering Task Force, April 1992.
http://www.faqs.org/rfcs/rfcl321.html.

33. David M. Murphy, “Building an Active Node on the Internet”, Master's Thesis, Massachusetts Institute of Technology, May 1997.
ftp://ftp.tns.Ics.mit.edu/pub/papers/MIT-LCS-TR-723.ps

34. Sally Floyd, Van Jacobson, Steven McCanne, Ching-Gung Liu, and Lixia Zhang, “A Reliable Multicast Framework for Light-weight Sessions and Applications Level Framing”, *ACM SIGCOMM ’95*, 25(4), pp. 342-356, October 1995.

35. Sanjoy Paul, Krishan K. Sabnani, John C.H. Lin, and Suratik Bhattacharyya, “Reliable Multicast Transport Protocol (RMTP)”, *IEEE Journal on Selected Areas in Communication*, Vol. 15(3), pp. 407-421, April 1997.

36. R. Yavatkar, J. Griffioen, and M. Sudan, “A Reliable Dissemination Protocol for Interactive Collaborative Applications”, *ACM Multimedia*, pp. 333-344, November 1995, San Francisco, USA.

37. Maria Calderon, Marifeli Sedano, Arturo Azcorra, and Cristian Alonso, “Active Network Support for Multicast Applications”, *IEEE Network*, Vol. 12(3), pp. 66-71, May/June 1998.

38. Li-wei H. Lehman, Stephen J. Garland, and David L. Tennenhouse, “Active Reliable Multicast”, *IEEE INFOCOM '98*, Vol. 1, pp. 581-589, March 1998, San Francisco, USA.
http://www.sds.ics.mit.edu/publications/postscript/infocom98-arm.ps

39. Theodore Faber, “ACC: Using Active Networking to Enhance Feedback Congestion Control Mechanisms”, *IEEE Network*, Vol. 12(3), pp. 61-65, May/June 1998.

40. Amit B. Kulkarni and Gary J. Minden, “Active Networking Services for Wired/Wireless Networks”, *Submitted to INFOCOM '99*, March 1999, New York, USA.
http://www.ittc.ukans.edu/-kulkarn/docs/infocom99.ps.gz