Building an Active Node on the Internet

by

David M. Murphy

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 1997

Author . . . .
Department of Electrical Engineering and Computer Science
May 27, 1997

Certified by . . . .
John V. Guttag
Professor
Thesis Supervisor

Certified by . . . .
David L. Tennenhouse
Senior Research Scientist
Thesis Supervisor

Accepted by . . . .
Arthur C. Smith
Chairman, Departmental Committee on Graduate Theses

Building an Active Node on the Internet

by

David M. Murphy

Submitted to the Department of Electrical Engineering and Computer Science
on May 27, 1997, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

An Active IP Network integrates two very different network programming models, an IP packet based model, and an Active Network capsule based model. This report shows how to integrate these two models into a single node, called an Active IP node, and how to integrate an Active IP node into an IP network. It also presents some preliminary ideas on the constraints network architects will face when building Active protocols for a heterogeneous network of Active and non-Active IP nodes.

By using a model of constant and variable processing, integrating the Active and IP architectures has lead to a clean and simple node design and implementation. Furthermore, mechanisms presented in this report, such as protected buffers, provide various safety constraints which aid in the integration.

Finally, this report presents some preliminary performance results which, when combined with the above characteristics, suggest that the Active IP platform will be appealing to researchers who wish to study application specific protocols for the Internet.

Thesis Supervisor: John V. Guttag
Title: Professor

Thesis Supervisor: David L. Tennenhouse
Title: Senior Research Scientist

Building an Active Node on the Internet

by

David M. Murphy

Submitted to the Department of Electrical Engineering and Computer Science
on May 27, 1997, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

An Active IP Network integrates two very different network programming models, an IP packet based model, and an Active Network capsule based model. This report shows how to integrate these two models into a single node, called an Active IP node, and how to integrate an Active IP node into an IP network. It also presents some preliminary ideas on the constraints network architects will face when building Active protocols for a heterogeneous network of Active and non-Active IP nodes.

By using a model of constant and variable processing, integrating the Active and IP architectures has lead to a clean and simple node design and implementation. Furthermore, mechanisms presented in this report, such as protected buffers, provide various safety constraints which aid in the integration.

Finally, this report presents some preliminary performance results which, when combined with the above characteristics, suggest that the Active IP platform will be appealing to researchers who wish to study application specific protocols for the Internet.

Thesis Supervisor: John V. Guttag
Title: Professor

Thesis Supervisor: David L. Tennenhouse
Title: Senior Research Scientist

Acknowledgments

I thank David Tennenhouse for being an excellent adviser on both my thesis and career. He has built a great research group, TNS, and a great research area, Active Networks. I am grateful that he has let me be part of both of them.

I thank John Guttag who has taught me a great deal about doing and presenting excellent research.

I thank David Wetherall, who has been a wonderful semi-adviser and colleague. I owe him a debt of gratitude for the help he has given me during the year.

I also thank everyone in the SDS group who has made working here such a wonderful experience. I also thank everyone for giving great comments on my practice thesis talks. In particular I thank Ulana Legedza who has supplied insightful comments on various chapters of this thesis.

Finally, I thank my family whose continual love and support throughout my whole life have kept me going.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

This report describes the design and implementation of a novel network node that integrates two very different network programming models, an IP packet based model, and an Active Network capsule based model. Both programming models have been integrated into the same implementation, and take advantage of the same type of packets and node resources. Importantly, users of the IP programming model do not need to be aware of the Active Network model. This allows the dual purpose node to be deployed seamlessly into an IP network. For users of the Active Network model, new types of constraints arise when designing Active protocols for a heterogeneous network of Active and non-Active nodes. This report presents some preliminary ideas on how these constraints might affect protocol development.

The node supports the next generation of Internet protocols, IPv6. The novel aspect of this node implementation, is that the node applies two types of processing to packets, constant and variable. The node applies constant processing, which represents a small subset of the IPv6 protocol, to all incoming packets. The variable processing represents an application-specific protocol requested by a packet. If a packet does not request a protocol then default IP processing is used.

In order to support variable processing, the node uses a capsule-based Active Network architecture implemented by David Wetherall. However, the constraint of co-existing with

IP led to a number of significant changes. Most notably, a new IPv6 option and payload has been defined to transport programming code and its associated state, the class structure of the capsules has been enhanced, and a protected buffer scheme has been implemented to ensure that integral fields within the IP header cannot be changed mid way through the network.

1.1 Active Networks

IP has become the *de facto* standard for network communication today. It achieves inter-operation among a variety of physical networks by serving as a virtual network layer on top of them. All the nodes on the Internet send packets amongst each other using IP. A wide range of applications use IP to transmit data amongst themselves. However, despite the diversity above and below the network layer, IP itself is rigid, slow to evolve and difficult to adopt to physical and application layer needs.

In 1990, Clark and Tennenhouse [11] described the need for an adaptable network. They claimed that future networks "must exhibit a significant degree of flexibility and be based on an architecture that admits a wide range of application demands and implementation strategies."

Today's network has a number of examples of the wide demand for customized network protocols. Unfortunately, the IP protocol provides little support for incorporating these needs into its architecture. For example, numerous network administrators have inserted firewalls, which filter packets, into IP routers that sit at the borders of their networks. These solutions tend to be implemented as *ad hoc* additions to networking code. Similarly the Multicast protocol has sat in beta-test limbo for years on the MBONE, thereby preventing commercial network administrators from reducing the bandwidth consumed by group broadcasts. Implementing multicast in the same *ad hoc* manner which developers have implemented firewalls would be problematic because multicast will only work if it is supported on a global scale.

In response to IP's inability to adapt to changing application needs, a new type of network architecture, an Active Network[31], has been proposed. This architecture allows applications to dynamically extend the functionality of the network by injecting customized protocols, also known as application specific protocols, into it. In this type of network, packets select the protocol by which they wish to be processed. Nodes within this network become execution environments that supply an Application Programming Interface (API) to the protocols delivered to them.

There are two major approaches towards transporting code in an Active Network, the out-of-band approach and the in-band approach. In the out-of-band approach, Active Network nodes are switches that contain a number of pre-defined protocols. The nodes load the protocols through an auxiliary, i.e., out-of-band, mechanism. Packets passing through these nodes request processing by one of the previously installed protocols. In contrast, packets within an in-band architecture do not request processing by a protocol at a node, but rather carry the protocol, in the form of code, with them as they travel throughout the network. Network nodes process the packet by executing the accompanying code. Since an in-band code loading scheme maps very well to a datagram network such as IP, it is the one that this implementation supports.

1.1.1 ANTS

ANTS (Active Node Transport System) [36] is a reference Active Network implementation, and the one which I have incorporated into my architecture. The ANTS implementation uses a variant of the in-band approach towards building an Active Network architecture. Instead of always transporting code with every packet, ANTS nodes cache the most recently used code in order to avoid reloading the code for a related group of packets. Packets, called capsules in the ANTS implementation, carry parameter values for a related piece of code. If the node that a packet passes through contains the related code, the node initializes the code with a packet's parameter values and then executes the code. If the code is not present on the node, the node requests the code from its nearest upstream neighbor. Using this type of code transport mechanism, Active nodes become primed "on the fly" by the packets that pass through them. This connectionless aspect of ANTS maps well into an IP environment.

1.2 Why integrate Active Networks into IP?

Because IP connects millions of nodes, it will be a good deployment mechanism for Active Networks in the future. Merging ANTS into IP gives researchers and developers the opportunity to study how Active Network nodes function both on local area networks (LANs) and wide area networks (WANs). From a practical point of view it also seems unreasonable to expect that the whole Internet will simultaneously convert to an Active Network model of network communication. Merging an Active Network into IPv6 allows the networking community to study how a real deployment of an Active Network would work, and which aspects of backwards compatibility with IPv6 nodes should be supported.

The integration of an Active Network into IP also gives researchers and developers the ability to quickly and easily experiment with making additions to the IP architecture. In today's environment, running experiments on changes to the IP layer involves changing the source code of an IP implementation and coordinating efforts with other researchers to upgrade their IP nodes to the experimental protocol. If this upgrade only needs to be done locally, the process will be time-consuming but at least manageable. Trying to do this on the scale of a WAN becomes a coordination nightmare and only occurs in the rarest of occasions, such as the upgrade from IPv4 to IPv6, or testing Multicast. In contrast, creating a WAN of Active IPv6 nodes would have a fairly large initial fixed cost, but the marginal cost of testing new changes would be small, i.e., it would only involve writing the code for the protocol, and injecting it into the Active IPv6 network.

Finally fully integrating an Active Network into IP will allow people to experiment with developing Active protocols in a heterogeneous network environment of Active and non-Active nodes. The application-specific protocols created for the ANTS implementation work under the assumption that all nodes and packets in the network will be Active. This architecture will allow network architects to determine the types of useful protocols that can be built when the above assumption does not hold.

1.3 Novel Aspects

The goal behind the design of an Active IPv6 (AIPv6) node was not to design a replacement for an IP node, but rather to design a node that enhanced IP's capabilities. Thus, an AIPv6 node had to be backwards compatible. This created two constraints,

- An AIPv6 node must be able to route both IPv6 and AIPv6 packets.
- AIPv6 packets must never cause a processing error in non-Active IPv6 nodes.

The majority of this report will describe the design and implementation of a node that satisfies these two constraints.

In satisfying the first constraint, I have built an IPv6 node that resembles most other network implementations. It supports network devices and Ethernet drivers, and supplies a programming interface (albeit a simple one) to applications. The node runs in the user-space of the Linux OS and processes raw Ethernet frames. It achieves comparable performance to user-space network implementations despite being written in Java.

What differentiates this node from other implementations is how it processes packets. The node applies a fixed set of processing to all entering packets, and then, if appropriate, processes packets with an application supplied protocol. Packets that do not have a customized protocol undergo default IP processing.

To support customized processing, the node integrates a capsule based Active Network architecture into its packet based IP architecture. Two major questions arise when combining these very different models. How can commonalities between packets and capsules be exploited in the implementation? How can the node achieve security for its resources?

To exploit the similarities between packets and capsules I have made IPv6 the default functionality for all capsule implementations. Building on the work presented in [37] AIPv6 packet, which are the IP analogue of ANTS capsules, utilize an option plus an IP payload to transport parameter values and programming code respectively throughout an IPv6 network. This ensures that non-Active IPv6 end nodes will not generate errors when processing AIPv6 packets. This approach is also novel in that it uses options, which were supposed to support *pre-defined* optional processing, to support *dynamically defined* optional processing.

In order to support security, an Active node should prevent Active packets both from corrupting and from abusing node resources. Though my implementation does not prevent resource abuse it does prevent resource corruption. I have utilized access control mechanisms provided by Java's package and type system to prevent AIPv6 packets from accessing potentially destructive methods. The node also restricts the areas of the packet buffer to which the AIPv6 packet can write. It uses a "protected buffer" scheme that prevents AIPv6 packets from modifying source address and hop limit fields of the packet, thereby insuring a level of accountability, and limiting the amount of time an AIPv6 packet can exist in the network. The protected buffer scheme follows ideas such as sand-boxing used in the design of operating systems - except that the address space has now become a packet's payload.

Finally, to support a programming environment for AIPv6 packets, I have developed a clean and explicit API that a node can export to AIPv6 packets. All customized protocols, including the default IP protocol, utilize this API for packet processing.

1.4 This Report

The following chapter discusses previous work of relevance to AIPv6 nodes. Chapter three describes the design and implementation of the core module in AIPv6 node, and chapter four describes how seamless code transport was achieved. Chapter five discusses the different classes of applications that can be used in a heterogeneous network environment. Chapter six discusses some performance measurements, and concludes.

Chapter 2

Background and Related Work

This chapter gives background information on the protocol and tools used to build an AIPv6 node; and relates this work to research done in the network and operating systems communities.

2.1 IPv6

The AIPv6 node supports the IPv6 protocol [13], which is the successor to IPv4. The major benefit IPv6 has over IPv4 is a larger address space; an IPv6 address has a length of 128 bits whereas an IPv4 address has a length of 32 bits. Additionally IPv6 supports a flow identifier field that enables source nodes to label a stream of packets, and its packet format has been streamlined to enable efficient processing by IPv6 nodes.

One of the goals of developing an AIPv6 node, and its related code transport protocol, is to provide an architecture for adding functionality to the IPv6 protocol. IPv6 tries to address the demand for optional processing by defining a set of extension headers that carry parameter values for pre-determined processing in IPv6 nodes. In IPv4 most of these extension headers were treated as options. The IPv6 specification still does not guarantee support by IPv6 nodes for heavily requested features such as multicast [12] and mobility [26]. Using an AIPv6 architecture in a significant portion of the IPv6 network would allow these features to be easily deployed.

2.2 Java

Java is a relatively new language that has a number of features which make it amenable to an Active Network. It provides an Active node some security when downloading foreign code into the Java execution environment. Also using an object-oriented language such as Java allows for compartmentalization and future upgrading of the node.

The key to security in Java is that the type safe checks performed at compile time can be verified during runtime.[35] Thus, the node can be assured that the Java bytecodes which it downloads will obey the interface boundaries of node resources.

2.3 Active Networks

2.3.1 ANTS

The work presented in this report has a direct relation to Active Networks projects at MIT. The AIPv6 node incorporates the Active Network functionality of ANTS [36]. The code transport protocol and resource interfaces in an ANTS node have direct counterparts in the AIPv6 node. ANTS does not support the IPv6 protocol though, and thus cannot interoperate with Active and non-Active IPv6 nodes. Thus, users cannot use ANTS to either experiment with changes to the IP protocol or study the effects of using Active protocols within a hybrid network of Active and non-Active nodes.

2.3.2 Active Options

The prime motivator for building an IP node that incorporates Active Network functionality has been the Active Options [37] work done here at MIT. This proof of concept work demonstrated the feasibility of building an Active Network within the IP protocol. IPv4 nodes within this architecture interpreted Tcl scripts sent by applications. The Active nodes supported a handful of procedures that the Tcl scripts could call.

There are a number of differences between the Active Options work and the work presented in this paper. The primary one is the level of functionality supported between the two implementations. The Active Option implementation made a few changes to the Linux networking code to study the feasibility of building an Active IP node. In contrast, the AIPv6 implementation has been built from the ground up with the goal of incorporating a more flexible Active Network architecture such as ANTS. Presently an AIPv6 node supports the essential portions of the IPv6 protocol which has made it easy to integrate Active Network functionality, and will make it easy to study and add security measures to the system.

2.3.3 An ActiveBONE

There presently is a proposal [3] to build a test Active Network within the Internet. This network would have similar characteristics to the 6Bone and MBONE, two virtual networks set up to test the IPv6 and Multicast protocols respectively. Both of those test networks connect two disjoint networks by tunneling through IPv4. The primary difference between an ActiveBONE[1] and other test networks is that intermediate Active nodes not specified as tunnel endpoints, will be able to apply Active processing to packets that flow through them. Unlike the AIPv6 implementation described within this report, the "ActiveBONE" proposal does not support direct interoperation with non-Active IPv6 end nodes.

2.3.4 Applications

A number of other researchers have focused on the use of Active Networks to address specific networking problems.

Researchers at Georgia Tech. [6] have focused on incorporating a small set of Active func-

[1]Note, this name has not yet been proposed. I am using it to create an analogy between a test Active Network and the test IPv6 and multicast networks of today.

tionality into an existing IP implementation. Their goal has been to study the benefits an Active Network could have towards reducing network congestion. For experimental purposes they use an out-of-band code approach in which Active packets can only call predefined methods within the network node.

The Protocol Booster [16] project at the University of Pennsylvania has focused on inserting customized processing into protocol stacks. For example, the processing could be a compression or decompression procedure. Also at the University of Pennsylvania is the Active Bridging [2] project that has focused on making an Ethernet bridge that can change its tree discovery protocol "on-the-fly".

2.4 Operating Systems

There has been a great deal work done within the operating systems community to build extensible operating systems that allow applications to customize system services. At the University of Washington work has been done that allows applications to dynamically load extensions into the kernel [5]. Like an AIPv6 node their implementation relies on a type-safe language to provide a level of protection between the resource interfaces and imported code. They have also defined a network architecture in which applications can insert protocol extensions [17]. Unlike the node presented in this report, their work applies only to network host nodes.

Work has also been done at MIT that has focused on safely exposing low level system resources to library operating systems [15]. A related activity also allows applications to insert customized protocol handlers into their kernels [34].

Finally, Sun Microsystems has also implemented a Java TCP/IP stack for their Java OS platform [23]. Though not much has been written about the details of their implementation it appears they have achieved reasonable performance which is a good omen for the AIPv6 effort.

Chapter 3

Node: Forwarding Engine

This chapter describes the design and implementation of the core module, the Forwarding Engine, of an AIPv6 node. What is novel about this implementation is that it divides the IPv6 protocol into constant and variable parts. The constant part of the IPv6 protocol must be applied to all IPv6 packets that traverse this node. The variable part of the IPv6 protocol, such as IP forwarding, can be replaced by application specific protocols. Both the IP forwarding routines, and the application specific protocols use the same node API. Using this approach, Active Network functionality can be cleanly integrated into an AIPv6 node.

The first section of this chapter motivates the purpose of the Forwarding Engine by giving an overview of the structure of an AIPv6 node. The following sections then describe which routines the Forwarding Engine treats as constant and variable. The chapter then moves into a discussion of the packet paths within the Forwarding Engine, and concludes with a presentation of the node API.

3.1 Overview of AIPv6 Node

An AIPv6 node processes both AIPv6 packets and IPv6 packets. AIPv6 packets, whose format is shown in Appendix B, are the IPv6 analogues of ANTS capsules. All packet processing occurs within the Forwarding Engine module whose relationship to the other node modules is shown in Figure 3-1. The device and driver modules, which serve as the link interface to the Forwarding Engine, are described in Appendix A.

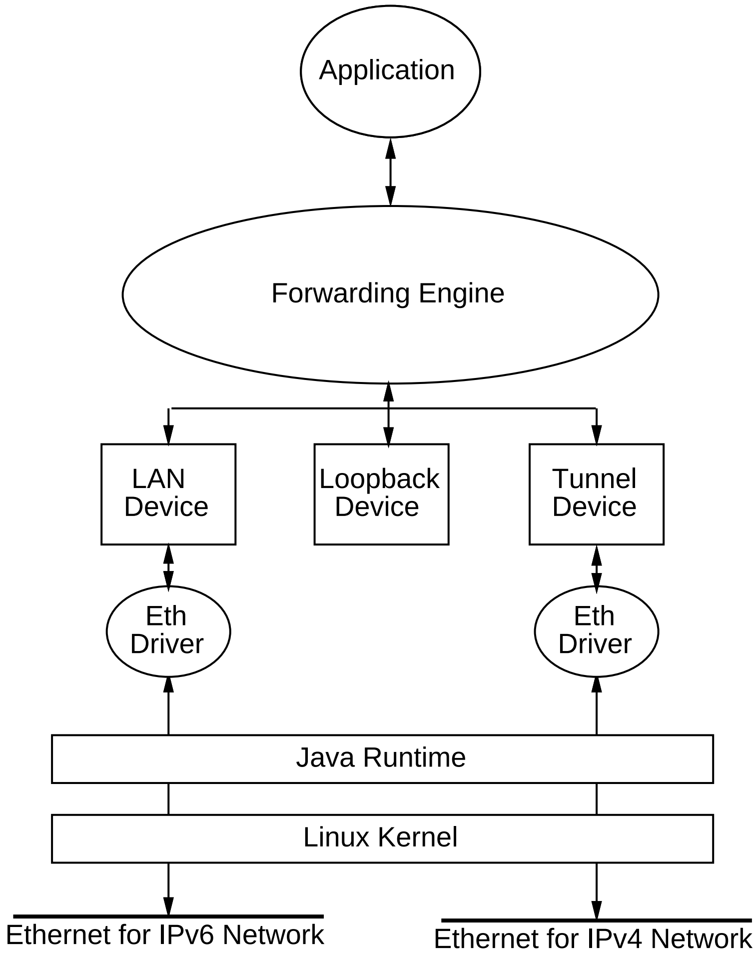

Figure 3-1: Structure of an AIPv6 node.

Inter-module Communication

Packets enter an AIPv6 node either from the network or from an application.[1] Since the Java Runtime does not easily support event and interrupt driven processing at the programming level (with the exception of graphical interface events), AIPv6 modules move packets between them by using upcalls [10]. Each module implements a *send* and *receive*

[1]If an AIPv6 node is used as an end node, then the application would be part of the primary path into the node. An application could also be attached to an intermediate node if it was used, for example, as a network manager.

method which forms the module's communication interface. This simple interface makes the interaction between modules explicit to the implementer, and also makes implementing additional modules, such as other types of network devices, a simple process. *Send* methods correspond to transmitting a packet out to the network, while *receive* methods correspond to receiving a packet from the network. Table 3.1 shows that not all the modules accept the same arguments to their *send* and *receive* methods. As the node gathers more knowledge about the packet buffer moving up its module stack, it converts the packet to a more specialized representation suitable for the module that the packet enters.

Module	Method	Argument	Called By
Application	*receive*	`IPv6Packet` or `Capsule`	Forwarding Engine
Forwarding Engine	*send*	`IPv6Packet` or `Capsule`	Application
	receive	`NetBuff` or `Device`	Devices
Device	*send*	`IPv6Packet`	Forwarding Engine
	receive	`NetBuff`	Driver
Driver	*send*	`NetBuff`	Devices
	receive	`NetBuff`	Linux OS (through Ethernet socket call)

Table 3.1: Communication interfaces implemented by each module.

3.2 Constant Processing

Constant processing is the common processing that all packets, both IPv6 and AIPv6, entering the Forwarding Engine must undergo. It represents the subset of the IPv6 protocol that must be applied to every packet.

The Forwarding Engine treats decrementing the hop count of a packet as constant processing. By requiring every node that a packet passes through to decrement the packet's hop count, the IPv6 protocol uses a hop count field to prevent routing loops among its packets. Packets with a hop count equal to zero can no longer be forwarded. By making this process constant, an AIPv6 node can ensure that all AIPv6 packets will not circumvent this requirement.

3.3 Variable Processing

Variable processing in an Active node is processing that is application dependent. This type of processing occurs after constant processing, and can either be an application-specific network protocol, or if no protocol has been supplied with the packet, the default IP forwarding routine. All variable processing methods, including the default IP forwarding routine, use the same node primitives to manipulate and transmit packets.

3.4 Packet Paths

Figure 3-2 shows the packet flow paths within the Forwarding Engine. Packets can enter the Forwarding Engine from two entry points, either from a device via the *receive* method, or from an application via the *send* method. Since the Linux kernel provides its own packet queues, a packet entering the Forwarding Engine via one of the communication methods does not need to be buffered and is processed immediately.

The first two classes listed in Table 3.2, the `IPForwarder` and the `ActiveForwarder`, form the heart of the Forwarding Engine. The `IPForwarder` provides all the IP packet processing methods to the AIPv6 node, while `ActiveForwarder` provides all the Active processing methods, such as the execution of AIPv6 packets, and the demand-loading of code across a network. The `ActiveForwarder` has been an implemented as a sublcass of the `IPForwarder` and has access to all its superclass's methods. When combined, the public methods and variables of the `IPForwarder` and `ActiveForwarder` classes provide a node API to AIPv6 packets.

3.4.1 *Receive* Processing

The *receive* method in the `IPForwarder` class (shown in Figure 3-3) defines processing for packets entering the Forwarding Engine from the network. This method partitions the processing into constant and variable routines. Neither the *receive* method nor the constant processing in the *receive* method can be overwritten by subclasses of `IPForwarder`. This

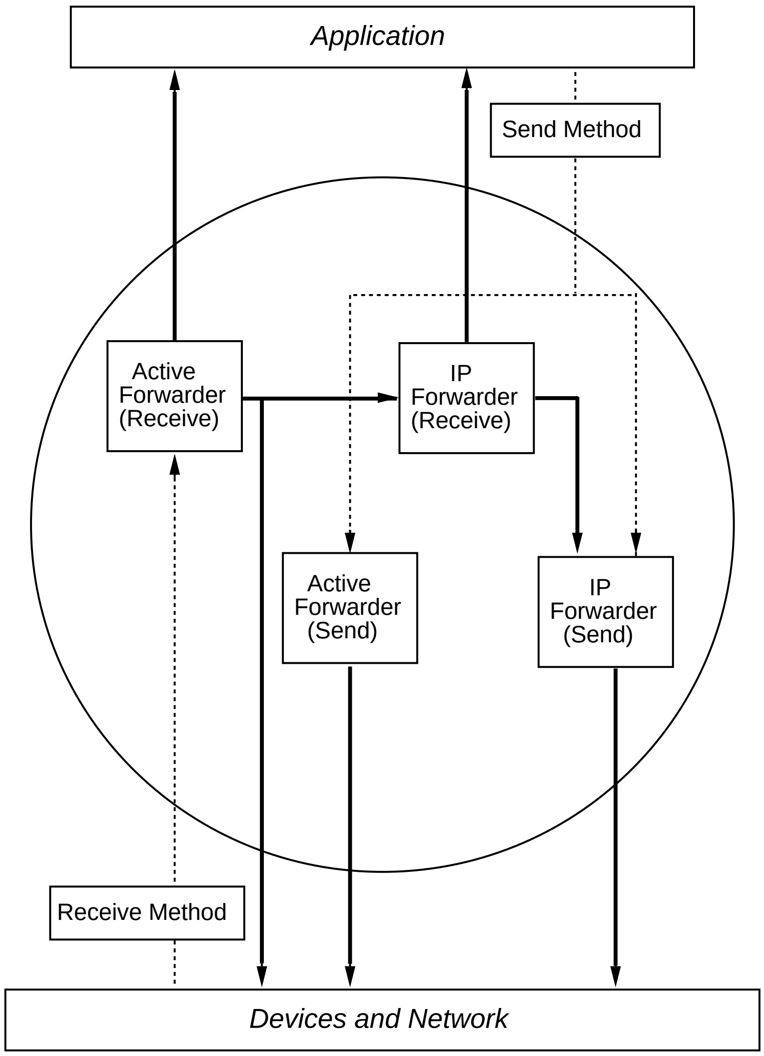

Figure 3-2: Packet paths through the Forwarding Engine

ensures that classes, such as the `ActiveForwarder`, which add Active processing to an IP node, must apply a constant set of IP processing to every packet that traverses the node. The variable portion of the *receive* method can be overwritten, and is used to add application specific processing mechanisms to the Forwarding Engine.

The *constantProcess* routine ensures that a packet has not exceeded its lifetime in the network. It checks the hop limit field of a packet buffer, and drops the buffer if its field's value equals zero. If the value does not equal zero then the routine decrements the value by one, signifying that the packet has passed through this node.

The Forwarding Engine contains two *variableProcess* methods, one in the `IPForwarder`

Classes	Description
`IPForwarder`	Handles all IP packet processing.
	Supports table lookup, *send* and *receive* routines.
`ActiveForwarder`	Handles all Active packet processing.
	Supports packet execution, and demand loading.
`DeviceManager`	Constructs and manages devices for the Forwarder.
`LANDevice`	Node's interface to an IPv6 network.
`TunnelDevice`	Node's interface to an IPv4 network
	through which IPv6 packets will be tunneled.
`LoopbackDevice`	Node's interface to itself.

Table 3.2: Core forwarding engine classes.

and one in the `ActiveForwarder` classes. These methods are described in more detail in sections, 3.5 and 3.6.

```
final public void receive(NetBuff buf, Device dev)
{
   try {
      constantProcess(buf, dev);

      variableProcess(buf, dev);
   }
   catch (ICMPErrorException e) {
      // drop packet and send ICMP Error message
   }
   catch (ICMPInfoException e) {
      // send ICMP Info message
   }
}

final public void constantProcess(NetBuff buf, Device dev)
   throws ICMPErrorException
{
   IPv6Packet.decrementHopLimit(buf);
}
```

Figure 3-3: *receive* and *constantProcess* methods in `IPForwarder`.

3.4.2 *Send* Processing

Packets entering the node via the *send* method do not undergo constant processing. Instead they will be processed directly by either their customized protocol or the default IP forwarding routine. Since packets have been instantiated locally, and correctly initialized, it is assumed that they are valid IP packets.

3.5 Default IPv6 Processing

The *variableProcess* method, shown in Figure 3-4, for the `IPForwarder` class is called only when a packet needs to undergo IP processing. After processing a packet's hop-by-hop extension header, the method applies some address specific processing to the packet if necessary. If the packet is not addressed to the node, then the engine will call the *routeForNode* method, shown in Figure 3-5, which handles the forwarding of a packet. This method applies a pruning algorithm to the routes in an AIPv6 node's routing table in order to determine the next route for a packet. The route returned by the lookup method contains both the next hop IP address to which the node should send the packet, and the name of the device that connects to the same network as the next hop IP address.

The lookup method supports two processing paths, the fast path and the slow path. Packets enter the fast path when their destination address equals one of the destination address keys in the node's routing table cache. A route containing the next hop address is returned from this cache hit. Packets whose destination address yields a cache miss must pay the penalty of having their route calculated by the pruning algorithm. The route calculated by this algorithm will be stored in the routing table cache for subsequent packets.

The `IPv6Packet` class, used by the processing routines in the `IPForwarder` and `ActiveForwarder` classes, has two purposes: it supplies easy access to the IPv6 header fields and to all extension headers plus payload in an IPv6 packet, and it enforces the IPv6 packet format rules set by the IPv6 specification.

3.6 Active Processing

The *variableProcess* method, shown in Figure 3-6, in the `ActiveForwarder` class applies Active processing to all the packets entering the Forwarding Engine from the network. IPv6 packets which do not contain Active fields default to the *variableProcess* method in the `IPForwarder` class. If the packet is an AIPv6 packet then it will undergo ANTS-based active processing in which the code associated with an AIPv6 packet will be demand-loaded and/or executed. Execution of an AIPv6 packet involves calling the packet's evaluate method and

```
public void variableProcess(NetBuff buf, Device dev)
   throws ICMPErrorException, ICMPInfoException
{
   IPv6Packet packet = new IPv6Packet(buf);

   processHopByHop(packet);

   if (isNodeAddress(packet.dest())) {

     processRemainingExtensionHeaders(packet);

     if (nodeIsAHost())
       deliverToApp(packet);

       return;
   }

   routeForNode(packet);
}
```

Figure 3-4: *variableProcess* method in `IPForwarder`.

```
public void routeForNode(IPv6Packet packet)
{
   Route rte = lookup(packet);

   if (!rte.isUp())
     return;

   if (rte.toGateway()) {
     devMgr.getDevice(rte.device).send(packet, rte.gateway);
     return;
   }

   devMgr.getDevice(rte.device).send(packet, packet.dest());
}
```

Figure 3-5: Route method for an IP packet.

supplying a reference to the ActiveForwarder object as the method's argument.

Applications inject new protocols into the network by subclassing the `Capsule` class and overwriting its evaluate, serialize, and deserialize methods. Since the `Capsule` class is a subclass of `IPv6Packet` all capsules have access to a packet's IPv6 extension headers and payload. Access to the first 40 bytes is restricted through the use of the protected buffering scheme described in Section 3.7.

```
public void variableProcess(NetBuff buf, Device dev)
   throws ICMPErrorException
{
    if (!IPv6Packet.isActive(buf))
      super.variableProcess(buf, dev);

    // the deserialize method demand-loads code when
    // necessary
    Capsule cap = deserialize(buf, dev.address());

    if (cap != null)
      cap.evaluate(this);
}
```

Figure 3-6: *variableProcess* method in `ActiveForwarder`.

3.7 Protected Packet Buffers

The code associated with an AIPv6 Packet needs access to the packet buffer in order change the values of existing fields. However, giving the code access to all of the IPv6 packet would allow the code to change the source address or hop limit fields in the header and could create a serious security problem for other nodes in the network.

Protected buffers prevent variable processing methods from modifying some fields of the packet, such as the source address and hop limit fields, by checking each byte access made into the buffer. Every attempt to modify the byte is checked at runtime. The `NetBuff` class, described in Appendix A, supports the protected buffer scheme by providing two versions of a *setElement* method. One version, used by classes considered unfriendly[2] by the `NetBuff` class performs this runtime check. The other version, used by friendly classes, has an equivalent *setElement* method that does not perform this runtime check.

3.8 API Available to Variable Processing

The essential part of an Active node is the API that it exports to mobile code embedded in packets. When mobile code enters an AIPv6 node it is supplied a reference to the forwarder object. The code has access to all the forwarder's public variables and methods. Table 3.3

[2] In Java, a friendly class to the `NetBuff` class would be one which resides in the same "package", or directory as `NetBuff`.

lists the API supported by an AIPv6 node. The following sub-sections describe in detail the interesting characteristics of the API.

Class	Methods
Routing Table	
	List All Elements
Forwarder	
	TableLookup
	ForwardPacket
	Create packet
	Create IP address
	Generate ICMP message
Soft-state cache	
	Add
	Delete
	Get

Table 3.3: Node API.

3.8.1 Resources

Soft-state Cache

An AIPv6 node incorporates a number of ideas from the ANTS implementation, the most important of which is the soft-state cache. The soft-state cache is an Active packet's scratch space that can be shared by a series of packets associated with the same application group or protocol family. Packets from the same group or family, can create, read, or write entries in the cache, though the node makes no guarantees about the persistence of such data.

Routing Table

The routing table is the most important resource for an IP node. It contains all the routing information needed to send a packet onto its next hop or destination. Since an AIPv6 node can be rendered useless without a routing table, AIPv6 packet only have read-access to the routing information in the table. AIPv6 packets which require different routes than those supported in the routing table, can add their routes to the soft-state cache.

3.8.2 ICMP Exceptions

One of the novel aspects of this node API is how it handles ICMP error processing. Normally, ICMP error processing is hidden within the thousands of lines of code in a node implementation, but within an AIPv6 node that processing has now become explicit. This section explains this idea in more detail.

ICMP

IP uses the Internet Control Message Protocol (ICMP) to communicate error and informational data between IP nodes. A node uses an ICMP error message to alert a source node that it has been sending packets with invalid fields or lengths. A node will use an ICMP informational message for resource discovery such as determining the members in a Multicast group, or whether a node is operational.

How Linux Handles ICMP

In Linux, a routine can generate, and send, an ICMP message without ever alerting its calling routine that it has done so. This type of implementation makes the code hard to extend and manage for a number of reasons. First, it disperses the control of sending messages throughout the whole node. Second, it moves knowledge about sending ICMP messages into code that probably does not need this information.

Exceptions

Exceptions separate abnormal processing from the common case, thereby simplifying programming code. Since most languages that support exceptions require method declarations to list any exceptions that might be thrown, the specification for that code becomes more explicit. This is especially important when defining an interface to programming methods and objects (such as in an AIPv6 node). Furthermore, exceptions also provide the benefit of allowing calling methods to decide how to handle exceptional conditions instead of hard-coding responses in lower level procedures.

ICMP Exceptions

An AIPv6 node utilizes exceptions to handle conditions which require ICMp processing. Methods that catch ICMP exceptions can elect to either handle them locally or pass them up the call stack to another method which might be in a better position to determine whether an ICMP message needs to be sent.

Chapter 4

Code Transport in AIPv6

This chapter describes the design and implementation of the code transport mechanism used by AIPv6 nodes. It begins with a discussion of the requirements for a successful integration of a code transport protocol into IPv6, and then moves into a detailed explanation of the demand-load protocol used in ANTS. It then discusses the various choices IPv6 presents for transporting data and concludes with the solution that has been chosen.

4.1 Goals for integrating demand-load capabilities into IPv6

The key to achieving interoperation between an AIPv6 node and an IPv6 node is the code transport mechanism. Since all AIPv6 nodes can process IPv6 packets, an AIPv6 node can be deployed in an IPv6 network. However if the code transport mechanism placed programming code, or state values, in a location in an IPv6 packet that caused IPv6 nodes to generate an error while processing the code, then the AIPv6 packet would never arrive at its destination. Any solution must avoid this problem in order to achieve complete interoperation with IPv6 nodes.

The proposed solution must also work within the existing IPv6 specification, and not, for the moment at least, require the IANA to define a new Active protocol. If the solution cannot be used in today's IPv6 network then it is not valid.

4.2 The Demand-load Protocol in ANTS

As mentioned in Chapter 1, ANTS uses an in-band code loading mechanism in which a node loads a packet's code into their memory while processing the packet. The extreme version of an in-band code loading mechanism requires each packet to carry both code and corresponding state values in order for the packet to be processed by each node it traverses. Thus, even though consecutive packets may be processed by the same code, each packet still carries its own code.

ANTS eliminates this redundant code transport by using a demand-load protocol. Instead of transporting code with every packet, ANTS relies on the nodes within its architecture to send out requests for code when they do not have the code available to process a packet. In the present implementation of ANTS, a node sends a code request only to the adjacent upstream node. Upon receiving a code response from the upstream node, the requesting node stores the code within its cache for some finite amount of time. It will then use this code to process all the successive packets that have the same identifier as the code. Figure 4-1 shows the complete demand-load process in ANTS. Since this process handles all the code transfer, applications need only to embed state values for a corresponding piece of code within the packets that they send out onto the network. As long as the applications supply programming code to the node on which they run, then the applications can be assured that their packets will be executed correctly throughout the network.

The ANTS demand load protocol has a number of nice properties. First, programming code is sent only to the nodes that request it. Applications need neither pre-determine a packet's route, and then upload code to every node the packet will traverse, nor broadcast the programming code to the whole network. Furthermore, since most network protocols usually use a stream of packets, the overhead of doing the initial demand-load is amortized over the number of packets sent during the protocol transmission. Of course, for protocols which use a small number of packets the demand-load overhead might be quite high.

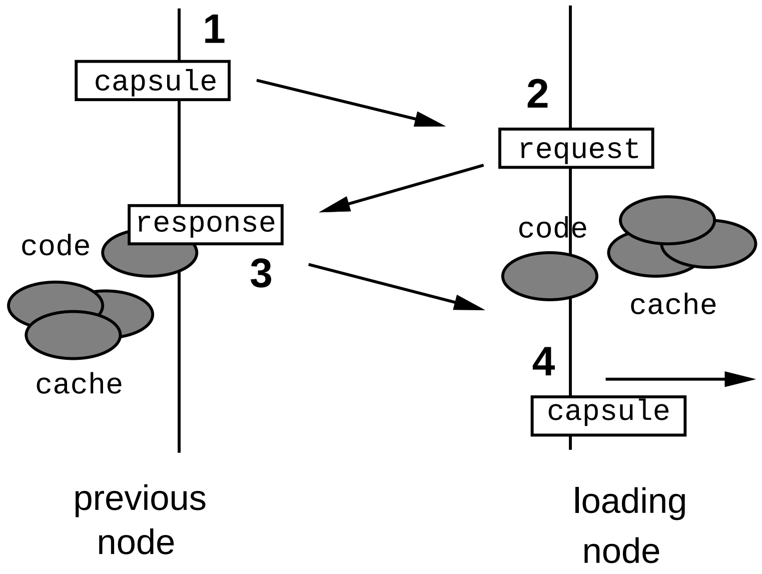

Figure 4-1: Demand loading in ANTS.

4.3 Choices for Code Transport

This section describes and analyzes the choices available for integrating an Active Network into an IPv6 network. Section 4.4 presents the solution which I have chosen for the AIPv6 protocol.

4.3.1 Tunneling

Tunneling involves the encapsulation of one protocol within another. It is used when two nodes cannot be linked using their own protocol.

The benefit of tunneling is that if one only wishes to connect two Active nodes that are separated by an IPv6 network then tunneling will work perfectly and not upset the IPv6 network. Tunneling has worked well for a number of networks such as the MBone and the 6bone. Both of these networks though require knowledge of the tunnel end points.

Tunneling would not integrate Active Networks into IPv6. Essentially what tunneling would do is treat IPv6 as a link layer. In this scenario an ANTS capsule would have to be encapsulated in IPv6 before it is sent to the next Active node. An application running on a pure IPv6 node would never be able to interact with Active protocols because its IPv6 node would never process Active packets destined for it.

4.3.2 Using Gateways

Another possibility is to tunnel between Active nodes and then to strip the Active payload before an Active packet is sent to an IPv6 node. This would ensure interoperability between Active and non-Active nodes. Within the networking field, a node which converts one protocol to another is called a gateway. This type of scheme would require an Active node to function as both a router and a gateway.

The difficulty in making an Active node a gateway is, that in order for the gateway portion of the node to function properly, the node will need to know the relative positions of all IPv6 nodes and Active nodes in the Internet. Because Active packets can be mapped onto IPv6 packets (just remove the programming code), but IPv6 packets cannot be mapped back onto Active packets, it is imperative that an Active node not remove programming code from a packet that might still pass through an Active node. The cost of insuring a correct removal (an IPv6 node discovery protocol, and an enormous table containing the locations of all IPv6 nodes) seem to outweigh any of the interoperability benefits that could be gained from this choice.

4.3.3 IPv6 Mechanisms

The alternative to using tunnels or gateways is to transport code and state values in either a payload or an option. This sub-section describes the benefits and drawbacks of each of these choices.

IPv6 Extension Header

Defining a new IPv6 extension header that supported Active Networks would be a Herculean task from a politically viewpoint. Since all IPv6 nodes must be able process IPv6 extension headers, all IPv6 nodes would need to be Active nodes. This does not appear to be a politically viable option.

Upper Layer Payload

An upper layer payload, is the section of an IPv6 packet that follows the IPv6 header and extension headers. It is not included as part of the IPv6 specification. Unlike an IPv6 extension header, the standardization of an upper layer protocol only applies to the end node implementations that support it. Therefore, a payload, whose size restriction is the maximum size of an IPv6 packet, would be a good vehicle in which to transport large fragments of code and/or state values. An option could also be inserted into the hop-by-hop options header that would alert AIPv6 routers to the presence of this payload within the packet. This type of option has been proposed in [21]. The benefit of this option is that an AIPv6 router would not have to parse every packet's extension headers in search of an Active header.

Unfortunately, defining a new payload which all IPv6 nodes do not support means that IPv6 end nodes would drop packets containing this payload and send an ICMP error message back to the packets' source nodes. This would violate my goal of seamless interoperation and therefore cannot be used to transport any data that might arrive at IPv6 nodes.

An Option in the Hop-by-Hop Options Header

An IPv6 option provides properties for pre-determined non-standard processing that an IPv6 node may apply to an associated packet. An IPv6 option would be a good choice for code transport because it can instruct IPv6 nodes (source, destination and intermediate) to ignore the option if the nodes do not understand it. Using an option to transport code continues on the work done in [37]. Furthermore since there will be network socket support for applications at the end nodes to insert options in and receive options from the hop-by-hop options header, using an option would allow an application to use Active Network technology while running over an IPv6 end node. The one unfortunate draw back about using an option is that its size is restricted to 256 bytes. This makes it a good vehicle to transport state values, which typically have a size of 50 bytes, but probably not code fragments, which typical have a size of 750-1000 bytes.

Since the hop-by-hop options header has a maximum size of 2056 bytes, and therefore can hold a number of options, one could break up a code fragment into a number of pieces, and place each piece into an option. The main drawback of this would be that each Active node would need to parse about five options for every packet contain about 1K code fragments. This seems like an unnecessary performance hit.

4.4 The Solution: Combine Option and Payload

The solution that I have implemented views the transport process as having two separate parts, code transport and state value transport.

In an Active network, code transport is an end-to-end process between two Active nodes. Active node A sends a code request to Active node B, and Active node B sends a code response to Active node A. Therefore in a hybrid network of Active and non-Active nodes, an IPv6 payload can be used to transport code, since the only two nodes that will attempt to process the payload will be Active.

State value transport, on the other hand, is an in-network process that involves all nodes in an Active network. In a hybrid Active network, this means that state values could potentially be processed by an IPv6 intermediate node or end node. This leaves the network architect with two choices:

- Place the state values in an option, whose maximum size is 256 bytes, and enable interoperation with IPv6 intermediate and end nodes.
- Place the state values in an IPv6 payload, whose maximum size is 2056 bytes, and enable interoperation with IPv6 intermediate nodes, but require that the source node only send packets to Active destination nodes.

I have decided to choose robustness of operation over robustness of size, and use an option to transport state values. Since one of the goals behind building an AIPv6 node is to provide researchers and developers a platform in which to experiment with a heterogeneous network of Active and non-Active nodes, I feel this choice this choice is warranted.

The AIPv6 transport protocol uses one option, the marshaled option, and one payload, the system payload, to transport code and state values in an IPv6 network. The marshaled option carries state values, and the system payload will carry data for demand requests, and responses.[1] The marshaled option maps directly to a marshaled capsule in ANTS, and the system payload maps directly to a marshaled system capsule in ANTS. Details on the format of the option and payload can be found in Appendix B.

[1]AIPv6 nodes did not support code transport at the time of this writing.

Chapter 5

Application Protocols

This chapter describes the types of application specific protocols that can be built for a network consisting of AIPv6 and IPv6 nodes. The first section discusses the constraints placed on Active protocols when run in a heterogeneous network, and the second section describes a potential real-world application that takes advantage of an AIPv6 architecture.

5.1 Constraints

A hybrid network of AIPv6 and IPv6 nodes solves the problem of integrating an Active Network layer into the IP layer. Within this architecture, an Active protocol is guaranteed to be executed by Active IPv6 nodes, and ignored and processed like an IP packet by non-Active IPv6 nodes.

This architecture, however, places a set of constraints on Active protocols migrate from a purely Active environment to a heterogeneous Active environment. In ANTS, for example, applications assume that Active protocols will, at the very least, be executed by at least two Active nodes, the source and destination node. If the Active protocol passed through intermediary nodes on its way to the destination node, then it would also be executed at those nodes. In an AIPv6 architecture the same assumptions cannot be made. Since applications using AIPv6 protocols can connect to IPv6 destination nodes, they can assume only that their protocols will execute at one node, the source node.[1] If the source node knows beforehand that the destination node is an AIPv6 node then the guarantee moves closer to the guarantee provided by ANTS.

5.1.1 A Compression Example

Figures 5-1 and 5-2 illustrate how an Active protocol can have different effects when running in an ANTS network versus running in a hybrid network of AIPv6 and IPv6 nodes. The source applications in both figures insert compression protocols into the network to be used on some packets. Neither of the destination applications can parse compressed data. The compression protocol compresses a packet's payload at a node with a low-bandwidth link, and must uncompress the payload before it reaches the destination application.

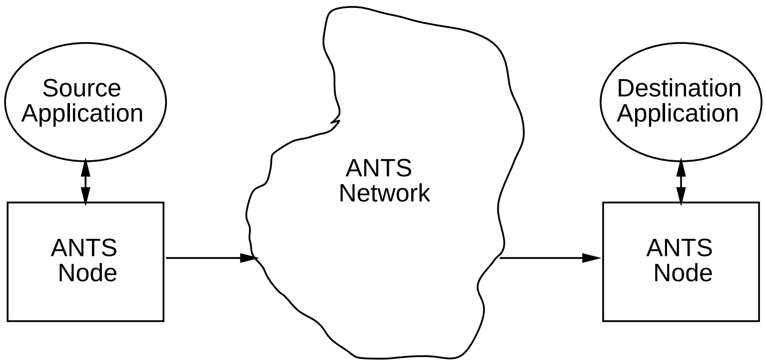

Figure 5-1: Active compression protocol in an ANTS network

Within an ANTS network, the compression protocol will always work correctly. It can compress the payload at an intermediate node, and decompress it at the destination node. It will never deliver compressed data to the destination application. Within an AIPv6 network though, the same guarantee of correctness no longer holds because an end node could be a non-Active node. In this case correctness can be assured only when either the source application transmits to Active destination nodes, or over routes with a known number of intermediate nodes, or the destination application understands compressed data.

[1] In the present AIPv6 implementation, an application must run over an AIPv6 node in order to insert an Active protocol into the network. If, in the future, some Active protocols are added to the core set of protocols supported by all AIPv6 nodes, then the source application will only need to transmit an Active option, and not need to run over an AIPv6 node.

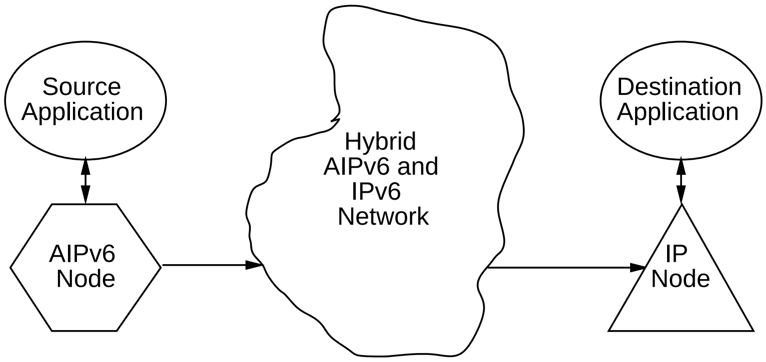

Figure 5-2: Active compression protocol in hybrid AIPv6 and IPv6 network.

5.1.2 Not all Active Protocols Transfer Directly

The previous example shows that there exist a class of Active protocols that cannot directly transfer to an AIPv6 network. To work correctly in an AIPv6 network these protocols require the source application to have knowledge about the types of nodes in the network, or require the destination application to have knowledge about the Active protocol.

5.2 Future Active Protocol: Load Balancing

This section describes a hypothetical Active protocol that would use a hybrid network of AIPv6 and IPv6 nodes. The purpose of this exercise is to demonstrate that the design goal of requiring interoperation between an AIPv6 source node and an IPv6 destination node can produce useful protocols.

5.2.1 Why Load Balance?

No server has yet been built that can quickly process all the **http** requests coming into the most popular sites on the World Wide Web (WWW). To combat this problem, sites distribute their **http** requests among a set of web servers. These sites try to ensure an equal distribution among their web servers by using load-balancing schemes.

5.2.2 Coarse-Grained Load Balancing

Coarse-grained load balancing uses the Domain Name Service (DNS) to return the IP address of the least loaded web server.[2]

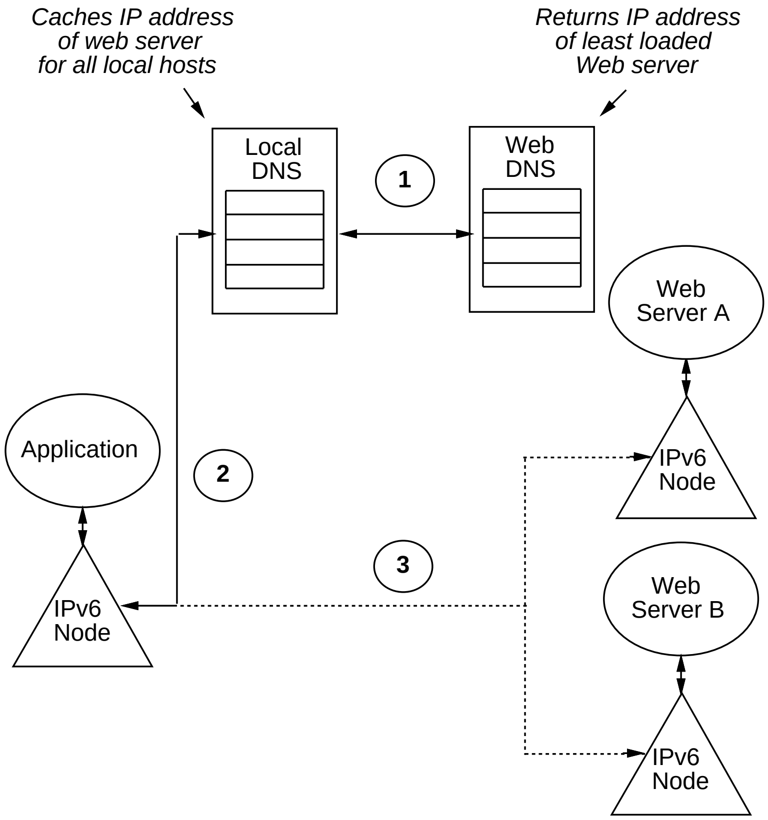

Figure 5-3: Coarse-grained Load Balancing.

Step 1 in Figure 5-3 shows an application sending a DNS resolution request for www.mit.edu to it's local DNS server. Since the local DNS does not yet have a binding for www.mit.edu it contacts MIT's DNS server. During Step 2, MIT's DNS server determines which web server is the least loaded, at the time of request, and resolves the DNS request to that server's IP address. For this example assume the DNS request resolves to web server A. Once receiving a reply from MIT's DNS server, the local DNS server will cache web server A's IP address, and then return the IP address to the requesting host. In step 3 the host will then initiate an **http** request with web server A.

[2]The purpose of DNS is to resolve a hostname such as, www.mit.edu, to an IP address. Each network site that advertises a hostname must setup a DNS server to handle the resolution.

There are two problems with this scheme that could disrupt the load balanced properties of MIT's web servers. One, since the application stores the hostname-IP address binding for the duration of its session, it will only connect to the least-loaded server during its first connection. After that point, what originally was the least-loaded server could quickly become the most-loaded server. Two, since the local DNS server caches the hostname-IP address binding for some period of time, other nodes at the local site will use the same binding, and increase web server A's load. This affects both the client, who faces longer delays in receiving data from MIT's web servers, and MIT web's site, which cannot process as many requests as it should be able to.

5.2.3 Fine-Grained Active Load Balancing

Fine-grained Active load balancing would build load balancing characteristics into the **tcp** protocol that underlies the **http** protocol. It would allow a web client to connect to the least loaded server during every web connection. Web clients that do not have access to Active Network technologies will always connect using the coarse-grained load balancing scheme.

Step 1 of the scheme involves an Active application querying its local DNS server for the IP address of www.mit.edu. Since the local DNS server doesn't have the binding, the server contacts MIT's DNS server. This DNS server, using the coarse-grained load balancing algorithm, then returns the IP address of the least loaded server. Upon receiving a response from MIT's DNS server, the local DNS server cache's a binding between www.mit.edu and the returned IP address, and then returns this IP address to the requesting application. In step 3, the first **tcp** connection packet that the application sends will search the AIPv6 node for a list of available web servers. It will then choose which web server is the least loaded and connect to it. The IP address of the server that it connects to will then be used for the rest of the connection. Since a new request requires a new **tcp** connection, the Active web client will always connect to the least loaded web server.

The beauty of this scheme is that if the application does not run this Active protocol, then all its web connections will default to the coarse-grained load-balancing scheme.

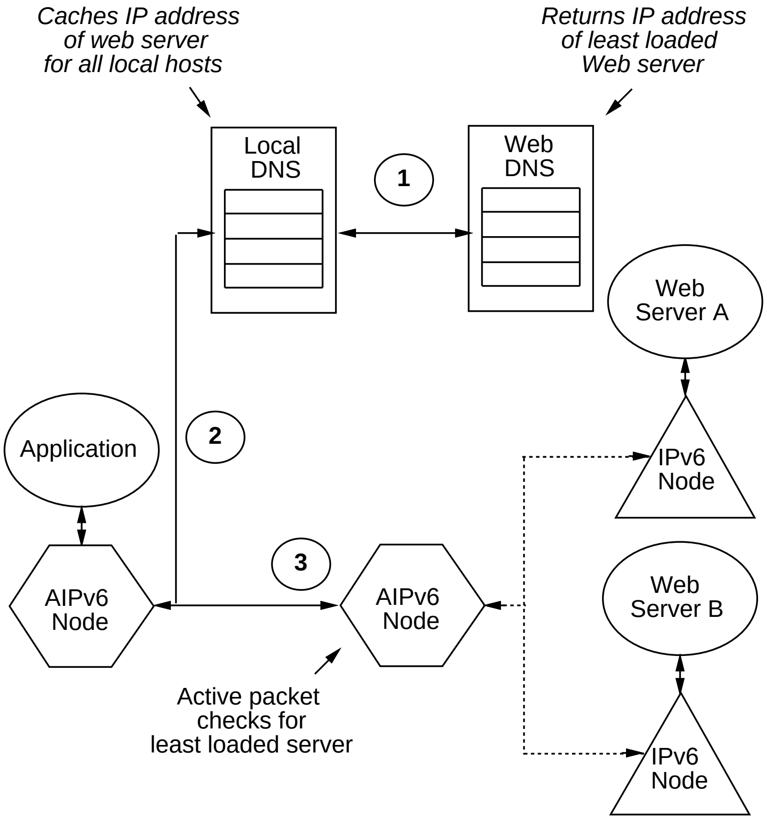

Figure 5-4: Fine-grained DNS load-balancing.

The changes required to implement this Active protocol would be a change in the initial hand shake code of a **tcp** connection (it will need to recognize that the destination address changes during mid-handshake). Changing an IP address during mid-connection could be a major security problem but it is quite similar to changing port numbers for an FTP connection (though with FTP if the client assumes the machine your connecting to is safe, then it most likely won't have an evil port number).

This protocol also requires a process running at the web site that queries the web servers for their load, and then inserts a load table into the AIPv6 border router.

Chapter 6

Performance Analysis and Conclusions

Despite being written in Java, an AIPv6 node achieves respectable performance. When routing only IPv6 packets, an AIPv6 node achieves an average fast-path throughput of 979 packets/sec. It achieves an average fast path throughput of 766 capsules/sec when routing simple AIPv6 capsules. These results are a significant fraction of the maximum throughput, 3750 pkts/sec, that can be achieved by user space network implementations on the Linux OS.[1].

6.1 Description of Experiment

Figure 6-1 shows the experimental setup used to test the throughput and latency of an AIPv6 node. Results were recorded for the AIPv6 router, shown as the intermediate node in the figure, as packets flowed from the source node to the destination. The intermediate node's routing table contained three routes, a loopback route, a route to the source node, and a route to the destination node.

[1] This upper bound was determined by implementing a user-space application in C that received Ethernet packets from an Ethernet socket and automatically forwarded them to a hardcoded destination. The machine used for this experiment was a 200 MHz Pentium Pro

All three machines used in the experiment ran the Debian 1.2.x distribution of the Linux 2.0.30 kernel. The intermediate and destination machines had Pentium Pro 200 MHz processors, while the source machine had a Pentium 120 MHz processor.

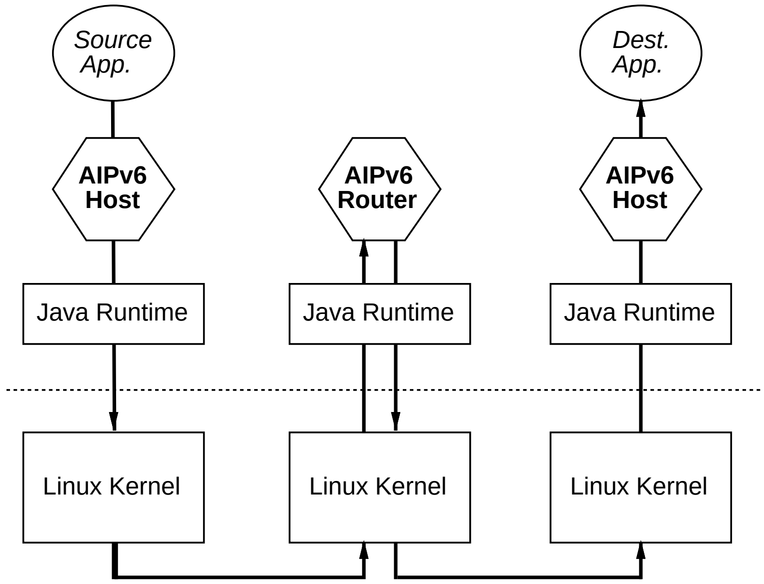

Figure 6-1: Experimental set up.

6.1.1 Packet Types

During each test, the source node sent one of two types of packets, a vanilla IPv6 packet that contained only an IPv6 header, or a simple Null capsule that routed itself through the nodes. Both packets were forwarded identically, but traveled through different processing paths within the node, the default and Active paths.

Vanilla IPv6 Packet

The vanilla IPv6 packet contained only the IPv6 header. Since it does not have a marshaled option, the node processes it using the default IPv6 routine shown earlier in Figure 3-4.

Null Capsule

The purpose of the Null Capsule is to test the performance associated with the node's Active processing path. Figure 6-2 shows the Null Capsule's evaluate method, which is a simplified version of the default IPv6 routine.

```
public boolean evaluate(Forwarder fwdr) {
  if (fwdr.isNodeAddress(dest()))
  {
    return fwdr.deliverToApp(this);
  }
  else
  {
    return fwdr.routeForNode(this);
  }
}
```

Figure 6-2: Null Capsule's evaluate method.

6.1.2 Fast Path vs. Slow Path

Since there is only one route between the source and destination nodes in this experiment, the second packet in a flow of packets always enters a node's fast path. Thus, to get meaningful results during the slow-path experiments, the node did not use a routing table cache.[2]

6.2 Results

Table 6.1 shows the maximum average throughput the intermediate node could sustain over a run of 5500 packets. The destination node recorded these values by calculating the throughput for every 500 packets that it received. The throughput for the first 500 packets was not included in the calculation to account for the node "warming up".

[2]The slow-path studied here is the one described in Section 3.5. It is presumed that once the AIPv6 implementation supports code loading, that the slowest path will be the code loading path and not the one associated with doing a table lookup.

Packet	Throughput (Fast Path) (packets/s)	Throughput (Slow Path) (packets/s)
IPv6	979	507
AIPv6 (Null Capsule)	766	444
C User-space App.	3750	NA

Table 6.1: Throughput results.

6.3 Discussion of Performance

The performance results in Section 6.2 are encouraging. Given that studies have shown that interpreting Java bytecodes is at least an order of magnitude slower than executing machine code, the performance of the system probably can be greatly improved by compiling the node down to native machine code, or, at least, running it with a Just-In-Time compiler. Both of these solutions are available today, but, because of time constraints, they were not utilized for these experiments.

There are also two other potential areas for improved performance: reducing the scheduling overhead caused by running an Active node over both the Java runtime and the Linux OS, and reducing the copying overhead caused by running the node in the Linux user address space. Both improvements can be realized by integrating a Java runtime into the Linux kernel, or by using an already integrated platform such as the Java OS.

Besides showing reasonable performance there is one other interesting characteristic about the results in Section 6.2; the throughput associated with processing capsules is less than that associated with IPv6 packets. This difference can be explained by the startup costs involved with executing a capsule in this implementation. It is a three step process that involves loading in the associated code from memory, instantiating an instance of the code using the state values from the Marshaled option of a IPv6 packet, and then executing the evaluation method of the instance. In addition, because of the protected buffering scheme used by this node, all byte accesses made to the packet buffer while creating a capsule must undergo a runtime check.

6.4 Summary and Conclusions

An AIPv6 node demonstrates that an Active Network capsule based architecture can be integrated into an IP packet based architecture. The AIPv6 node takes advantage of that which is similar between these architectures by dividing its processing into constant and variable segments. By using control mechanisms, such as protected buffers, the node ensures that capsules will not change important fields in the IPv6 packet header.

A unique aspect about the AIPv6 architecture, is that it supports interoperation between an Active source node and a non-Active destination node. This places a new set of constraints on Active protocols which had been designed for a pure Active Network. Section 5.1 provided some preliminary ideas about what these constraints are.

Supporting interoperation should also enable the creation of new types of protocols, such as the one proposed in Section 5.2, that could not be implemented using only an IP architecture or an Active Network architecture. This characteristic, combined with its encouraging performance results, suggests the AIPv6 platform will be appealing to researchers who wish to study application specific protocols for the Internet.

Appendix A

Node: Link Interfaces

This appendix describes the link level interfaces supported by an AIPv6 node.

A.1 Network Devices

The AIPv6 device classes translate outbound IP addresses to Ethernet addresses, and apply link specific processing to incoming and outgoing packets. The AIPv6 node supports three generic types of devices: loopback, tunnel, and LAN. The following subsections describe them.

A.1.1 LAN Device

Some devices do nothing more than pass the packet they receive down to the device driver which handles placing bits onto the physical network. This is the purpose of the IPv6 network device which represents a node's interface to an IPv6 network. No extra processing is needed to send IPv6 packets to other IPv6 nodes located on the same physical network.

A.1.2 Loopback Device

The loopback device represents the node's interface to itself.[1] When the Forwarding Engine passes a packet to the loopback device, the device automatically returns the packet to the Forwarding Engine's receive queue (by calling the forwarder's receive function). This prevents the packet, which is destined for same node, from entering the physical network.

A.1.3 Tunnel Device

The tunnel device is the node's interface to an IPv4 network. Because IPv6 has not been implemented across the Internet there are some places of the network in which IPv6 packets will have to travel through IPv4 nodes in order to reach their IPv6 destinations. A node must encapsulate these IPv6 packets within an IPv4 packet so that IPv4 nodes will be able to route them. The tunnel device handles this encapsulation, hiding it from the Forwarding Engine. The tunnel device also decapsulates any IPv4 packets that it receives and passes the embedded IPv6 packet up to the Forwarding Engine.

A.2 Device Drivers

The driver handles the link-specific aspect of transmission and receive. The AIPv6 drivers run in the user-space of the Linux OS and utilizes Ethernet sockets to gain access to Ethernet frames. It supports the Ethernet driver interface in the event that, in the future, the node software is moved into the Linux kernel or a different link protocol is used to transmit data between computers.

Like the Ethernet driver in Linux, the AIPv6 driver module supports two modes, sending and receiving. The driver is always in the receive mode listening for packets that have the Ethernet protocol type specified by the device object that claims ownership over the driver. This has its own thread, and upon receiving an Ethernet frame the thread invokes the receive method of the driver's owner. This receive mode operates as its own thread.

[1] The loopback address in IPv6 is *::1*.

Sending can be requested at any time by a device. This mode does not operate as its own thread, and instead is executed by the thread of caller.

A.3 Buffer Management

A fundamental problem with processing a network packet centers around the numerous addition and removal of headers that occur to a packet as it moves through the protocol stack. Each layer, from the Ethernet layer to the IP layer to the TCP layer and so on, prepend a header on packet data so that it can be processed by the respective protocol. If done naively, prepending and removing network headers can drastically reduce the performance of a node. For instance if the whole packet needs to be recopied every time the node adds or removes a header from it then the data copying can consume the majority of packet processing time.

Both of the standard freeware network implementations, NetBSD and Linux implement somewhat different solutions to the above problem. The NetBSD implementation uses a linked list of buffers, called a chain of Mbufs, to hold the packet's data. A buffer can either be a header or a packet's payload. When a network header needs to be added to the packet, the networking code adds an Mbuf to the beginning of the chain. This type of buffer structure places a requirement on the Ethernet drivers that they understand how to move a chain of Mbufs from the kernel address space to the memory on the Ethernet card.

Linux uses a derivative of Mbufs, called sk_buffs, as the cornerstone of its buffer management strategy. Instead of allocating a chain of buffer structures, the Linux kernel allocates one sk_buff for each packet that passes through its protocol stack. Each sk_buff contains the maximum amount of space for additional headers that a packet could possibly use while it undergoes processing. For instance when the kernel creates a packet at the transport layer, the sk_buff it uses will also have space for the IP and Ethernet headers that lower layers will need to prepend to the packet before it can be sent out onto the network. In addition, if there is the possibility that the packet might need to tunnel through a medium different than IP, then the sk_buff will also have space for the tunneling header. Even though not all the memory in an sk_buff will be used by the kernel for processing the common packet, the simplicity of its implementation, along with the characteristic that drivers do not need to support chained buffers tends to make it a better choice over Mbufs.

Active Node's NetBuffs

This implementation uses a buffer management strategy similar to the sk_buff strategy used in Linux. The primary reason for emulating the Linux buffer management strategy is that none of the Linux system calls can handle chained buffers. I have implemented a class named `NetBuff` that serves as a layer of abstraction above the byte array which is passed to the node from an Ethernet socket. The `NetBuff` class stores an index to the first byte where valid non-IPv6 data (such as Ethernet or tunneling fields) should be placed in the byte array, and also stores another index to the first byte where valid IPv6 data should be placed. Classes which use the `NetBuff` class have access only to the IPv6 area of the byte array.

Figure A-1 shows the NetBuff's layout. The start pointer tells the Ethernet methods where the Ethernet frame begins in the byte array. All bytes with an index less than start is invalid.

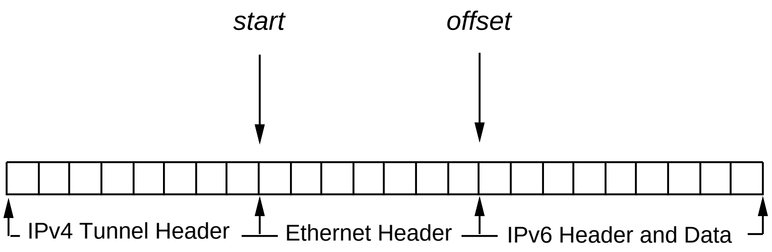

Figure A-1: The byte array hidden by the `NetBuff` class.

Passing pointers to specific values in the IPv6 area of a `NetBuff` is difficult in Java because Java does not support direct addressing of memory. Thus, to facilitate passing indexed data, I have implemented an IndexedBuff class that contains a reference to a `NetBuff` object, and index values of the beginning and end points of the indexed data.

Appendix B

Format of Option and Payload

This appendix chapter describes the format of the marshaled option and demand payload.

The fields that the option and payload have in common are:

- 128 bit identifier that identifies a code class. In the present implementation this identifier is a randomly chosen number, but in future versions the identifier could be an MD5 fingerprint that authenticates every code class.
- Active Type field. Serves to differentiate the different types of formats that might exist in a marshaled option and in the system payload.

B.1 IPv6 Option

IPv6 requires every option to have a type value. The two highest order bits of this value instruct all IPv6 nodes whether to skip the option and continue processing the packet if they do not understand the option. The third highest order bit notifies nodes that whether the contents of the option may change en-route. End to end security protocols need this information when calculating their checksums.

Every option also has a length field. The value of this field depends on the number of state values embedded within it.

To ensure proper byte alignment within headers, the IPv6 specification specifies padding fields. These padding fields are defined as options. IPv6 supports two types, the Pad1 option and the PadN option. The Pad1 option has a length of one and a value equal to zero. The PadN option has a value equal to one for its first field, and a value equal to the length of the padding in the second field. A stream of n zeroes, where n equals the value of the length field, then follows.

B.2 The Marshaled Option

The Marshaled Option contains the state values for programming code identified by the identifier field. It has a value equal to 54. IPv6 nodes must ignore this option if they do not understand it. Figure B-1 depicts the layout of this option. The option has an Active type value equal to one. It also has a 128 bit address field which holds the IP address of the last Active node that processed this option. I have inserted this field in order to support ANTS-style demand-loading in which a node requests code from its closest upstream neighbor.

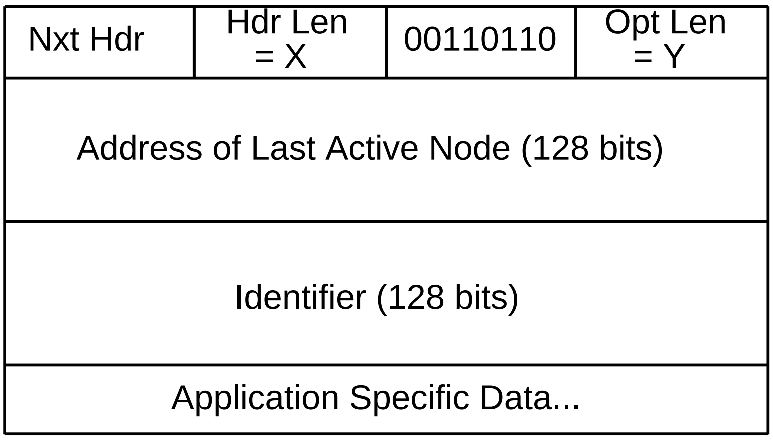

Figure B-1: The marshaled option.

In order to reduce the performance cost of demultiplexing a packet to the Active processor or to the default IPv6 routine, the Marshaled option must be the first option in a hop-by-hop options header, and that there must not be any padding between the Marshaled option and the beginning of the header. Combining these requirements with the IPv6 requirement that the hop-by-hop options header be the first extension header in a packet, reduces the demultiplexing cost to the time required to check the values of two bytes, the next header byte in the IPv6 header, and the option type byte in the hop-by-hop options header.

The risk in using an option in the hop-by-hop options header to transport state values is that IPv6 specification forbids IPv6 nodes from fragmenting the hop-by-hop options header. Therefore, even if a packet is fragmented, the state values in a marshaled option can be used to execute code at every node. The problem though is that the code for a packet might be executed more than once as the packet travels through each node because the code will be executed for each fragment. This can be a benefit or drawback depending on an application's requirements. Since fragmentation can only occur at the source node, the best solution is to turn off IP fragmentation at the source node.

B.3 The System Payload

Both a demand request and a demand response will be transported within the same type of payload, the Active payload. The Active payload has a protocol value equal to 89. Since only AIPv6 nodes transmit this payload to and from each other, it does not need to be immediately defined by the IANA. AIPv6 nodes will understand how to process it. The first byte of the payload specifies the protocol value of payload that follows this one. The second byte specifies the length of this payload in units of eight octets. The value of this field excludes the first eight octets. I have reserved the two ensuing bytes for future use by the Active Networking community. There is a proposal to create one Active payload for the whole Active Network community. If this comes to fruition then these two fields will be used to hold general Active Network data.

B.3.1 Demand Request

The demand request payload depicted in Figure B-2 has an Active type value equal to 2. When an AIPv6 node receives a marshaled option for which it does not have the corresponding code it will send a demand request payload to the upstream Active node identified in the Marshaled option. The identifier field specifies the type of code the requesting node needs.

Nxt Hdr	Hdr Len = 3	Reserved	Reserved
Identifier (128 bits)			
Act Type = 2	0	0	0
0	0	0	0

Figure B-2: The demand-load request payload.

As shown in the figure, the demand request payload has a fixed which is less than the maximum size of an option, and could have been implemented as such. In making the demand request a payload I am assuming that, in general, the closest upstream Active node will be the one identified in the Marshaled option. If for some reason a network does not use symmetric routing (a router might fail and all packets need to be rerouted around it), thereby making the route used by the Marshaled option to travel to the requesting node from the upstream Active node different from the route used to travel back to that upstream node, then there is the possibility there might be a closer Active node that potentially could service this request. The likelihood though of a marshaled option, with the same identifier, having actually traveled through that node on its way to another destination seems quite small. The likelihood of a route being changed is also quite small.[1] Thus, the probability of there being a closer Active node than the one specified in the Marshalled option, and that that Active node contained the requested programming, does not justify the performance hit suffered by every IPv6 node that attempts and fails to process this option.

[1] Contrary to popular belief, most packets originating from the same source node usually do not travel different routes to the same destination.

B.3.2 Demand Response

The demand response payload, depicted in Figure B-3 contains the programming code specified by the 128-bit identifier value in the payload. It has an Active type value equal to 3. The four high order bits, named *Total*, in the byte following the type field, indicate how many fragments the code has been divided into for transport over the Internet. The four low order bits, named *Sequence*, in the same byte, indicate the fragment sequence number of the bytecodes in this packet. A value of zero indicates that this is the first fragment, a value of one indicates that this is the second fragment and so on.

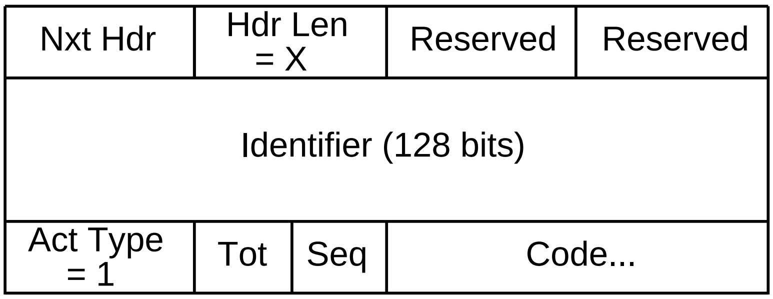

Figure B-3: The demand-load response payload.

Bibliography

[1] Mark B. Abbot and Larry L. Peterson A Language-Based Approach to Protocol Implementation September 23, 1993.

[2] D. Scott Alexander et al. Active Bridging. To appear in *ACM Sigcomm*, 1997.

[3] D. Scott Alexander et al. Active Network Encapsulation Protocol (ANEP) RFC. July 1997.

[4] Ken Arnold, James Gosling. The Java Programming Language. Addison-Wesley Publishing Company, Inc. 1996.

[5] Brian N. Bershad et al. Extensibility, Safety and Performance in the SPIN Operating System. In *Proceedings of the Fifteenth Symposium on Operating Systems Principles*, 1995.

[6] Samrat Bhattacharjee, Kenneth L. Calvert and Ellen W. Zegura. An Architecture for Active Networking 1996.

[7] Andrew D. Birrell and Bruce Jay Nelson. Implementing Remote Procedure Calls. In *ACM Transactions on Computer Systems*, Vol. 2, No. 1, February 1984.

[8] Andrew Birrell, Greg Nelson, Susan Owicki, and Edward Wobber. Network Objects. In *SIGOPS*, December 1993.

[9] David C. Clark. Modularity and Efficiency in Protocol Implementation. RFC 817. July 1982.

[10] David C. Clark. The Structuring of Systems Using Upcalls. In *Proceedings of the Tenth Symposium on Operating Systems Principles*, 1985.

[11] David D. Clark and David L. Tennenhouse. Architectural Considerations for a New Generation of Protocols. *ACM Sigcomm*, 1990.

[12] S.E. Deering and D.R. Cheriton Host Groups: A Multicast Extension to the Internet Protocol RFC 966. December 1985.

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

[14] Peter Druschel and Gaurav Banga. Lazy Receiver Processing (LRP): A Network Sub-

system Architecture for Server Systems. In *USENIX 2nd Symposium on Operating Systems Design and Implementation*, Seattle, Washington, October 1996.

[15] Dawson R. Engler, M. Frans Kaashoek, and James O'Toole Jr. Exokernel: An Operating System Architecture for Application-Level Resource Management. In *Proceedings of the Fifteenth Symposium on Operating Systems Principles*, 1995.

[16] D.C. Feldmeier, A.J. McAuley and J.M. Smith. Protocol Boosters. Submitted to *IEEE JSAC*.

[17] Marc E. Fiuczynski and Brian N. Bershad. An Extensible Protocol Architecture for Application-Specific Networking. In *Usenix Winter Conference*, 1996.

[18] J. Gosling. Java Intermediate Bytecodes. In *SIGPLAN Workshop on Intermediate Representations*, January 1995, AMC.

[19] James Gosling, Bill Joy, Guy Steele. The Java Language Specification. Addison-Wesley Publishing Company, Inc. August 1996.

[20] N. C. Hutchinson and L. L. Peterson. The x-Kernel: An architecture for Implementing Network Protocols. *IEEE Transactions on Software Engineering*, 17(1):64-76, Jan. 1991.

[21] Dave Katz and Randall Atkinson. IPv6 Router Alert Option. Work in Progress. draft-ietf-ipngwg-ipv6-router-alert-01.txt.

[22] Lok Tin Liu, Alan Mainwaring, Chad Yoshikawa. White Paper on Building TCP/IP Active Messages.

[23] Peter Madany, Susan Keohan, Douglas Kramer, and Tom Saulpaugh. JAVAOS: A Standalone Java Environment (Whitepaper).

[24] Jeffrey C. Mogul and Richard F. Rashid. The Packet Filter: An Efficient Mechanism for User-level Network Code. In *Proceedings of the Eleventh ACM Symposium on Operating Systems Principles* , 1987.

[25] Allen B. Montz et al. Scout: A Communications-Oriented Operating System (Whitepaper). June 17, 1994.

[26] Perkins, C.E. IP Mobility Support RFC 2002. October 1996.

[27] J.H. Saltzer, D.P. Reed and D.D. Clark. End-To-End Arguments in System Design. *ACM Transactions on Computer Systems*, Vol. 2 No. 4, November 1984, Pages 277-288.

[28] J.S. Shapiro, S.J. Muir, J.M. Smith and D.J. Farber Operating System Support for Active Networks

[29] J.M. Smith et al. SwitchWare: Accelerating Network Evolution (White Paper). June 1996.

[30] W. Richard Stevens. TCP/IP Illustrated, Volume 1, The Protocols. Addison-Wesley Publishing Company, 1994.

[31] David L. Tennenhouse and David J. Wetherall. Towards an Active Network Architecture. In *Multimedia Computing and Networking*, January 1996.

[32] C.A. Thekkath, T.D. Nguyen, Evelyn Moy and E.D. Lazowska. Implementing Network Protocols at User Level. *ACM Sigcomm*, 1993.

[33] Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham. Efficient Software-Based Fault Isolation. In *Proceedings of the Fourteenth Annual Symposium on Operating Systems Principles*, 1993.

[34] Deborah A. Wallach, Dawson R. Engler and M. Frans Kaashoek. ASHs: Application-Specific Handlers for High-Performance Messaging. *ACM Sigcomm*, 1996.

[35] David Wetherall. Safety Mechanisms for Mobile Code. Area Exam Paper, MIT Laboratory for Computer Science, November 1995.

[36] David Wetherall, John Guttag and David Tennenhouse. ANTS: A Toolkit for Building and Dynamically Deploying Network Protocols. Submitted to *IEEE OPENARCH'98*, 1998.

[37] David J. Wetherall and David L. Tennenhouse. The Active IP Option. In *Proceedings of the Seventh ACM SIGOPS European Workshop*, Sept.1996.

[38] Gary R. Wright and W. Richard Stevens. TCP/IP Illustrated, Volume 2, The Implementation. Addison-Wesley Publishing Company, 1995.