Service Introduction in an Active Network

by

David J. Wetherall

E.E. Massachusetts Institute of Technology (1995)
S.M. Massachusetts Institute of Technology (1994)
B.E. University of Western Australia (1989)

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

Doctor of Philosophy

at the

Massachusetts Institute of Technology

February 1999

Author ....
Department of Electrical Engineering and Computer Science
November 2, 1998

Certified by ....
John V. Guttag
Professor, Computer Science
Thesis Supervisor

Certified by ....
David L. Tennenhouse
Senior Research Scientist, Computer Science
Thesis Supervisor

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

Service Introduction in an Active Network
by
David J. Wetherall

E.E. Massachusetts Institute of Technology (1995)
S.M. Massachusetts Institute of Technology (1994)
B.E. University of Western Australia (1989)

Submitted to the Department of Electrical Engineering and Computer Science
on November 2, 1998, in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

Abstract

In today's networks, the evolution of wide-area services is constrained by standardization and compatibility concerns. The result is that the introduction of a new service occurs much more slowly than the emergence of new applications and technologies that benefit from it. To ameliorate this problem, an *active network* exploits mobile code and programmable infrastructure to provide rapid and specialized service introduction. A viable active network has the potential to change the way network protocols are designed and used, stimulating innovation and hastening the arrival of new functionality. There are, however, a number of challenges that must be overcome in the design of an active network. Chief among them are how to express new services as network programs, and how to execute these programs efficiently and securely.

In this thesis, I present a novel network architecture, ANTS, that tackles these challenges, and describe its prototype implementation in the form of a Java-based toolkit. The main finding of this research is that ANTS is able to introduce new services readily and at a reasonable cost. Experiments with multicast and Web caching services provide evidence that new services can be constructed despite a restricted programming model and constraints such as a network that is only partially active. Measurements of the toolkit show that the forwarding mechanism requires little processing beyond that of IP, such that even the user-level Java toolkit is able to run at 10 Mbps Ethernet rates. Moreover, ANTS achieves this efficiency while raising few new security concerns compared to IP, the most significant of which is that the global resource consumption of a service must be certified as acceptable. I conclude that the active network approach shows great promise as a means of promoting network evolution.

Keywords: active networks, Internet, protocol evolution, mobile code

Thesis Supervisor: John V. Guttag
Title: Professor, Computer Science

Thesis Supervisor: David L. Tennenhouse
Title: Senior Research Scientist, Computer Science

Acknowledgments

Katrin, my wife, provided the support and encouragement that saw me through and for which I will always be grateful.

David Tennenhouse and John Guttag, my supervisors, proved to be a most effective team, helping me to strike out in a new direction and arrive safely at a destination. Their guidance has helped me to start a career, rather than finish a thesis.

Vanu Bose, my friend, did more than most to maintain a jovial atmosphere as we paced each other in these last years. (Wait, maybe that's why it took so long?)

Members of the Active Networks project and Software Devices and Systems group here at MIT helped me turn ideas into a working system with their suggestions, questions and (most useful of all) code. In particular I would like to single out Jon Santos, and thank Ulana Legedza, Dave Murphy, Steve Garland, Erik Nygren and Li Lehman.

Frans Kaashoek and Butler Lampson, my readers, were a source of valuable and constructive feedback that has improved this work. I wish I had sought them out earlier.

I am also grateful to the users of ANTS and members of the wider active network community, especially Larry Peterson and Jonathan Smith. Releasing ANTS was a very positive experience that helped to drive my research forward.

This work was supported by DARPA, monitored by the Office of Naval Research under contract No. N66001-96-C-8522, and by seed funding from Sun Microsystems Inc.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In today's networks, the rate of change is restricted by standardization and compatibility concerns. The result is that the introduction of new services occurs much more slowly than the emergence of new applications and technologies that benefit from new services. To ameliorate this problem, an *active network* exploits programmable infrastructure to provide rapid and specialized service introduction.

A viable active network has the potential to change the way network protocols are designed and used, stimulating innovation and hastening the arrival of new functionality. There are, however, many challenges in the design of an active network, among them how to express new services as network programs, and how to execute these programs efficiently and securely. There are also opportunities that exist in an active network but not a conventional one, in particular the availability of computation within the network.

In this thesis, I present a novel network architecture, ANTS, that tackles these challenges and takes advantage of these opportunities. Network programming in ANTS is based on *capsules*, special packets that combine data with a customized forwarding routine. The type of forwarding is selected by end user software when a capsule is injected into the network. The corresponding forwarding routine is transferred along with the data as the capsule passes through the network, and it is executed at programmable routers along the forwarding path.

The remainder of the introduction is organized as follows. In the first three sections, I argue the need for network infrastructure in which new services can be introduced readily and discuss how the Internet fails to provide the requisite flexibility. The next section describes how the ANTS architecture increases flexibility, gives examples of the kind of new services it is intended to support, and discusses architectural ramifications. In the final three sections, I summarize the method, contributions, and organization of this work.

1.1 The Need for Flexible Infrastructure

Networking as a field is characterized by rapid change. New technologies and new applications have emerged quickly for at least a decade, and this trend shows no signs of abating. In turn, new ways of using the network often benefit from new services within the network that enhance functionality or improve performance to better accommodate the new modes of use. Examples of ongoing changes in today's Internet illustrate this interaction:

- Multi-party applications benefit from multicast services because they greatly reduce the bandwidth needed to communicate from one host to many others. For example, conferencing with *wb* [Floyd *et al.*, 1997] is implemented using IP Multicast [Deering, 1989].
- Real-time applications benefit from resource reservation and service differentiation mechanisms because they assist the delivery of time-sensitive data in a timely fashion. For example, the performance of Internet telephony applications could be improved by using mechanisms such as RSVP [Braden *et al.*, 1997], Integrated services and Differentiated services[1].
- Laptops benefit from host mobility services such as Mobile IP [Perkins, 1996]. These services allow a portable computer to be connected to the network at different sites without the need to reconfigure address information.

It is not surprising that the way a network is used affects the design of its services. In communication networks, two factors combine to translate this interaction into the need for flexible network services.

First, predicting the popular modes of network use is a challenging task. Both email and the Web are benefits of the Internet that were not anticipated [Leiner *et al.*, 1997]. No-one knows what new services will emerge in the next decade, though it is clear that the use of the Internet is diversifying. Applications such as distance learning, electronic commerce, Internet telephony, and customized entertainment broadcasting all have the potential to explode in usage and impact network services in the process. In such a dynamic world it is not practical to design an optimum network service, nor is it realistic to limit users to a "lowest common denominator" service.

Second, the utility of an internetwork is related to its connectivity, with larger networks generally considered to be the most valuable. These are precisely the networks that cannot easily be replaced because the existing investment is large and must be preserved. Consequently, large networks must be upgraded, preferably in a fashion that is compatible with existing services. This process is continual since large networks tend to persist for a long time and usage patterns change over that period.

Given this interaction, the ability to accommodate new services within the infrastructure is of prime importance. This fact is receiving increasing recognition. The Next Generation Internet (NGI) initiative explicitly recognizes the need for infrastructure that is able to accommodate new services as they emerge [CRA, 1997]. In the Internet today, new switching technologies such as Multi-Protocol Label Swapping (MPLS)[2] are being explored to simultaneously boost raw forwarding performance and ease the introduction of new routing services. Finally, in telecommunications networks, the Intelligent Network architecture[3] is

[1]Refer to the IETF Integrated Services and Differentiated Services Working Groups for current details. Note that in this dissertation, by convention, footnotes are used to refer to unpublished material such as Web pages, and references are used for otherwise published material.

[2]Refer to the IETF MPLS Working Group for current details.

[3]See, for example, *IEEE Communications Special Issue: Marching Toward the Global Intelligent Network*, Vol. 31, No. 3,March 1993.

being standardized to speed the development of value-added services such as 1-800 numbers, while the Open Signaling community[4] is incorporating programmability in the control plane as a means to express new services.

In summary, to be most effective, network services need to be flexible. It is impractical to either design an optimum service based on usage, or to construct new networks as usage patterns change.

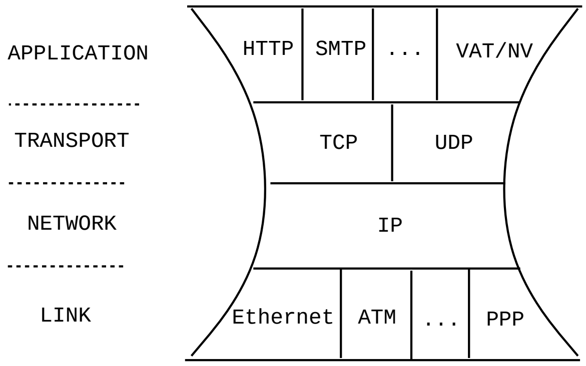

Figure 1-1: The Internet Protocol Stack as an Hourglass Model

1.2 Today's Networks Lack Flexibility

Despite the need for flexibility, the process of changing network services in the Internet is lengthy and difficult. As an illustration of the magnitude of these difficulties, none of the example changes described in Section 1.1 have been fully deployed, despite clear-cut needs that have often been recognized for years. Recent debate about IPv6 illustrates these difficulties [Koprowski, 1998].

To explore network flexibility, I describe different types of change according to their location in a layered protocol stack, the standard organizational model of network functionality. To be concrete, the discussion is based on the Internet protocol layers, as abstracted in Figure 1-1. The Internet is arguably the most successful example of a modern and general purpose computer network. However, much of the discussion applies to other layered systems as well.

Change at the link and application layers can occur relatively quickly because it is limited in scope. The effects of new technologies are limited by physical scope, and so can be introduced by upgrading the affected area. This is demonstrated by the rapid emergence of technologies such as Fast Ethernet, soon to be joined by Gigabit Ethernet. Similarly, the effects of new applications are confined to their participants, and (unlike TCP and UDP) application protocols are typically distributed as part of the application itself and run in user space. Thus Web browsers and servers, and multi-player games such as Quake, have been deployed as quickly as people have been willing to use them.

[4]See http://comet.ctr.columbia.edu/opensig/ for more information.

Change at the network and transport layers are the bottleneck because they are the basis for interoperability between different link layer technologies and higher layer applications. As such, it is important that they are stable and widely available to provide continuity of service. To preserve these properties, each change must first proceed through a lengthy standardization process, typically lasting years. This must then be followed by manual deployment. Since the implementation of IP is spread across many routers, and that of TCP/IP across many operating systems, some changes might not be effective for a period of years.

The difficulty of effecting network and transport layer change has further consequences. It leads to requirements for backwards-compatibility and incremental deployment.

Backwards-compatibility is necessary to provide connectivity between new and old service areas since it is not possible to upgrade all portions of the network simultaneously. This complicates protocol design. The net result is an emphasis on minimal changes that serve general purpose needs, on one all-purpose protocol rather than many special-purpose protocols. In the case of TCP, O'Malley argues that this approach is ineffective, and suggests that a versioning mechanism support evolution [O'Malley and Peterson, 1991]. Regardless of the mechanism, however, it is clear that the current system does not match the increasing diversity with which the Internet is used and does not encourage the experimentation necessary for innovation.

Incremental deployment is necessary to allow a new service region to grow until the service is available across the entire network. It is enabled by localizing or otherwise minimizing the scope of the change in terms of network nodes. This occurs naturally for some changes in router and end-system processing because their effects are transparent to other nodes, for example, RED gateways [Floyd and Jacobson, 1993] and TCP refinements to congestion control [Stevens, 1997]. It can be made to occur for other changes that are backwards-compatible, such as Mobile IP [Perkins, 1996]. In the case of TCP, additional processing that is not transparent can be negotiated between the two end-systems during connection-establishment, as is done for TCP long-fat pipe and selective-acknowledgment extensions [Jacobson *et al.*, 1992, Mathis *et al.*, 1996]. In the case of IP, some changes in processing are required at only a pair of nodes, such as IP security extensions [Atkinson, 1995]. Services that can be made to correspond to these situations can be deployed with relatively few operational constraints.

Unfortunately, some new services are not easily localized at the network layer. In this case, the only reasonable deployment mechanism is to implement the new service as an overlay, that is, as a layer on top of the old network service instead of in place of it. This strategy is currently used to provide IP multicast service encapsulated within IP to hosts that participate in the MBONE[5]. Similarly, it is used to provide IPv6 service on the 6BONE[6].

Overlays used in this manner can be a temporary step towards full deployment. They are useful for experimentation and early provisioning of a new service because, by their nature, they isolate it from the old network service. They are not a long-term solution because overlays duplicate mechanism and incur performance overheads, often beyond that of packet encapsulation. In the case of the MBONE, for example, multiple copies of a

[5] Refer to the MBONE Deployment Working Group of the IETF for current details.
[6] Refer to the Next Generation Transition Working Group of the IETF for current details.

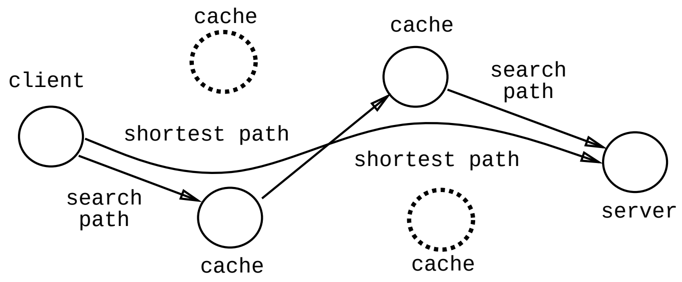

Figure 1-2: Combining Web Cache Lookup with Network Routing

message will traverse a link — the very situation multicast seeks to avoid — when the overlay topology does not match the underlying topology. Further, services such as real-time cannot be deployed as an overlay effectively because the underlying IP layer does not provide the required functionality, in this case bandwidth reservation. In this case, a new service will be only partially-available, or simply unavailable, over the wide-area until there is a significant installed base.

In summary, the current process is manual, slow and complicated because of the need for consensus and concerns for backwards-compatibility that are necessary to preserve an existing investment. Effectively, only those changes that are localized in scope are able to be incorporated, and the network layer does not explicitly support such localization.

1.3 A Web-Caching Example

Web caching serves as an example of both new modes of use impacting the network infrastructure, and the current difficulty of introducing changes. With the dramatic increase in the popularity of the Web, caching mechanisms are needed to ensure that the growth of Web traffic does not congest the Internet. Early efforts in client-side caching and proxy caching improved client behavior to avoid redundant transfers and redirected queries through a local agent to improve sharing, respectively. Ongoing research is focussed on the sharing of information between caches at different locations in the network infrastructure. The central questions are: how to locate a copy of a document by searching through the caches, and how to propagate documents to caches for future queries.

A key observation ties what would otherwise be a purely application layer function — Web document caching — with the network layer: an important goal is to reduce network load, and to do so effectively requires that cache transfer paths be well-matched to the underlying network topology. Hypothetically, a straightforward and effective approach would be to combine the network routing of query and response with caching as shown in Figure 1-2. Here, as a query travels from client to server, nearby caches are consulted along the way, and as a document travels back to the client, nearby caches are replenished. In terms of network load, this scheme has the desirable property that (modulo cache size limitations) a given document will only traverse each network link once. It can also result in globally cost-effective cache search paths in the same sense that multicast routes based on shortest paths are known to be effective in practice [Doar and Leslie, 1993].

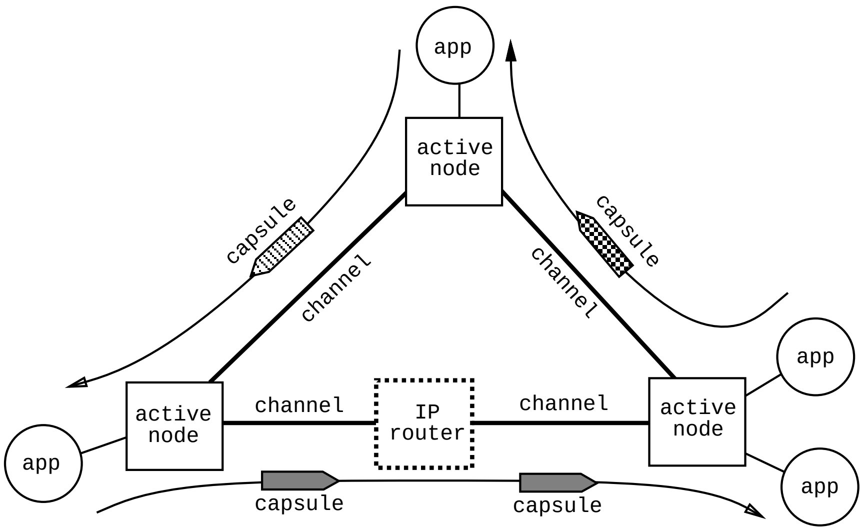

Figure 1-3: Entities in an ANTS active network

Yet this hypothetical service cannot be provided in today's Internet. This is because it requires that forwarding be modified *within* the network, whereas IP provides only a "black-box" service that forwards *across* the network. The same observation applies to any other service that affects forwarding along network paths.

Given this mismatch, other caching systems must approximate this strategy by other means. For example, Squid [Wessels and Claffy, 1998] requires manual configuration of a search path, which is tedious and likely non-optimal, while Adaptive Web Caching [Zhang *et al.*, 1998] uses scoped multicast as a form of indirection to cluster caches, a more dynamic but complex mechanism. Coming at the problem from a different end, a recent spate of commercial products provide network-layer support for Web caching based on "layer four" switching technologies. Cisco's Cache Director[7], for example, attaches to a router and intercepts Web traffic that would otherwise be forwarded for cache processing. In this manner, selected routers can act as cache nodes within the network in a manner that is transparent to their clients.

All of these approaches incur costs, such as additional bandwidth or layers, that might reduce performance compared to the service that retains queries within the network layer. None of them capture the simplicity and power of the service that combines routing and caching directly.

1.4 Introducing New Services with ANTS

In contrast, the hypothetical Web caching service can be introduced readily with ANTS, the network architecture presented in this dissertation. ANTS is an *active network* [Tennenhouse and Wetherall, 1996] that can be re-programmed to provide services that were not

[7]See http://www.cisco.com/warp/public/751/cache/ for product information.

anticipated when it was originally designed. The key hypothesis explored in this work is that, compared to a conventional network, an active network can facilitate the rapid and automatic introduction of new services at a reasonable cost in terms of performance and security overheads. I introduce ANTS by providing an overview of its functionality, describing the kinds of service it is able to support, and discussing the architectural consequences of such an active network.

1.4.1 Overview of ANTS

The entities of an ANTS network are shown in Figure 1-3. Applications use the network by sending and receiving special types of packets called *capsules* via a programmable router referred to as an *active node*. Each active node is connected to its neighbors, some of which can be conventional IP-style routers, by link layer channels. Compatibility with existing routers supports incremental deployment of the architecture itself, a necessary consideration to bootstrap an active network in the Internet. More fundamentally, it suits the heterogeneity of large networks, in which some nodes such as backbone routers might be computationally limited, while other nodes near the edges of the network might be bandwidth constrained.

The innovative properties of an ANTS network stem from the interaction of capsules and active nodes. I describe the architecture in terms of three components:

- the capsule programming model with which network services are customized;
- the *demand pull* code distribution system that helps to realize this model in a manner compatible with high performance; and
- the active node operating system that restricts the interactions between different programs so that they can safely be run in the network simultaneously.

Capsule Programming Model

In ANTS, network programming is based on the *capsule* model. Capsules are special packets that encapsulate packet data with a customized forwarding routine. The type of forwarding is selected by end user software when a capsule is injected into the network. The corresponding forwarding routine is transferred by using mobile code techniques along with the data as the capsule passes through the network. The routine is executed at each active node that is encountered along the forwarding path. At conventional nodes, IP-style forwarding is used.

The kinds of custom forwarding routine that can be constructed depend on the capabilities of the active node. ANTS provides API calls to query the node environment, manipulate a *soft-store* of application defined objects that are held for a short time, and route capsules towards other nodes. Since this programmability exists within the network layer, new services can interact with routing and topology information, packet loss events, and link characteristics. None of this information is directly available to overlay solutions.

This model provides one answer to the consensus and deployment problems experienced today. Deployment of forwarding routines, and hence the service they provide, is fully automatic. It is part of the network architecture rather than an external task. Because the new forwarding routines affect only the capsules with which they are explicitly associated, different users can deploy different forwarding routines, thereby introducing different services, without conflict or concern for backwards-compatibility. This eases the consensus problem, since only those users participating in a service at a given moment need agree on its definition. Rather than standardize on one service and then implement it, ANTS allows many variations of a service to be tried. The standard version is simply the most popular one over time.

Demand Pull Code Distribution

To improve the performance of the capsule model in practice, forwarding routine code is cached at active nodes rather than transferred with every capsule. This technique is valuable because the traffic flows generated by higher level connections imply that the same forwarding routine is likely to be invoked many times at a given active node. In this case, separating the transfer of code and data improves performance because fewer network resources are expended transferring code and caching provides a basis for runtime optimizations that allow the forwarding routines to execute more efficiently.

To distribute the code within the network before it is cached, ANTS provides an automatic *demand pull* code distribution system. It is designed for forwarding routines that are compact. It transfers the code that implements a new service to nodes along the path that a capsule using the service follows. The scheme takes advantage of the small code size to provide lightweight but unreliable transfer: code distribution will either succeed rapidly, in which case the interruption to forwarding should be transparent to applications, or occasionally fail, in which case the capsule that triggered the load mechanism will appear to have been lost and must be recovered by applications using the network in the normal manner.

Active Node Operating System

To ensure that new service programs are run securely within the network, ANTS provides an active node operating system. It is specially designed, rather than built on top of a conventional operating system such as Unix, because its mechanisms must run efficiently for tasks that complete at the capsule forwarding rate. Further, because of the scale of the Internet and its composition as a collection of autonomous systems, the node operating system is biased towards mechanisms that treat forwarding routines as untrusted code. Unlike authentication and authorization schemes, these mechanisms allow code to come from any source and prevent accidental as well as malicious errors. Ideally, this approach should make it possible for any service code to be run within the network, and if poorly designed do no harm to the users of other services.

The node operating system provides protection mechanisms that isolate the code of different services and prevent it from corrupting the network. These mechanisms are built on top of safe evaluation techniques (the prototype ANTS implementation is written in Java) that are assumed to provide efficient type and memory safety. To extend this basic level of safety to protection between services, the capsule model is used to simplify program interactions as follows. When a capsule is injected into the network it is marked with a customized forwarding routine that is selected by the sender. Effectively, the sender is authorized to determine the handling of that capsule. By extension, within the network, only the selected forwarding routine (and related ones that form the new service) should be used to determine the path of that capsule. In ANTS, this form of isolation is developed into a *fingerprint-based* protection model. This means that it is not possible, for example, to construct a service that hunts for and discards packets belonging to a particular user.

Finally, the node operating system provides resource management mechanisms that prevent the operation of one service from unreasonably interfering with another. Forwarding routines are monitored as they run at a node to prevent them from running too long, and infinite loops across nodes are broken in the same manner as the Internet today. A more difficult problem is to prevent the global resource consumption of one service from interfering with others, for example, by saturating a link with a poorly written program that ping-pongs capsules across it. In ANTS, these problems are prevented by requiring that service code be certified by a trusted authority before it can be freely executed. This mechanism differs from the remainder of the node operating system, which is able to work with untrusted code, and is an important area for further research. Nonetheless, the current system still allows new services to be deployed much more quickly than in the Internet.

1.4.2 Potential New Services

Because capsules specify a customized forwarding routine, ANTS services are typically variations on IP forwarding. Unlike services that work above IP, they can directly take advantage of topology and load information to express novel types of routing, flow setup, congestion handling, measurement, and other network layer functions.

The following hypothetical examples are intended to show the kinds of new services that can be readily introduced with ANTS, but are difficult to provide in the Internet today. None of these services can be implemented efficiently by users above the network layer because they rely on network layer state. Further, since none of these services are transparent to network users, they cannot be deployed by individual router vendors without concern for standardization.

- Forwarding support for reliable multicast. Many reliable multicast schemes require that one sender manage the error control channels of multiple receivers. To do this quickly and efficiently when there are many receivers, network support might be used for many-to-one aggregation as well as partial multicasting. Recent proposals for such mechanisms include [Papadopoulos *et al.*, 1998], [Lehman *et al.*, 1998], and [Levine and Garcia-Luna-Aceves, 1997], as well as Cisco's PGM[8].

- Explicit Congestion Notification. By allowing routers to notify hosts of impending congestion, it is possible to avoid unnecessary packet drops in short or delay-sensitive connections [Floyd, 1994, Ramakrishnan and Floyd, 1999].

- QOS route establishment. Current reservation mechanisms such as RSVP [Braden *et al.*, 1997] are based on shortest paths. To achieve a desired bandwidth, latency,

[8]Internet Draft, *PGM Reliable Transport Protocol Specification*, D. Farinacci et al., January 1998.

or other quality-of-service target, however, alternative paths may be indicated. A scheme for constructing them is described in [Zappala *et al.*, 1997].

- QOS forwarding. Once a reservation is made along a network path, it is desirable that traffic follow that path. This may not occur with IP routing, since each packet is independently forwarded along the current shortest path. To address this mismatch, new forwarding services, for instance the combination of MPLS and RSVP[9], must be defined.

- Multipath routing and forwarding. Internet routing currently makes use of a single shortest path. Multipath schemes such as [Chen *et al.*, 1998] make use of multiple disjoint paths to improve available bandwidth or reliability.

- Path MTU discovery. Because packet size is known to have a significant effect on performance, it is recommended that modern TCP implementations perform path Maximum Transmission Unit (MTU) discovery [McCann *et al.*, 1996, Mogul and Deering, 1990]. This is currently accomplished in an *ad hoc* manner that triggers error messages. A better approach may be to combine path MTU queries with connection setup.

- Anycasting with application metrics. Anycasting is a service that delivers a message to the "closest" member of a group [Partridge *et al.*, 1993]. While the traditional metric is routing distance, other metrics, such as a random member or the closest two members, can be appropriate depending on the application.

There are of course new services that the ANTS architecture is not designed to support. These fall roughly into three categories, each of which can be tackled in a complementary manner. First, ANTS deliberately isolates different services within the network. This is because each user is able to select their own service and without isolation complicated dependencies would arise. This means that processing which cuts across many services, such as firewalls or RED gateways, is not well supported. Second, the code distribution mechanism is designed for forwarding routines that are small; a limit of 16 KB is used in the prototype. Without extensions, it is not an effective means of transferring routines that are large compared to the packet size. Third, assumptions that are widespread, such as the form of the addresses that are encoded as part of all capsules, are not easy to change because they are not readily localized.

1.4.3 Architectural Ramifications

Active networks are fundamentally different than the networks in use today. Traditional protocols rely on prior agreement of message processing between communicating parties, typically with sharp distinction between the roles of intermediate and end-systems. Instead, an active network relies on prior agreement of a computational model. Within this model, many forms of distributed processing that blur the roles of router and host can be expressed to implement different services.

The ANTS capsule model has a number of architectural ramifications. First, it emphasizes innovation by providing an automatic means of upgrading remote parts of the network,

[9]Internet Draft, *Use of Label Switching With RSVP*, B. Davie et. al, March 1998.

rather than relying on backwards-compatibility because change must be effected manually. This provides a clean and systematic means of network evolution because changes in network handling are explicit. In contrast, Internet evolution, has resulted in IP being treated as a packet format rather than a protocol definition in the sense that any processing that is compatible with its existing operation can be implemented. Such implicit change sometimes has unfortunate side-effects, as is the case for Network Address Translation (NAT) boxes[10].

Second, the capsule model encourages the use of many special-purpose protocols that are tailored to applications, rather than one general-purpose service that targets the "lowest common denominator". One way of viewing these specialized network services is as part of the application that has been "pushed" into the network infrastructure. Two arguments suggest that leveraging computation in the network in this manner might ultimately *improve* performance, despite the overhead of supporting multiple services at active nodes.

To begin, efficient processing at a network node may depend on the application, the local environment, or a combination of both. With the increasing diversity of networked applications and environments, it is difficult to argue that a single forwarding routine will provide the best service in all cases. A number of examples illustrate this:

- Slow start TCP is known to interact poorly with lossy media because loss is interpreted as congestion. To compensate for this, the Snoop-TCP protocol augments forwarding at wireless basestations to transparently caching and retransmitting TCP packets and their acknowledgments as necessary [Balakrishnan *et al.*, 1996]. This has been shown to improve performance by as much as 100% to 200%.
- Real-time transfers can benefit from different handling in times of congestion. A recent evaluation of drop priority for layered video [Bajaj *et al.*, 1998] found a 50% increase in system utility compared to the current end-to-end protocol in widespread use, RLM [McCanne *et al.*, 1996]. Similarly, a study of the transfer of MPEG-encoded video found that selective discard schemes (which understand dependencies between I, P, and B frames) improved the useful throughput in times of congestion [Bhattacharjee *et al.*, 1996].

Further, the ability to alter the location of application processing by shipping code may improve performance by interposing computation with communication. This observation is the basis for earlier studies of programmable RPC [Stamos and Gifford, 1990, Partridge, 1992]. Traditional RPC offers a fixed set of interfaces that, by definition, cannot suit the needs of all users. By transferring and evaluating programs that use these interfaces, a single customized RPC that combines what would otherwise be multiple RPCs can be synthesized. The result is a savings of both bandwidth and latency, as multiple round-trip exchanges are collapsed into a single exchange. A number of examples show how these arguments apply to the network infrastructure:

- IP multicasting reduces the aggregate network bandwidth compared to multiple unicasting between end-systems because it uses the network topology to advantage.
- Legedza investigated a protocol for caching time-sensitive data within the network infrastructure [Legedza *et al.*, 1998], and found that it was able to significantly reduce server load and bandwidth.

[10] Internet Draft, *Architectural Implications of NATs*, T. Hain, November 1998.

- Legedza investigated an online auction protocol in which bid rejection tasks were delegated to nodes within the network [Wetherall *et al.*, 1998]. As well as improving successful bid throughput, this decomposition decreased client notification delays by placing the required functionality closer to the clients.

1.5 Evaluation of the Hypothesis

The key hypothesis explored in this dissertation is that an active network can introduce new services rapidly and automatically, yet at a reasonable cost when compared to a conventional network architecture. The remainder of this dissertation explores that hypothesis by: defining ANTS; describing its implementation; demonstrating how it can be used to introduce new services; and evaluating the tradeoffs that it makes in order to do so.

To evaluate the hypothesis, several questions must be answered:

- Does the programming model live up to its promise of automatically introducing non-trivial services? Capsule programs must be able to express useful services despite the inherent difficulty of operating at the network layer (where application data is segmented and transfer is unreliable) and constraints such as partially-active networks and a limited node API.

- What is the impact of the model on performance? Supporting the ANTS model might degrade local node performance. This effect must be small enough that users not relying on the model do not suffer, while users making use of the model are able to improve overall performance with new services.

- What is the impact of the model on security? A programmable network might introduce many new security hazards. At a minimum, the potential effects of different services on one another must be well-understood and limited, the network must be robust in the face of malicious or accidental programming errors, and resource consumption must be arbitrated.

To answer these questions, a framework for assessing network properties is needed. The strategy pursued in this research is to facilitate a comparison with conventional networks. The design of ANTS is based on the Internet, arguably the most successful example of a large internetwork for data communications. Its best-effort IP service is enhanced with the capsule model to provide flexibility, rather than seeking to design a superior network from scratch.

1.6 Summary of Contributions

The main finding of this research is that ANTS is able to introduce new services rapidly and at a reasonable cost. Experiments with multicast and Web caching services provide evidence that it is possible to construct and deploy new services with the capsule model. Measurements of the ANTS toolkit show that, compared to IP, the additional mechanism needed to support capsules is lightweight. The prototype is able to run at rates in excess of 1700 capsules/second for small packets and 16 Mbps for large packets, despite the fact that it is largely limited by Java, which is not required by the architecture. Further, ANTS raises few additional security concerns compared to the Internet. Most threats can be handled by active nodes locally and without trusting service code or external parties, with the exception that the global resource consumption of a service must be certified as reasonable by a central authority.

More broadly, I conclude that the active network approach shows great promise as a means of lowering the barriers to innovation, stimulating experimentation, and otherwise hastening the arrival of new services.

I also make the following specific contributions:

- The definition of an *active network* approach that is based on *capsules*, along with a specific architecture, ANTS, that embodies this approach.
- An analysis of ANTS architecture in terms of the combined levels of flexibility, performance and security that it is able to achieve.
- An understanding of how the capsule programming model can be used to express a variety of network services.
- A *demand pull* code distribution protocol that automatically deploys service code along network paths yet is compatible with performance and security goals.
- A *fingerprint-based* protection model that efficiently separates the state of different services within the network in a manner equivalent to conventional protocols and networks today, despite the use of untrusted code.
- A choice of active node API that allows the capsule programming model to be realized efficiently as *extensible packet forwarding*, such that forwarding requires few processing steps beyond those needed for IP.

1.7 Organization

The remainder of this dissertation presents the ANTS architecture, its prototype implementation and use in the form of the ANTS toolkit, and an evaluation of its effectiveness as a network that facilitates the introduction of new services.

The specifics of the ANTS architecture are presented in Chapter 2. This includes details of the capsule programming model, the API available to custom forwarding routines at active nodes, the demand pull code distribution protocol, and the node operating system. Deployment considerations are also discussed.

To experiment with the architecture, a Java-based prototype that runs on PCs was designed and implemented. This is the ANTS toolkit, described in Chapter 3. While such an implementation cannot compete with the hardware-based designs of commercial routers in terms of performance, it is sufficient to study the characteristics of the ANTS programming model, and has sufficient performance to support routers near the edges of the network today, for example, nodes with T1 (1.5 Mbps) and modem line rates. In Chapter 4, the toolkit is used to demonstrate how new services can be introduced in an ANTS network. Two case studies, for multicast and Web caching services, are described.

Based on experiments with the toolkit and characterizations of the architecture, an evaluation of the ANTS is given in Chapter 5. This includes the measurements of the performance of an ANTS node, a description of its security properties compared to the Internet, and an assessment of the type of services that can be supported. Finally, related work is presented in Chapter 6, before concluding remarks and suggestions for future work in Chapter 7.

Chapter 2

An Extensible Internet Architecture

The ANTS architecture adds service extensibility to the Internet with a programming model based on *capsules* and *active nodes*. This chapter contains an overview of the model, followed by a description of the three main components of the architecture that realize it. Each is presented along with a discussion of the tradeoffs that were made during its design to balance the competing concerns of flexibility, performance and security. In the final section, I discuss deployment considerations.

2.1 The ANTS Model of Extensibility

In this thesis, the term *active network* is used to refer to a network in which mobile code programs can be run within the infrastructure to provide customized services [Tennenhouse and Wetherall, 1996]. There are many ways in which this could be arranged. The ANTS model of extensibility combines a flexible network programming model based on *capsules* with its practical implementation as *extensible packet forwarding*. Both are described shortly, after I first clarify some terminology that is used throughout the thesis. The term "service" has been loosely linked to the execution of programs within the network. More precisely:

- *processing* refers to the execution of a forwarding routine at a network node. An example is the execution of the IP forwarding routine at a router.

- *protocol* refers to the behavior that results when a forwarding routine is executed at nodes throughout the network. An example is IP itself, which is the result of executing the IP forwarding routine at all routers.

- *service* refers to the behavior that a client of a protocol obtains by using the protocol. An example is the connectionless datagram service provided to network users by IP.

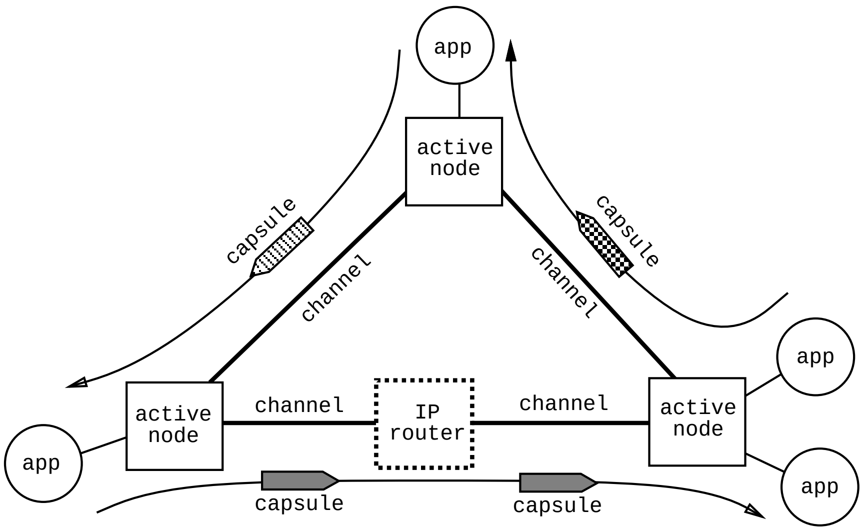

Figure 2-1: Entities in an ANTS active network

2.1.1 Capsule Programming Model

An ANTS network is composed of the entities shown in Figure 2-1. Applications use the network by sending and receiving special packets called capsules via their local active node, which acts as a programmable router. Each active node is connected to its neighbors (only some of which may be active, as described shortly) by link layer channels to form the overall network. The key to obtaining customized service is the selection of different types of capsules by the end-user software, and the subsequent interaction of these capsules with active nodes within the network infrastructure. The application and channel components are not innovative, and are simply modeled on the equivalent components of conventional networks.

In this dissertation, *capsules* are defined to be the specialization of mobile agents to the network layer. The term capsule is intended to convey the encapsulation of data packet and customized forwarding code. Like a mobile agent, the code is executed at each active node that the capsule visits as it navigates the network. Thus, to express a new service it is only necessary to construct a new set of capsule types with associated forwarding code that captures the desired processing.

Telescript [White, 1994] illustrates the agent model. It allows entities to navigate the network under their own control in order to perform tasks such as electronic commerce on behalf of a particular user. Like most distributed programming systems, Telescript builds on services such as reliable transport and stable storage. In contrast, capsules must operate within the constraints of the network layer because their purpose is to provide new network services. Capsule programming must be compatible with: unreliable transport, message duplication and reordering, dynamic and asymmetric routes, and strict size limits. The result is that, compared to most agent systems, capsules are restricted in their capabilities so that lightweight implementation mechanisms can be used.

This model differs from service evolution in the Internet in two respects. Most significantly, it is able to express customized processing along entire network paths, rather than upgrade processing pointwise. This allows new services to be introduced in a single step, without the requirement for incremental deployment or the need to construct an overlay. Further, new services are selected explicitly by choosing the type of capsule, rather than implicitly by router upgrades. This places service control in the hands of end-system software, rather than the router administrator, and lessens the dependence on backwards-compatibility. To be most effective, this model should be coupled with a directory service that maps service names into implementations. In this manner, end-user software can automatically select the latest version of popular protocols.

2.1.2 Extensible Packet Forwarding

The capsule programming model provides much flexibility and is straightforward to understand, but is expensive to implement in the general case. To facilitate a high performance implementation, the model is restricted in two ways. I refer to the resulting implementation as *extensible packet forwarding*, since it is a step up from the IP forwarding mechanism.

First, the forwarding of a capsule is decomposed into transferring code (which may happen relatively infrequently), and invoking code (which may happen relatively frequently). Separating the implementation of forwarding into these components allows each to be implemented with the most appropriate mechanism, and leads to the solution in which code is cached at active nodes to increase performance.

Many mechanisms can be used to transfer code depending on the size and frequency of the transfers. In ANTS, a *demand pull* protocol is used to carry code hop-by-hop as it is needed and along the same path that it would have followed were it carried within each capsule. This scheme is intended for compact forwarding routines (on the order of the capsule size) and traffic patterns for which caching is effective. The transfer is lightweight and unreliable: it will usually succeed rapidly and without interrupting the connectionless service, but can occasionally fail, in which case the capsule appears to have been lost. Larger changes in active node functionality might also be downloaded and cached at active nodes by using some means of initiating a connection that is independent of the stream of capsules. Such larger changes in functionality are not studied in this work, but instead are modeled as part of the heterogeneity of active nodes.

Second, the capsule model is modified to be compatible with a network in which only selected nodes are programmable. Inbetween these active nodes, capsules are forwarded in the same manner as regular IP packets. Rather than organize active nodes into an end-system overlay, however, the ANTS architecture views them as special routers embedded in the overall network topology. This has the advantage of allowing the nodes access to network layer information that can be valuable for the construction of new services: routing and topology information, packet loss events, and link characteristics. None of this information is directly available to the nodes of an overlay.

This restriction is intended to match ANTS to the heterogeneity of modern networks. In the Internet today, high-end routers in the backbone are valued in terms of their packet per second throughput, and the addition of computation may have an adverse impact on performance. Conversely, routers connected to access and wireless links tend not to be capacity limited for economic or technical reasons that restrict the bandwidth of their links.

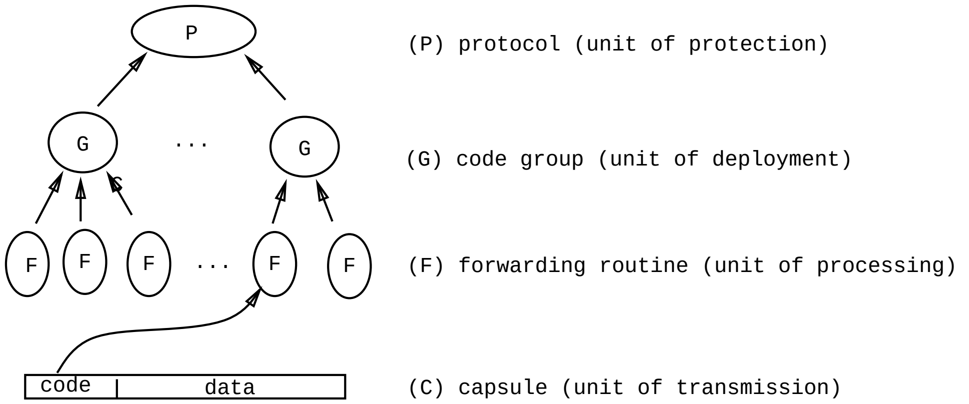

Figure 2-2: Capsule Code Abstractions and their Relationships

Define $HASH(x)$ to be the fingerprint of the sequence x.

Let the forwarding type $f_{i,j,k} = HASH(F(i,j,k))$ be the fingerprint of $F(i,j,k)$, the forwarding code of the ith capsule type of the jth code group of the kth protocol.

Let the code group type $g_{j,k} = HASH(f_{1,j,k} \ldots f_{n,j,k})$ be the fingerprint of a code group comprised of n forwarding routines.

Let the protocol type $p_k = HASH(g_{1,k} \ldots g_{m,k})$ be the fingerprint of a protocol comprised of m code groups.

Then the fingerprint $c_{i,j,k} = HASH(f_{i,j,k}, g_{j,k}, p_k)$ is the type of the ith capsule of the jth code group of the kth protocol.

Figure 2-3: Calculation of Capsule Types

For these routers, the addition of computation in the form of new services may have a beneficial impact on performance if it causes scarce bandwidth to be better utilized.

The result is that the capsule model can be implemented as though each new service were provided by an its own overlay, only an overlay that is embedded within the network layer instead of operating on top of it. The processing routines associated with services are transferred to those active nodes that require them when the services are first used. This dynamically creates the overlay-like structure. Subsequent use of services consists simply of executing the cached routines as capsules arrive at the active node. This requires no more than an extensible packet forwarding mechanism that maps a capsule type to its corresponding forwarding routine and then executes the routine.

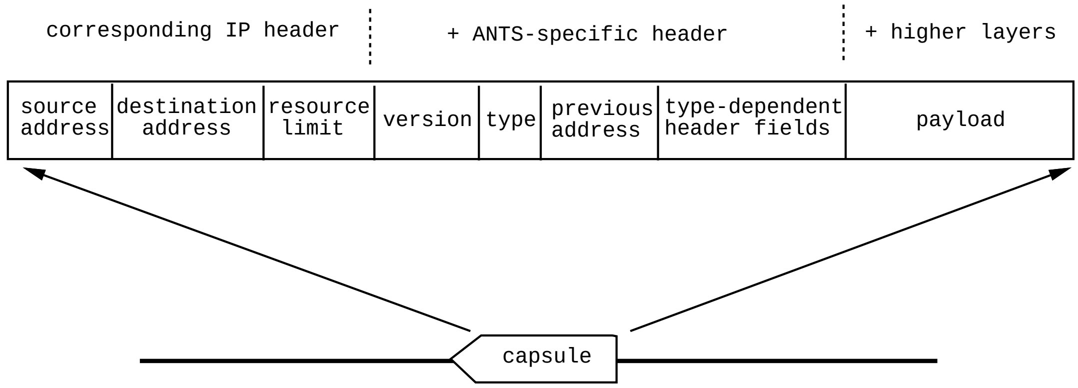

Figure 2-4: Key Features of the Capsule Format

2.1.3 Architectural Components

The architecture that realizes capsules and extensible packet forwarding is composed of three related components. Each of these is described in greater detail in the subsequent sections. Each component also introduces an abstraction of forwarding code as part of its function. The relationship between these abstractions is shown in Figure 2-2. They are described briefly here, rather than as part of each component, because knowledge of their existence is intertwined between all three components.

- A *capsule programming model* is used to express new services in terms of customized *forwarding routines* that are executed at active nodes.
- A *demand pull* code distribution protocol is used to deploy new services within the network infrastructure. Code is transferred in *code groups*, collections of forwarding routines that can refer to each other when executed.
- A *node operating system* is used to execute new services in a safe manner that prevents unwanted interactions between services. The node separates service code in terms of *protocols*, collection of code groups that are allowed to share state within the network. This usage is consistent with the sense in which "protocol" was defined at the start of this chapter.

In adding service extensibility to the Internet, the design of ANTS is geared towards preserving the character of the basic connectionless datagram service provided by IP. As such its components avoid a dependence on connection-oriented schemes, external resource allocation mechanisms, centralized operations that limit robustness, and overheads that limit scalability.

2.2 Capsule Programming Model

I describe the remaining details of the capsule programming model in three parts: the format of capsules, their forwarding rules, and the node API that can be called from within the forwarding routines.

2.2.1 Capsule Format

The architecturally significant features of capsules as they are carried across link-layer channels, for example as Ethernet frames, are sketched in Figure 2-4. This figure shows the type of fields that will always be present in an implementation and their approximate grouping in terms of protocol layers. Each capsule carries the following fields:

- A source and destination address, which are analogous to IPv4 and IPv6 source and destination addresses.
- A resource limit, which is analogous to the IPv4 Time-To-Live (TTL) or IPv6 Hop Count field. Its use is described more fully in the section on the node operating system.
- A version number, which identifies the ANTS version and implies the specifics of the remaining fields that are common to all capsules.
- A type, which identifies the associated forwarding routine, along with its code group and protocol.
- A previous address, which is used to transfer code within the network infrastructure. Its use is described more fully in the section on the code distribution protocol.
- Other header fields, whose number and size varies depending on the type of capsule as indicated by its type field.
- A payload, which contains the higher layer information that is opaque to processing within the network.

The distinction between header fields that are analogous to those of IP and those that are ANTS-specific is made in support of nodes that are not active. At these nodes, a default forwarding service that requires IP field information is provided. It can be implemented with existing IP routers by defining default forwarding to be identical to IP forwarding and defining capsule formats to be an extension of the IP packet format. For example, the ANTS-specific header could be identified as a higher layer protocol with the IPv4 protocol identifier or the IPv6 next header, or as an extension of the IP header with an IP option field. In this manner, IP and ANTS can be combined without introducing another layer of addressing or duplicating mechanism such as that used to prevent forwarding loops.

Of the ANTS-specific headers, the key field is the type, which identifies the forwarding routine, code group and protocol to which the capsule belongs. (Each of these bits of information is needed to process the capsule at an active node, and it is not sufficient to identify only the forwarding routine because one routine can be combined with others into different code groups and protocols.) The requirements for the type are that it be unique, easy to check, and easy to allocate. For these reasons, I have based it on fingerprints of the associated code. In the ANTS toolkit, the MD5 message digest [Rivest, 1992] is used as the fingerprint function.

The algorithm for calculating a capsule type is given in Figure 2-3. A straightforward scheme would be to carry three fingerprints: one of the forwarding routine, one of all forwarding routines in the code group, and one of all forwarding routines in the protocol. I refined this scheme in two ways. First, code groups and protocols are identified with hierarchically constructed fingerprints. This allows them to be safely used within the network where not all of the associated code is available. (This is described as part of the code distribution scheme.) Second, to eliminate the overhead of carrying multiple fingerprints, the three fingerprints are further combined into a single compound fingerprint.

Defining capsule types in this manner provides them with two key properties that stem from the properties of fingerprints:

- New types can be allocated quickly and in a decentralized fashion. This is because fingerprints are effectively unique and the type depends only on the associated code; one need only choose a fingerprint function with a sufficiently large range to make the probability of a collision acceptably low. For a random 128 bit hash such as MD5 aims to provide, a collision is not expected until on the order of 2^{64} fingerprints are computed, a number far greater than the expected universe of network services. Given this, new services can be supported by mutual agreement among interested parties, rather than requiring centralized registration or agreement between all parties.
- Types form a secure binding between capsule and corresponding code. Assuming the hash is *one-way*, this eliminates the danger of code spoofing, since each node can rapidly verify for itself, without trusting external parties or requiring external information, that a given set of code corresponds to a given capsule. (By *one-way* I mean that a preimage cannot be found for a given fingerprint, nor a second preimage for a given fingerprint and preimage. This is a much weaker property to require than collision-resistance, by which I mean that two preimages that hash to the same fingerprint cannot be found.)

If a collision were to occur such that these properties did not hold[1], the association between a capsule and its code would be ambiguous. This can be handled in two ways. An occasional collision can be side-stepped by simply using slightly different code. This is facilitated with a directory service that maps a service name (such as "IPv6") to the latest collision-free implementation. A systematic breaking of the fingerprint function, on the other hand, would compromise the network until an entirely new fingerprint function is used. To facilitate this and other incompatible changes, ANTS includes a version field in the capsule format.

2.2.2 Active Node API

So far, capsule processing routines have been described in the abstract. In reality, the kind of routines that can be constructed depend on the active node API available to service developers. For example, without the ability to store and access node state, individual capsule programs will be unable to communicate with each other. Further, the compactness and execution efficiency of capsule programs will be affected by the API. Both are enhanced if the API is a good match for the processing, and degraded otherwise. For example, the neighbors at a given node may be found either by walking the entire routing table looking for adjacent nodes, or by asking the question directly of the node, depending on which API calls are supported. The direct query can be represented compactly and executed efficiently as a single API call, while the other program cannot.

[1]There is some evidence that the collision-resistance of MD5 may be broken in the foreseeable future [Dobbertin, 1996, Robshaw, 1996], but to my knowledge no evidence that the one-way property is in question.

Method	Description
`int getAddress()`	Get local node address
`ChannelObject getChannel()`	Get receive channel
`Extension findExtension(String ext)`	Locate extended services
`long time()`	Get local time
`Object put(Object key, Object val, int age)`	Put object in soft-store
`Object get(Object key)`	Get object from soft-store
`Object remove(Object key)`	Remove object from soft-store
`void routeForNode(Capsule c, int n)`	Send capsule towards node
`void deliverToApp(Capsule c, int a)`	Deliver capsule to local application
`void log(String msg)`	Log a debugging message

Table 2.1: Node API exported to capsule forwarding routines

This suggests that the node API is an important research question in itself. In this thesis, I address the question only so far as it relates to the main hypothesis. I provide the smallest API that can be implemented efficiently and with which I am able to demonstrate that many different, useful and short processing routines can be constructed. The choice of calls is based on the experience with an earlier proof-of-concept active network, the ACTIVE IP system [Wetherall and Tennenhouse, 1996], and early versions of ANTS.

The active node API exported to capsule forwarding routines in the ANTS toolkit is listed in Table 2.1. It provides calls in three main categories:

- *environment* calls such as `getAddress()` and `time()` return information about the local active node environment. Many similar calls, such as those that enumerate the local default routing table or examine forwarding statistics, can be added as they are needed in practice.

- *storage* calls, namely `get()`, `put()` and `remove()`, manipulate a soft-store of application-defined objects, including other capsules. Unlike a reliable store, objects placed in the soft-store are not guaranteed to be present after the forwarding routine completes; to simplify programming, however, they are guaranteed to be retained for at least this long. The objects are cached to retain those in regular use and so improve performance, and expired after an application-defined interval to ensure that stale information will not be retained in the network. Objects are also separated by service using the protocol type.

- *control operations* direct the flow of capsule processing and its propagation to other parts of the network. `routeForNode()` forwards a copy of a capsule towards a node using default routes, decrementing the remaining resource limit in the process. Custom routing services typically express their routes in terms of default routes, since this is a robust way to navigate the network. `deliverToApp()` delivers the capsule to a local application. If neither of these two functions are called for a capsule (either during its forwarding or after it has been placed in the soft-store) then it is discarded. `log()` signals informational or error messages to the node, which can also result in subsequent processing.

I chose each category because it is fundamental in some sense, yet still open to an efficient implementation. The omission of any category would greatly limit the type of new services that could be expressed, since the calls of one category cannot be synthesized from the calls of the others. Each category is intended to support new types of service with the minimum runtime cost at nodes within the network. Two decisions illustrate this bias towards a simple and minimal API.

First, the node state accessible via the soft-store is defined to be *soft-state* [Clark, 1988]. This means that its stability is not guaranteed, and so network protocols must be designed so that the loss of this state does not affect their correct operation (though it will typically impact their performance). Reliable storage would appear to simplify the design of some new services, but is more expensive to implement. What is more, soft-state is fundamental to the design of networks for good reason. Hard state does not solve the problem of node state that is no longer accessible because of a change in connectivity or routing patterns. Further, it introduces the potential problem of stale session information being retained in the network due to partial failures. Given these tradeoffs, ANTS is based on soft-state that is expired after an interval defined by the service developer.

Second, there are no timer functions. Timers are often included in other protocol construction kits, for example, the x-kernel [Hutchinson and Peterson, 1991], because they have a straightforward application to the implementation of reliability functions. I decided not to provide them because it proved possible to emulate them without complicating the active node design; they could be added to a future version of the architecture as long as their implementation is compatible with performance and security constraints. In the current architecture, timers can be emulated by using the arrival of packets within the network as timing events, effectively pushing the implementation of timers to within applications at the edges of the network. For example, capsules can be merged for an interval at a network node by combining them in the soft-store as they arrive and ending the collection operation in response to the arrival of a "flush" capsule, rather than the expiration of a timer.

There are also some obvious categories that are likely to be part of a production system, but are not required to explore the active network approach in the sense that, while their omission will limit new services in particular areas, it will not affect the basic utility of the programming model. For example, libraries for cryptography and transcoding will be useful to support new security and multimedia services, both areas that are prime candidates for exploration. A further set of API calls that interact with the node scheduler would enable integrated or differentiated flow services. This area too is ripe area for exploration.

In conjunction with the abilities provided by the extensible forwarding model (to combine these operations with "computational glue" and to carry and update customized packet fields) these API categories are sufficient to support the kinds of new services outlined in the introduction:

- Network characteristics can be discovered and new paths computed by using environment access functions in conjunction with header fields and controlled forwarding.

- Novel routing services can then be provided by using the soft-store to maintain newly computed routes and controlled forwarding to follow them.

- Multicast or other replicated services can be provided by creating new capsules within the network and having them follow routes maintained in the soft-store.

- Many-to-one suppression services can be constructed by using the soft-store in conjunction with the controlled discard of capsules.
- More speculative services such as network-based caching and merging are enabled by the ability to place capsules themselves in the soft-store.

2.2.3 Capsule Forwarding

The extensible packet forwarding model itself is driven around the capsule type, except at non-active nodes where default IP forwarding is performed. At each active node that a capsule reaches, the type field is used for demultiplexing to the corresponding forwarding routine, which is transferred separately. This dispatch process is conceptually identical to the demultiplexing performed using the Ethernet type and IP protocol fields within traditional implementations of link-layer and IP endpoints. It is also a basis for capsule forwarding that is known to be efficient from prior research on packet filters [Yahara *et al.*, 1994, Bailey *et al.*, 1994, Engler and Kaashoek, 1996] and more recent research on "layer four" switching techniques [Srinivasan *et al.*, 1998, Lakshman and Stiliadis, 1998].

Once identified by demultiplexing, the corresponding routine is executed to forward the capsule. This occurs within an execution model that is constrained in a number of ways that are intended to lead to efficient implementation strategies and to simplify the development of new services.

First, forwarding routines are permitted to run for a only short time, where short is determined by the target capsule forwarding rate of the active node. Active nodes enforce this handling by monitoring capsule runtimes. If the limit is exceeded, the node aborts the forwarding in progress, purges the active node of state associated with that capsule type, and maps that capsule type to default forwarding for future processing. This prevents both inconsistent state from being retained and future disruptions, but is somewhat severe. To allow new services to negotiate their processing requirements with active nodes more gracefully, a future version of the architecture might allow forwarding routines to specify hints, such as the expected and worst case run lengths. In any case, the result of restricting the runtime of forwarding routines is that processing is bound to those active nodes that are capable of supporting it.

Second, only a small amount of state that is explicitly identified as mobile is carried with the capsule. Other state must be explicitly stored at the node for later access or it will be reclaimed by the active node runtime. This differs from other distributed programming systems, where mobile agents represent ongoing computations that evolve as they migrate through the network, rather than the transfer of inert information. In these systems, language level mechanisms are often provided to control migration. For example, the "go" directive of Telescript [White, 1994], packages up the state of the current computation, transports the agent across the network, and resumes the computation where it left off. Similarly, in Kali-Scheme [Cejtin *et al.*, 1995] a thread of computation can be captured as its continuation and lazily migrated between address spaces and machines at any time.

These mechanisms are too heavyweight for extensible packet forwarding because they can require a large amount of state to be identified and transferred. Instead, ANTS allows the service developer to define additional capsule fields that can be accessed inside the network. These type-dependent header fields are defined along with the capsule forwarding routine and interpreted only by that routine, not by other generic code executed at active nodes. This mechanism allows service developers to structure evolving computations and so support many forms of service that could not otherwise be expressed, for example, routing or format conversion that depends on the congestion or latency encountered so far. Yet, unlike mechanisms in more general systems, this mechanism can be implemented with a minimum of runtime overhead.

Third, the external references that can be made by a forwarding routine are limited to those that can be resolved locally. This means that the routine can be run to completion without delay at each active node in the network. Forwarding routines can always make calls against the local node via its API. In some cases, however, one forwarding may need to call into a related one. For example, a capsule that follows a custom route may, within the network, create and forward a related capsule that reestablishes the route if it finds that the route is no longer fresh. For this call to succeed, the forwarding code of both types of capsule must be transferred together so that all of the required code will be locally available. This leads to the definition of the code groups — the transitive closure of forwarding code called from forwarding code — that are transferred by the code distribution protocol as a means of ensuring that local resolution of external references will succeed.

Drawing the line at late-binding between forwarding code makes a reasonable trade of generality for efficiency. ANTS already allows the type of a capsule, which determines the potential processing that it can undergo, to be selected at the moment the capsule is injected into the network. Disallowing the later binding of capsule type to processing within the network seems to sacrifice little practical expressive power. On the other hand, this restriction does simplify the implementation because it implies that forwarding routines will run to completion without stalling. When combined with bounded runtimes, this model provides the basis for efficient implementation strategies (similar to IP on Unix uniprocessors) that sequentially run forwarding routines to completion without requiring multiple threads or other mechanisms for concurrent processing. Further, this restriction enhances network robustness because it ensures that intermediate states that result from errors resolving external references will not be observed.

Forth, the active node runtime ensures that it appears to the service developer as if capsules are forwarded sequentially. (This is actually what occurs in the prototype. In an alternative implementation it would be possible to parallelize forwarding in ways that are consistent with this constraint, for example, with one thread of control per protocol.) This model simplifies the task of the service developer because there is no need to acquire and release locks during forwarding to protect against concurrency errors. It also simplifies the implementation of the active node because there is no need to check service code to ensure that deadlock and livelock are avoided and locks are released.

A final consideration is the handling of errors that arise during forwarding. Errors can result from insufficient resources to transmit a capsule, the lack of a default route, and a number of other conditions. To be efficient and robust, error processing must be handled locally. Yet to facilitate recovery and debugging, errors must be related to the original source of the capsule. In ANTS, errors that arise during forwarding are signalled locally as exceptions. This ensures that service code has an opportunity to handle the situation itself with alternative processing, or at least to place data structures in a consistent state. The latter is important because node cleanup in response to an error is generic, and so cannot ensure the integrity of service-specific data structures that persist across multiple capsules. In any case, the node runtime catches exceptions that propagate beyond the forwarding routine. If this occurs, the processing of the capsule is cleanly terminated (so that associated execution state is released) and the node moves on to subsequent capsule processing.

This local cleanup has proved sufficient for experimentation with the toolkit. In a production system, it would be straightforward to extend this outer layer of error processing with handling modeled after ICMP [Postel, 1981a, Conta and Deering, 1995], the control or error protocol of IP. In this case, errors that are not handled would result in the generation of an ICMP message to the source address carried by the capsule. ICMP safeguards, such as rate limiting and the suppression of error messages in response to error messages, would also be needed for robustness.

2.3 Code Distribution

Once a new service has been expressed in terms of capsules, it must be deployed within the network infrastructure before it can be used. ANTS provides an automatic *demand pull* code distribution system that makes it straightforward for an application to begin using a new service. In this section, I describe the demand pull scheme, beginning with its design goals.

Code distribution in ANTS is designed for traffic patterns for which caching is effective and service code that is compact. The *demand pull* system transfers the code that implements a new service to nodes along the path that a capsule using the service follows. The code is cached at these nodes for later use. The scheme takes advantage of the small code size to provide lightweight but unreliable transfer: code distribution will either succeed rapidly, in which case the interruption to forwarding should be transparent to applications, or occasionally fail, in which case the capsule that triggered the load mechanism will appear to have been lost and must be recovered by applications using the network in the normal manner. Experience with the ANTS toolkit suggests that code groups for real services can be relatively small, typically less than 16 KB, and I assume this to be the limit of code group size.

2.3.1 Design Goals

Code distribution is a new problem that arises in active networks but not the Internet. Its overall goal in ANTS is to preserve the semantics of carrying code with every capsule, while providing the performance and security properties achieved when the code is statically loaded into all of the nodes of the network. The performance of the scheme is important because, assuming the efficient execution of mobile code, the runtime overhead of using a new service is directly related to that of the code distribution mechanism. The security of the scheme is important because, if not well designed, code distribution activity may cause the network to become unstable or vulnerable to attack.

I identified the following specific goals:

- Adapt to changing routes and node failures.
- Scale to large networks.
- Minimize the amount of code stored at nodes and the distance that it is transferred. Ideally, a given forwarding routine would only be transfered once per network link, and only to those active nodes that require it.
- Minimize the time that elapses from when a capsule arrives at an active node to when the corresponding code is obtained. This latency directly extends the time that a capsule can take to travel across the network.
- Prevent code spoofing and denial-of-service attacks. It should not be possible to use the code distribution mechanism to attack the network.
- Not cause congestion.

The design of a code distribution mechanism that meets these goals depends on the characteristics of new service deployment. Two extremes are possible. If deployment is rare, a relatively static scheme that depends on management channels would be sufficient. If new capsules almost always require new services, a fully dynamic scheme in which code is carried with capsules would be required. The design of ANTS is geared to a situation inbetween these extremes. I expect that most services will be invoked by many capsules, and so require that the same processing be invoked many times at many different nodes. The basis for this expectation is the existence of traffic flows in existing networks. Further, while an active network allows potentially many services to be loaded for the purpose of experimentation over time, it is likely that a small number of services will account for a significant fraction of the total traffic at any given instant, especially given that real services would likely be selected from third-parties rather than developed by each user. Together, these assumptions suggest that code should be stored at nodes since the cost of its distribution can then be amortized across a substantial number of capsules.

To store code at nodes, I first considered an explicit connection setup mechanism, but abandoned it for three reasons. An explicit loading phase prior to use does not adapt easily to changing routes. Nor does it suit new routing services, where the route a capsule will take is not known before the customized forwarding routine is executed. Finally, it fundamentally changes the network layer to be connection-oriented, which is a major departure from the IP service model that has made the Internet so successful. I also avoided schemes that use reliable connections, such as a TCP connection between each active node and a code server, because of the difficulty of implementing a connectionless service in terms of a connection-oriented one.

Instead, the strategy used in ANTS is to pull code along the network path that is followed by a capsule with a connectionless protocol and as it is needed, and to cache the code at active nodes while it is in use. To bootstrap this process, applications at the edges of the network obtain the code that implements a new service before they begin using the service. I refer to this style of code distribution as a *demand pull* scheme, and describe it in the following sections. This scheme maintains code state along network paths while it is in use, and might be thought of as a "soft" kind of connection. Like traditional connections, it supports implementation strategies that benefit from the ability to amortize the overhead of distributing code across many capsules, for example, just-in-time compilers. Unlike traditional connections, it adapts to changes in routes and does not depend on reliable transfer mechanisms or require an explicit setup phase.

2.3.2 Code Distribution at End-Systems

Code distribution for a new service begins at end-systems, where applications gain access to the ANTS network. An application must first obtain the code that implements a new service before it is able to use that service. By "code", I mean all of the capsule forwarding routines that implement the protocol corresponding to the service, along with the organization of these routines into code groups. To perform this step, an application consults a directory service that maps a service name, such as "Sparse-Mode-PIM", to the corresponding service code. In the ANTS toolkit, the directory is simply the local filesystem.

This mapping serves to separate interface from implementation. Applications refer to services using high-level names that denote an interface to the network and are human readable. Within the network layer, however, services are manipulated solely in terms of their capsule types, which are based on fingerprints and directly name an implementation. In this manner, the directory service can be used to automatically upgrade the use of a service as it is debugged and improved by returning the latest version of its implementation. For example, applications resolving the "Sparse-Mode-PIM" service might receive and then use the implementation that corresponds to the most recent Internet Draft.

Once an application has obtained the required service code, it registers the code with the local active node. The node is then able to compute the types that will be stamped on capsules using the service as they enter the network, and has the code necessary to bootstrap the deployment of that service within the network. At this stage, the application is free to send capsules belonging to the new service into the network via the local node. These capsules will trigger code distribution within the network as described in the next section. Finally, when the application has finished using the new service, it should unregister the code.

2.3.3 Code Distribution Within the Network

At nodes within the ANTS network, code distribution is invoked in response to the arrival of an unrecognized type of capsule. The *demand pull* scheme is then used to draw code from the previous node that the capsule visited, and cache it at the current node for future use. I describe code distribution in terms of what code is distributed, and how code distribution capsules are generated and processed by active nodes.

Transfer Format

The code that is distributed should satisfy several goals. It must be sufficient to forward the capsule for which code is being loaded. It should be as small as possible so that code distribution does not consume more resources than is necessary. Finally, it should allow the fingerprint-based capsule types to be verified so that service code cannot be spoofed.

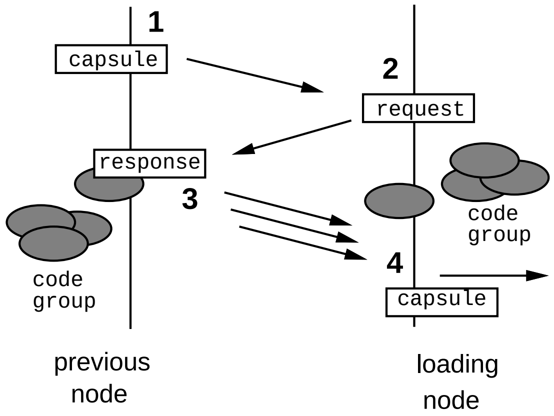

Figure 2-5: Transfer of Code Groups with the Demand Pull Protocol

1. A capsule is forwarded by an active node (the "previous" node) that possesses a copy of the corresponding code group. Before it is sent on an outgoing link, the `previous address` header field is set to the address of the forwarding node.
2. When the capsule arrives at the next active node (the "loading" node) and the code that matches its type is absent, a load request capsule is generated. It is sent to the node identified by the `previous address` field.
3. When the load request arrives at the previous node, a series of load response capsules are generated. These carry the requested code group and other to the loading node.
4. As load responses are received by the loading node, they are reassembled to form the requested code group and the result is checked. When reassembly is complete the load is also complete, and the code group is then used to forward the capsule that triggered the load sequence, as described in the first step.

Figure 2-6: Steps in the Demand Pull Protocol

These goals lead to a transfer format based on the notion of code groups. Recall that the extensible forwarding model requires that capsule processing run to completion without stalling. This in turn requires that code for the capsule forwarding routine plus all other forwarding routines that can be called from it be present. This leads to the definition of a code group as the transitive closure of forwarding routines that can be called from each other within the network. The minimum amount of code that can be transferred to satisfy the first goal is thus an entire code group.

Transferring a code group by itself does not meet the last goal, the verification of the capsule type, because the protocol type cannot be computed. To meet this goal, it would be sufficient to transfer all of the forwarding routines in the protocol plus their arrangement into code groups. Then all of the capsule types associated with the service could be computed and compared with that of the capsule waiting to be forwarded. But this is excessive.

In ANTS, code is transferred in units of the forwarding routines that belong to the needed

```
define CACHE = (the code cache, keyed by capsule type)
define NODE = (the address of this active node)

CAPSULE = (a new capsule from the network)
ENTRY = (the entry in CACHE associated with CAPSULE.TYPE)

try {

  forward CAPSULE with ENTRY.FORWARD           // any outgoing capsule will
                                               // have PREVIOUS set to NODE

} catch (code loading exception) {

  if (ENTRY == null) {
    ENTRY = (a fresh entry in CACHE)           // can flush old code group
  }

  ENTRY.TRIGGER = CAPSULE                      // save most recent trigger
  ENTRY.EPOCH = (current time)

  REQUEST = (a fresh load request)
  REQUEST.LOADTYPE = CAPSULE.TYPE
  REQUEST.PARTS = (complement of ENTRY.PARTS)  // ask for missing parts
  send REQUEST to CAPSULE.PREVIOUS             // REQUEST.PREVIOUS set
}
```

Figure 2-7: Generation of Load Request capsules during forwarding

code group plus remaining code group types. Because a protocol type is defined only in terms of code group types, this information is sufficient to compute the required capsule type and prevent code spoofing. Since code group types are much smaller than the forwarding routines of the code group, this format results in a significant savings in code size when there is more than one code group.

A pragmatic issue that arose in transferring code groups is that of encoding calls between the forwarding routines that make up a code group. This cannot be done directly in terms of capsule types, because this would require that recursive fingerprint equations be solved. Instead, calls are expressed in terms of local names that are embedded within the code of the forwarding routine. These local names are linked only within the transferred code group to prevent code spoofing as before. For example, one routine that constructs routes may be named simply "producer," while the other routine that follows them may be named "consumer." Any producer routine could be combined into a code group with any consumer routine, while securely allowing calls between only the selected routines.

Demand Pull Protocol

The processing of code distribution capsules implements an unreliable transfer protocol that is appropriate for small code groups. The common case loading sequence that the protocol aims to provide is shown in Figures 2-5 and 2-6. Loading occurs in four steps: a capsule

```
define CACHE = (the code cache, keyed by capsule type)
define REQUEST = (the load request received from the network)

ENTRY = (the entry in CACHE associated with REQUEST.LOADTYPE)

if (ENTRY == null) return                        // no code present
if (ENTRY.FORWARD == null) return                // or code incomplete

foreach i in (REQUEST.PARTS & ENTRY.PARTS) {

  // send each requested part that we have
  RESPONSE = (a fresh load response)
  RESPONSE.LOADTYPE = REQUEST.LOADTYPE
  RESPONSE.PART = i
  RESPONSE.CODE = (part i of ENTRY.CODE)
  send RESPONSE to REQUEST.SOURCE
}
```

Figure 2-8: Processing of Load Request capsules

is forwarded with the required code and has its previous address field updated; the capsule arrives at the next node where the required code is not present, which causes a load request to be sent to the previous node; the load request arrives at the previous node and a series of load responses is generated to transfer the required code; and the responses arrive at the loading node, where they are combined to obtain the required code and forwarding is resumed.

I describe this *demand pull* protocol in detail by presenting pseudo-code for the processing by active nodes of regular capsules, load requests and load responses. Besides the common case, the pseudo-code specifies the handling of a number of different scenarios in which code is loaded concurrently and capsules are lost or corrupted. In the pseudo-code, the code cache of an active node is modeled as an associative store that maps a capsule type to an entry. The entry is a record that has slots for several distinct kinds of information: a forwarding routine, the code that is being loaded, a parts field describing the completeness of the load, a trigger capsule waiting to be forwarded, and a timestamp. The cache is managed in terms of code groups in a least-recently-used fashion.

The generation of load request capsules during capsule forwarding is shown in Figure 2-7. Here, once a capsule arrives at an active node, its entry in the code cache is looked up using the capsule type. Normally this entry will contain the required forwarding routine, which was either previously loaded or supplied by a local application. In this case, the routine is simply executed to forward the capsule. To support subsequent loading, the previous address field of each capsule sent during forwarding is set to the address of the current active node.

Code loading is the exceptional case that occurs when there is no complete entry in the code cache. If no entry exists at all, it is created, possibly evicting an old code group from the cache in the process. This action corresponds to the first load attempt for a given type of capsule. Otherwise, if an incomplete entry exists, a previous load attempt has not yet succeeded. In either case, the capsule that triggered code loading is stored as part of the entry, along with a timestamp, while the load proceeds. Note that storing the most recent trigger capsule implicitly discards any older trigger capsule; it has been lost in the network. This mechanism limits the buffering needed at active nodes.

```
define CACHE = (the code cache, keyed by capsule type)
define RESPONSE = (the load response received from the network)

ENTRY = (the entry in CACHE associated with RESPONSE.LOADTYPE)

if (ENTRY == null) return                       // no load in progress
if (ENTRY.FORWARD != null) return               // already complete

add part RESPONSE.PART to ENTRY.CODE
ENTRY.PART = (ENTRY.PART | RESPONSE.PART)

if (ENTRY.PART == (all parts)) {                // last one arrived

  OK = (verify fingerprints for ENTRY.CODE match ENTRY.TYPE)
  if (!OK) {                                    // code is corrupt
     delete ENTRY                               // so purge it
     return
  }

  // install all routines in the loaded code group
  foreach forwarding routine i of type t in ENTRY.CODE {
    E = (the entry in CACHE associated with t)

    if (E == null) {
      E = (a fresh entry in CACHE)
      E.TYPE = t
    }

    E.FORWARD = i
    E.CODE = ENTRY.CODE

    // forward any waiting capsules
    if ((E.TRIGGER != null) && ((E.EPOCH - (time)) < 1 second)) {
      forward E.TRIGGER with E.FORWARD
      E.TRIGGER = null
    }
  }
}
```

Figure 2-9: Processing of Load Response capsules

Next, a load request capsule is constructed. It identifies the missing code by carrying the capsule type that triggered the load, along with a parts field. The latter field is used to allow multiple loads to each make incremental progress. It identifies the missing parts that have not already been transferred. In the case of an initial load it has a value representing all parts. In the toolkit, this field is implemented simply and efficiently as a bit vector. The load request is then sent to the node carried in the previous address field of the trigger capsule. Once this is done, the active node can continue with the task of processing other capsules. All of the state necessary to continue loading is stored in the code cache, and the node need not suspend execution or set any timers.

The processing of load request capsules is shown in Figure 2-8. When a load request is received, the code cache is checked for an entry corresponding to the requested capsule type. It is highly likely that this entry exists and is complete because the code was used recently to forward the trigger capsule. In this case a series of load response messages is generated, one for each part that is requested and available. Each part is self-describing in the sense that it has sufficient information to be processed as it is received without depending on other parts. The parts are sent to the origin of the load request capsule. In the rare case that the requested code is not present, the load request is simply discarded. In this and other situations that will occasionally arise (such as the loss of a load request or response capsule or transfer of corrupted data) the load sequence is permitted to fail.

Thus in an ANTS network there is an additional form of loss, besides congestion and bit errors: capsule loss due to code distribution failure. When this loss occurs, applications using the network will detect it and respond (as they already do for other forms of loss) to provide whatever level of reliability is appropriate. The active network must clearly be engineered such that this form of loss is infrequent to provide a useful basic service. Nevertheless, I believe that exposing code distribution failures to applications rather than attempting to hide it within the network will ultimately benefit network performance.

The processing of load response capsules is shown in Figure 2-9. As each arrives it is merged with the code that is already present. First, the code cache is checked for an entry corresponding to the type of code carried by the response. If none is found or the code is already complete, the response is discarded. Otherwise the part information is used to merge the code it carries with that already in the cache. If the code is now complete, the entire load can be completed and the trigger capsule forwarded.

Before activating the code, its fingerprint-based types are computed and compared with the capsule type that triggered the load sequence. If they do not match, the code is corrupt, and the entire entry in the code cache is deleted. Otherwise, the entry in the code cache is completed with the corresponding forwarding routine, and further entries are created for each other forwarding routine in the code group. Finally, the forwarding routines are executed to forward any trigger capsules that are waiting in the code cache. Before each is forwarded, the delay it has incurred is checked. If the delay exceeds a fraction of the network lifetime (one second is used here) the capsule is discarded. This check greatly reduces the hazard of an old capsule being delivered by the network.

Other Considerations

In addition to the processing described, there are several other considerations that are only partially addressed by the ANTS architecture until there is more experience with large scale active networks: loss and congestion; thrashing and livelock; and denial of service.

Since the protocol is unreliable, there is a chance that code distribution will fail and the capsule be lost, even if code groups are small and transfers infrequent. Loss due to bandwidth congestion becomes a concern when code transfer sizes become large relative to the available network buffering or loading activity begins to consume a significant amount of network bandwidth. In this case, load adaptive mechanisms should be explored, for example, TCP slow-start. Even in the case that code transfer does not cause congestion, the probability of loss increases with the number of messages sent because the network is shared with other traffic. Concerns about loss will ultimately limit the code transfer size to a small number of messages. If the level of loss is problematic, mechanisms that isolate code transfer traffic (RED, WFQ, priorities) or increase its redundancy (FEC) should be explored.

Loss and congestion effects are limited in the ANTS toolkit simply by limiting the maximum code group size to 16 KB. This limit is large enough that a variety of new services can be explored, and small enough that demand loading is plausible.

A separate concern is that loss not adversely interact with the assumptions of the transport layer protocols that use the network. For example, TCP congestion control mechanisms are known to interact poorly with wireless media, where a significant amount of loss can arise from bit errors rather than congestion. These kind of effects are unlikely to be significant in ANTS because the vast majority of code distribution failures are expected to occur in response to congestion. It is important, however, that repeated load failures do not occur, because they may cause an application to falsely determine that a host is unreachable. This is unlikely to be a problem in ANTS because code loading is able to make incremental progress. This means that in the rare case the first load does not succeed, the second is significantly more likely to succeed.

As with any cache, thrashing can occur if the "working set" of code groups does not fit in the code cache. In the extreme, livelock is possible if concurrent loading activity causes partially loaded code to be evicted from the cache. Neither of these behaviors will occur for the traffic patterns I have targeted and in a properly engineered network, and so no protection against thrashing is built into the ANTS toolkit.

Nonetheless, it is desirable to prevent thrashing in a large-scale network in order to guarantee that the network is robust. This requires that load be shed when it exceeds the sustainable number of new services in use, for example, by returning an ICMP-style error message. One possibility for future versions of ANTS is to to enforce a minimum caching time for each code group, as measured from the first moment of loading activity for that code. Such a minimum lifetime translates into a maximum rate of cache turnover. An alternative would be to retain code while it is in use, that is, not evict code until it has been idle for a specified time. Compared to a minimum caching time, retaining code while it is in use improves the stability of existing services at the expense of the ability to introduce new services under high load.

Finally, it should not be possible to use the code distribution system to mount a denial of service attack on the network, for example, by causing large amounts of code loading activity or consuming a large portion of the code cache. In ANTS, some protection is afforded by the hop-by-hop nature of the code transfer path. With the demand-pull protocol, code is loaded at each hop along the path beginning at the node where an application first injected a new type of capsule into the network. This means that a distant node cannot be attacked without first attacking a neighboring node, and so the potential for anonymous attacks in a large network is greatly reduced. Further, the hop-by-hop path provides a basis for tracing the path along which code is loaded. (This line of reasoning also requires that applications cannot make code loading capsules "materialize" in the middle of the network. The node operating system enforces this restriction.)

In the toolkit, hop-by-hop loading is enforced by processing code capsules at the first node that they encounter, rather than allowing them to continue on to a further destination. This suffices because all toolkit nodes are active and so all transfers occur between adjacent nodes. The implementation of this strategy is more complicated in networks with asymmetric routes and relatively few active nodes. This is because the route from one active node to the previous may legitimately pass through a third active node because of the asymmetry. If this is a problem in practice, the request and response capsules can be authenticated to check they originated at neighboring nodes.

2.3.4 Properties and Extensions

Though it has a straightforward design, the *demand pull* scheme possesses a number of important properties. At a high-level, it provides an adaptive connection setup that is compatible with new routing services. It is scalable, since it draws code along network paths, growing a neighborhood where it is in use, rather than attempting to deploy new code into an entire region of the network. The use of fingerprint-based types prevents code spoofing without depending on digital signatures and shared public key infrastructure. The hop-by-hop nature of the protocol limits the scope of denial-of-services to adjacent active nodes.

More subtly, by following the path of the capsule and exposing losses during code loading to the application, the scheme limits bandwidth consumption and latency. By bounding these quantities tightly, I expect that code loading activity will be transparent to applications using the network, being below the "noise" threshold. This is not necessarily the case for other solutions, such as loading code from well-known remote servers or with reliable transports. Section 5.2 contains an analysis of these considerations. The properties are:

- The bandwidth consumed by the network in response to a capsule of unrecognized type arriving at a node is limited to be $M+1$ message-hops, where M is the size of the maximum code group in messages and a hop is the distance between neighboring active nodes.
- The latency to load code is roughly the round-trip time if the hop is long in terms of messages, or $M+1$ times the latency to send a capsule if the hop is short in terms of messages. Again, M is the size of the maximum code group in messages and a hop is the distance between neighboring active nodes.

There are also some extensions to the basic protocol that could be incorporated once there is sufficient experience to suggest that they will be worthwhile.

- When default routes are followed, code can be loaded into multiple network nodes in parallel by forwarding code from one node to the next before it is fully assembled. This strategy can be supported in a more general fashion than strictly default routes by allowing a capsule to carry a small amount of code that enters other code into the local node cache. This facility can then be used to construct capsules that "prime" a path with code. Loading in parallel would reduce the latency of loading, especially for larger code groups that flow along long paths.

- Code can be speculatively transferred from one node to the next by using heuristics. For example, when one node must load the code for a new service, it is reasonable to assume that subsequent nodes will need to load the same code, and so it can be transferred after the capsule is forwarded but before it is requested. Speculative transfers would reduce load latency by a small amount at the cost of some redundant code transfers.

2.4 Node Operating System

Once a new service has been deployed, it must acquire network resources in order to be of use. A key difficulty in designing an active network is to allocate resources to services as they execute while preventing unwanted interactions. Not only must the network protect itself from runaway protocols, but it must distribute resources between co-existing protocols and offer them an independent view of the network.

The node operating system performs these tasks. Rather than implement capsule processing on top of a Unix-like operating system in terms of time-sharing and address spaces, active nodes rely on a specialized operating system. It provides sharing and protection mechanisms that are tailored to extensible forwarding, and so support it in a manner that is compatible with high levels of performance. Capsule processing resembles a system in which there are many short-lived tasks that are performed on behalf of many simultaneous users. Operating system mechanisms must be able to run at the capsule forwarding rate to perform well in this domain, and so must require minimal initialization. This excludes heavyweight processes and login schemes. Instead ANTS is built around safe evaluation techniques and fingerprint-based separation of state.

Several threats must be addressed. Nodes must ensure that service code cannot corrupt them, interfere with other service code, or consume an excessive amount of network resources. It is important to guard against both accidental errors and malicious attacks. This is because the network is a large system shared by many users, in which some accidental errors are inevitable, and some users are bound to be malicious. My goal has been to accommodate service designers who are not trusted and enforce protection and arbitrate resource consumption with mechanisms inside the network infrastructure. My approach has been to build on safe execution techniques to the extent possible, rather than infer safety through trust relationships. This is because authentication and authorization schemes are vulnerable to accidental errors as well as difficult to manage in a large network in which many parties act independently.

I describe the node operating system in terms of its protection mechanisms that isolate faults, and its resource management mechanisms that prevent services from monopolizing the network.

2.4.1 Protection

A protection model isolates services so that they are unable to influence the behavior of each other in unintended ways. In ANTS, this requires mechanisms for ensuring the integrity of the active node runtime, and separating the code, node state, and capsules of different services.

I rely on recent safe execution technologies to ensure that the node is not corrupted while executing untrusted service code. Recent advances in operating systems and programming language technology have the potential to implement protection mechanisms with execution efficiencies that are comparable to embedded native code. They are especially useful for active node design because they support tight coupling between protection domains and most work is done only once at code loading time. In contrast, traditional address space protection is not effective for tightly coupled domains and performs all work at runtime. Three developments illustrate this progress.

- Software-based fault isolation (SFI) [Wahbe *et al.*, 1993] provides a means of implementing address space-style memory protection in software by inserting run-time checks into binary programs. Compared with hardware fault isolation, it trades a small run-time overhead for a greatly decreased cost of crossing the isolation boundaries. The overhead of this scheme depends on how frequently isolated code interacts with other modules and the form of safety required, but is typically less than 20%. It is possible to insert the run-time checks at load time, an approach that is used as the basis of the Omniware mobile code system [Adl-Tabatabai *et al.*, 1996].
- Java bytecodes allow efficient interpretation [Gosling, 1995]. Aspects of the individual bytecodes (the incorporation of high-level type information) together with its approved usage (a single type-state of the stack along all execution paths) allow rapid load-time verification of a number of properties (no stack overflow or underflow). Once these properties are checked at load-time, further checks can be omitted at run-time to yield a more efficient execution that does not sacrifice safety.
- Proof-carrying code (PCC) [Necula and Lee, 1996, Necula and Lee, 1998] combines a program with a formal proof that it respects a safety policy, such as type-safety. The proof is generated from safety properties that are checked by the compiler. Since it is generally more efficient to check a proof than to generate it, PCC provides a relative efficiency when a program is to be evaluated many times. Since proofs can be generated for restricted C and assembly language programs, and no additional runtime overhead is incurred, this approach can be very efficient.

The ANTS architecture uses these kind of mechanisms to provide efficient memory and type-safety. The ANTS toolkit, for example, is written in Java. The focus of the design effort pursued here is on how to extend this level of safety to other forms of protection and resource management, rather than on how to improve its baseline efficiency.

To separate service code, the ANTS architecture enforces a basic form of protection suggested by the capsule programming model: each capsule is authorized to determine its handling, but not the handling of other capsules. This handling is specified by the value of the type field that is stamped on each capsule as it enters the network. Because of the definition of type values, the associated forwarding code is identified unambiguously, and is already specified in full. That is, the meaning of a capsule computation is fixed at the moment it is injected into the network, regardless of the actions of other parties. This form of protection is enforced by the code distribution system.

To separate node state, one possible approach is to sign and authenticate every capsule against a principal. This would allow the active node to enforce a protection model whereby state maintained on behalf of a principal could only be manipulated by that principal or other principals on an access control list. However, compared to the Internet today, this scheme would impose a computationally expensive step at a very low level in the system.

Instead, ANTS provides service-based protection model that is analogous to existing protocols in the Internet: capsules can only manipulate state associated with the protocol to which they belong. This is implemented by using capsule types as names that partition the soft-store at active nodes. This scheme is computationally inexpensive, since the types are already provided by the programming model. The security properties of fingerprints provide a probabilistically strong guarantee that it is not feasible to construct different forwarding routines that share the same capsule type.

The basic functioning of the system is simple. Forwarding routines that need to share state within the network are combined (in terms of their code groups) into a protocol. For example, one forwarding routine may be used to construct specialized routes, while another may be used to follow them. Both routines should be treated as a single unit of protection within the network. This is achieved by computing protocol types as described in Figure 2-3. Objects placed in the soft-store are marked with the type of the corresponding protocol, and can later be manipulated only by forwarding routines that belong to the same protocol. Note that this model benefits from node storage that is automatically expired, as opposed to the "hard" or persistent state that is used in a distributed filesystem. This is because state cannot be manually removed after it is created unless provision was previously made when defining the protocol. To further support a simple form of sharing across different protocols, the forwarding code that belongs to a given protocol can selectively mark its state world readable and/or writable.

By the definition of types, once a capsule type is calculated, the contents of the corresponding protocol are fixed. This means that new forwarding routines that purport to belong to it in order to manipulate its data in a different manner are disallowed, and it cannot be extended by any party so that new types of capsule can manipulate old state that exists within the network. Any extension appears to be a new service to the network, and so is isolated from the previous definition.

The model is further enhanced by restricting the ability to create arbitrary capsules at arbitrary places in the network. This is because the arrival of a capsule at a node generates a call that manipulates node state, and limiting where calls can be made increases the utility of code-based protection. In the Internet today, for example, UDP packets cannot be transformed into ICMP informational messages just before they reach their destination. If they could, then network attacks such as ICMP smurfing[2] would be easier to mount. The analogous behavior is straightforward to provide in ANTS by using active nodes to restrict forwarding code in two ways:

- It cannot create capsules of another service.
- It cannot mutate capsule types to leave the original service.

When these rules are added, the ANTS protection model mimics the implicit protection model of Internet protocols and packet forwarding. Like the Internet, this model does not

[2]See CERT Advisory CA98-01, *Smurf IP Denial-of-Service Attacks*, January 1998.

place any restriction on which host can send what type of capsule. Nonetheless, the overall model does prevent the construction of some classes of service that would pose security problems. It is not possible, for example, to create a new service that:

- Searches the network and discards capsules belonging to another service. Each capsule determines its own forwarding.
- "Spoofs" capsules at arbitrary network locations. Capsules must travel through the network from source to given location in a legitimate manner.
- Constructs a capsule that accesses the state of another service, capturing private information. The types of capsule that can manipulate service state are fixed when the service is constructed and cannot be extended within the network.

This last example highlights the tension between protection and composition of services. By preventing the definition of a service from being modified, an individual service is protected, but the means by which users can effect a composite service, for example, multicasting plus path MTU discovery, has been limited. In ANTS, this tension is partially resolved by providing protection within the network and composition at the edges. Composite services can be formed at any time by combining the corresponding set of forwarding routines. The composite service is then assigned new type by the fingerprint-based naming mechanism, and so is treated by the network as an entirely different service. Similarly, new versions of a service are treated within the network as separate services. This scheme is workable but has the drawback that composed services cannot share state within the network, even when they share code. World-readable protocol state helps, but does not solve this problem. Suggestions for the evolution of the existing model to a file system model are given in the future work portion of the conclusion.

The ANTS model contrasts with the protection model typically supported by filesystems. In a filesystem, principals are typically users, and the sharing of state is specified in terms of a small number of common operations, such as read versus write permission for specified users. In ANTS, the principal is effectively a set of code. This code encapsulates the state, so that it can be manipulated by sending capsules, but only in ways allowed by the code, since it guards the associated state. That is, rather than protect state by authorizing each access, we protect state by authorizing each type of manipulation. This is equivalent to the protection implicit in network protocol implementations today, despite the fact that it is implemented in terms of untrusted mobile code.

If stronger protection is required, authentication and authorization checks can be layered on top of this basic protection by including them as part of the forwarding routine. This approach would allow, for example, the join and leave operations of a multicast service to be restricted while not slowing send operations. Since send operations are expected to be much more frequent than join and leave operations in the same manner that signaling is infrequent compared to data transfer, this strategy increases security without significantly affecting performance. Trading occasional updates that are restricted for frequent access that is unrestricted is likely to be common. For example, the process that maintains the default routing information must restrict its updates (since many services depend upon them) but need not restrict its use otherwise (since all services are permitted to use default routes via the node API).

2.4.2 Resource Management

The protection model separates the state of different protocols, but does not ensure that network resources are allocated in a reasonable manner. As well as bandwidth, several different types of node resources must be managed: capsule processing time, the soft-store, and the code cache. The latter is managed by the code distribution protocol. Additional mechanisms described here control the allocation of the other resources, both at individual nodes and across a group of nodes. In the design of these mechanisms, the emphasis was on lack of starvation (in the sense that the design allows each protocol to make progress) rather than fairness. This pragmatic decision reflects the operation of existing networks, in which mechanisms such as FIFO queuing and RED gateways allow the network to be reasonably utilized but do not ensure that resource allocation is fair.

Given that the basic network service is connectionless and lacks resource reservations, all resources must be allocated implicitly by the flow of capsules. To accomplish this, I treat all resources as rate renewable and limit their consumption on a per capsule basis. This approach is straightforward, yet effective if treated as a mechanism to prevent gross imbalance rather than to perform precise accounting. Processing time and link bandwidth are already rate resources. Soft-storage is converted into a rate resource by treating the soft-store as a cache and evicting old objects when new storage is needed.

Processing time is allocated at each node by allowing each capsule to execute for the time accepted by the node. Preferably, it will prove feasible to enforce this limit statically, either with a restricted language approach similar to [Deutsch and Grant, 1971] or by using proof-carrying code techniques [Necula and Lee, 1996]. In the prototype, the limit is implemented dynamically with a timer. This requires that care be taken when aborting long running computations to ensure that protocol data structures are left in a consistent state.

Bandwidth and soft-store are allocated across all nodes by carrying a resource limit field with each capsule. The resource limit field functions as a generalized form of the TTL (Time-To-Live) field in IPv4 or the Hop Limit field in IPv6. It is decremented by nodes as capsules are sent and objects are put into the soft-store, and capsules are discarded when the value of this field reaches zero. For this limit to be meaningful, only nodes can alter this field. Further, in order to reason about total resource bounds, resources must be transferred when one capsule creates another inside the network, rather than simply manufactured anew. ANTS decrements the resource limit of a capsule by one before it can create another capsule within the network. Since this behavior is compatible with that of the TTL and Hop Limit fields in IPv4 and IPv6 today, a single combined mechanism can be used to prevent infinite loops across both active and non-active nodes.

These mechanisms prevent too many resources being consumed by one capsule at an active node, and prevent program errors that cause infinite loops across many nodes. They do not, however, guarantee that network resources will be consumed in a reasonable manner. To see this first requires a definition of "reasonable". I postulate that the resource consumption of one capsule is reasonable if it visits each node no more than k times, assuming there are no loops in the standard routing tables, and where k is a small number such as one or two. This definition readily allows multicast services and accommodates Internet protocols, but disallows loops and flooding behavior such as ICMP smurf attacks. I refer to protocols that satisfy this definition for some value of k as $k - bound$.

The use of TTL schemes to achieve this definition of "reasonable" fails because it does not provide a tight enough bound on global resource consumption and it does not prevent activity from being concentrated on a small region of the network. The bound is not tight because the resource limit of a capsule is simply decremented and copied when one capsule creates another within the network. The total resource limit that can be consumed can therefore be multiplied through exponential growth. ANTS uses this definition of resource limit despite this weak bound because it readily extends from unicast to multicast schemes and is compatible with Internet behavior today.

To tighten the bound, an alternative that I considered is the strict division of resource limits between an existing capsule and freshly minted ones within the network. This rule implies that the total resource usage is bounded by the resource limit of the original capsule as it enters the network. Though it appears a useful property for bounding global resource consumption, it is rendered ineffective by multicast: to support a large multicast session the limit needs to be unreasonably large. For example, to reach around 1000 receivers at a distance of 10 hops, a limit of around 2000 is needed. A limit this large can cause much harm if one capsule divides and bombards a small number of receivers. Further, strict division is complicated in practice by the need to manage information on how to divide resource limits, given that distribution trees are not generally balanced. Without division that is matched to a distribution tree, the tightness of the bound is weakened. For example, if the tree is unbalanced such that there are twice as many branch points on the longest path, then the naive strategy of sharing the resource limit equally at each branch point will require a resource limit that is the square of the resource limit needed if division were matched to the tree.

Further, a TTL scheme cannot prevent resource consumption from being concentrated at a small number of nodes even in the case that a capsule is not replicated as it travels through the network. This is because a capsule must be stamped with a resource limit that has a reasonable margin of safety, since it is not generally known whether it must travel across a large portion of the network or to a neighbor. It is therefore possible to construct a malicious service that routes a capsule until it reaches a particular node and then proceeds to loop the capsule between that node and an adjacent one until the resource limit is exhausted. With a maximum resource limit of 255 (corresponding to the 8 bit field used in IP) the forwarding work of roughly 100 capsules could be inflicted on a target node for the cost of one capsule at the sending node.

A better scheme is therefore needed to limit global resource consumption. Prior work in this area is limited, with other systems typically relying on quotas, peer-pressure, or centralized control. ANTS protects against poorly designed services by requiring that service code by certified by a trusted authority with a digital signature before it can be freely executed. The purpose of this signature is to vouch that the protocol will not use network resources in a harmful manner. Service code that is not certified can still be safely run at a reduced level of performance by restricting all such code to use no more than a small amount (say 10%) of the available bandwidth. In this manner its impact on the network is limited. Alternatively, code that is not known to be trustworthy can be mapped to the default forwarding service. These mechanisms are abstracted in the ANTS architecture into a node security policy because their form can very depending on local administrative requirements. As long as the mapping from code to forwarding method is relatively stable it will not affect the ability to program the network as a whole because the notion of heterogeneity is built into the design of the programming model.

Despite requiring that service code be certified, ANTS provides a system that is capable of evolving much more rapidly than is possible today. This is because certification differs from standardization in that it does not seek to define a single preferred behavior. Rather, it seeks to establish that a service makes reasonable use of overall network resources, regardless of whether it is considered an effective means of accomplishing a particular task. The ability to run new but not yet certified services at a reduced level of performance simultaneously allows for rapid experimentation. Nonetheless, this mechanism differs in character from other aspects of the node design and is a candidate for further research. It relies on trust, which weakens security, and runs counter to the philosophy of active networks in that it depends on a centralized authority, which slows innovation.

In the long run, the issue of global resource consumption may be less important as more sophisticated scheduling algorithms, for example, RED and fair queueing, provide better separation between flows. Further, it may be possible to automatically prove that certain capsule program forms are safe for use. For example, routines that forward to a fixed address and do not send more than one capsule clearly satisfy the definition of "reasonable", and this program form can easily be recognized. Though it may sound restrictive, this form can express selective discard, congestion notification and network merging services.

2.5 Deployment Considerations

Ironically, before a flexible architecture can be exploited to introduce new services, it must first be introduced as a new service available in the Internet. The focus of this dissertation is on the properties of a fully deployed ANTS network, rather than a transition plan to deploy it in the first place. Nevertheless, ANTS was deliberately designed to lend itself to incremental deployment as a means of constructing a large-scale active network that is part of the Internet.

The two restrictions on the capsule model that support extensible packet forwarding also support incremental deployment. First, since a mix of active and non-active nodes is permitted, it is straightforward to deploy an ANTS network by incrementally activating nodes. These nodes are likely to begin with end-systems, along with routers connected to bandwidth-limited wireless and access links, and followed by increasingly high performance internal routers. As more nodes are activated, especially those at strategic locations, the efficacy of introduced services will rise. For example, a greater number of potential branching points will lead to an increase in the bandwidth savings of a multicast service.

Second, though provision is made for the dynamic loading of new services, there is no reason that the cache cannot be pre-loaded or primed with services that are likely to be used. This allows popular services to be configured and optimized as trusted code in order to deliver high levels of performance on a given platform.

Further, it is possible to benefit from the combination of these strategies by constructing routers that are active for selected services and not active otherwise. This can extend the performance range of active nodes because compute intensive services that are not dynamically loaded because they cannot be forwarded at the line packet rate might instead be statically loaded if capsules that use them are expected to arrive infrequently. This is analogous to the way that ICMP and other services are handled separately from the IP "fast path" in high-bandwidth routers. This line of reasoning suggests that hybrid active node implementations, for example, a PC that is attached to an IP router and receives packets via a filter, will be useful for combining the performance of high-end routers with the flexibility of programmable services.

Chapter 3

The ANTS Toolkit

The description of the ANTS architecture in the previous chapter sketched the main components of a system and their function, but left unspecified details of their use and implementation. In this chapter, I describe the design of the ANTS toolkit, a reference implementation that provides one concrete set of answers to such questions[1].

The toolkit is written entirely in Java as a class-based framework. Each node and its locally connected applications are run in a single Java virtual machine, typically as a user-level process on a Unix-based operating system. Note that, for reasons of convenience, the toolkit is not built as an extension of an existing IP implementation, but rather as a complete network layer that uses the services provided by the standard Java libraries.

The intent of building the ANTS toolkit was to allow the suitability of the active network approach to be evaluated. I have used it to experiment with the Web caching and multicast protocols that are described in the next chapter. It has also been used by others to study novel protocols: an auction service [Legedza *et al.*, 1998]; a reliable multicast protocol [Lehman *et al.*, 1998]; a network-level packet cache [Johnson, 1998]; and a defense against TCP SYN-flooding [Van, 1997].

In the remainder of the chapter, I describe the toolkit design and implementation along two lines. First, I describe the interfaces of the key Java classes from the point of view of a network programmer, including an example of how the interfaces are used to introduce a new service. Second, I describe the implementation strategies and decisions underlying the toolkit, as well as issues that are not fully addressed in the current mapping between architecture and implementation.

3.1 Overview of Key Interfaces

The ANTS architecture defines the concepts of *active nodes* that are connected by *channels* to form a network, and *capsules* that are injected into the network by *applications* and forwarded through it with customized routines. In the ANTS toolkit, these concepts are mapped to the Java classes shown in Figure 3-1 and described in the following paragraphs.

[1] The ANTS toolkit is available in source code form at `http://www.sds.lcs.mit.edu/activeware`.

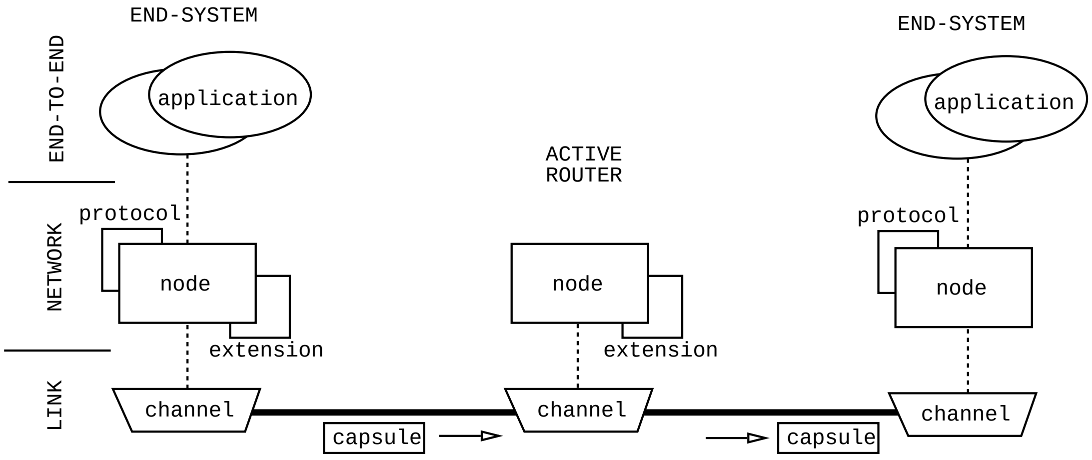

Figure 3-1: Key classes of the toolkit and their relationship

- Each active node, whether acting as an end-system or router, is represented by an instance of the class `Node`. A node extension architecture allows for heterogeneity in which different nodes support different service components, for example, caching or transcoding, as appropriate. Extensions are developed by subclassing the *abstract*[2] class `Extension`.
- Each network interface is represented by an instance of the class `Channel`. Channels connect individual active nodes into a complete network. Different types of point-to-point and shared medium channels can be used as appropriate.
- New applications that make use of the active network are developed by subclassing the abstract class `Application`. Applications contain the functionality implemented by traditional distributed applications and operating systems at end-systems, for example, `vat`, `nv`, `wb`, or the combination of HTTP and TCP.
- New services can be developed at any time by subclassing the abstract classes `Capsule` and `Protocol`. The subclasses describe the new types of capsules and their organization into protocols, and how they should be forwarded at active nodes in terms of the API exported by the class `Node`. Unlike other code, this code is transferred between active nodes by the code distribution protocol when the service is used.

Each node is maintained in its own Java virtual machine, typically as a user-level Java process running under a Unix-based operating system. Application instances are run within the Java virtual machine of the active node to which they are locally connected. With this setup, the toolkit can be used to construct small networks based on a physical topology by running one node per router, and to emulate larger networks based on a virtual topology by running many nodes per host.

Note that the architecture itself is in no way tied to Java. I chose Java because it was convenient for rapidly prototyping an evolving design, provided safety and mobility infrastructure, and is likely to improve in performance to be more competitive with compiled

[2]Abstract classes are those that can only be used via subclasses that complete their implementation. They are denoted by the keyword `abstract` in Java.

Method	Description
`int getAddress()`	Get local node address
`ChannelObject getChannel()`	Get receive channel
`Extension findExtension(String ext)`	Locate extended services
`long time()`	Get local time
`Object put(Object key, Object val, int age)`	Put object in soft-store
`Object get(Object key)`	Get object from soft-store
`Object remove(Object key)`	Remove object from soft-store
`void routeForNode(Capsule c, int n)`	Send capsule towards node
`void deliverToApp(Capsule c, int a)`	Deliver capsule to local application
`void log(String msg)`	Log a debugging message

Table 3.1: Node API exported to capsule forwarding routines

Exception	Description
`ResourceLimitException`	Too few capsule resources for operation
`TimeLimitException`	Forwarding has run too long
`NoSuchRouteException`	No default route for sending
`NoSuchApplicationException`	No local application for delivery

Table 3.2: Exceptions during capsule forwarding

environments as its runtimes mature. It has facilitated the exploration of the design space at the cost of higher absolute performance. Higher performance implementations are a natural step towards deployment once the design and use issues are understood. One such effort that explores this direction is PAN [Nygren, 1999], and a further approach would be to execute binary proof-carrying-code within an operating system kernel.

In the remainder of this section, I describe the important features of the key classes in detail. This is done in a bottom-up fashion, beginning with the node API available to capsules, then capsule and protocol classes themselves, followed by applications. Finally, extensions and channels are discussed, along with the supporting conventions and tools that allow complete active networks to be created and managed.

3.1.1 Node

The `Node` class represents the runtime of a single network node, including its soft-store and code distribution system. It exports its services to:

- capsules, to execute their forwarding routines.
- applications, to register protocols and send and receive capsules.
- node extensions, to augment local node services.

These activities are logically separate and have different protection requirements. For example, capsule activity must be contained, while extensions benefit from more relaxed access to the node. Only the node API available to capsules is described in this section. Other APIs that support applications and extensions are described in their respective sections.

The node API exported to capsules is listed in Table 3.1. Capsules can invoke these calls during the execution of their forwarding routine, which is passed a reference to the local node. The calls fall into three categories.

Functions such as `getAddress()` and `time()` return information about the local active node environment. The `getChannel()` method returns the channel from which the capsule currently being processed was received, or null if the capsule was created locally. The `findExtension()` method returns a handle to a named extension that augments the node API if it is locally installed, or null otherwise. Many similar functions, such as those that enumerate the local default routing table or examine forwarding statistics, could be added as they are needed in practice.

The next category of functions manipulate the soft-store, a repository of application-defined objects that are cached for a short interval and then expired. `put()` enters an object, possibly a capsule, into the store after decrementing the resource limit on the capsule currently being forwarded. `get()` looks up an object in the store and returns a reference to it. A third function, `remove()` purges the store of an object. There is no guarantee that an object stored during the processing of one capsule will still be retained during the processing of subsequent capsules. This is because periods of high demand can cause the node to cycle through its available storage relatively quickly. To simplify programming, however, objects entered into the store while a capsule is being forwarded are guaranteed to be retained at least until the forwarding is completed.

In the store, objects are cached under application-defined keys and aged for application-defined intervals, after which they are evicted from the store if still present. Caching serves to retain objects that are in regular use to improve performance, while aging guarantees that stale state will not be retained in the network. The store separates objects by service (using the protocol type) so that network programmers do not need to worry about namespace collisions between services. Within a service, programmers must separate the state of concurrent sessions by using whatever naming scheme is appropriate. For example, a multicast service might key routing data by group address.

The remaining functions control the flow of capsule processing and its propagation to other parts of the network. `routeForNode()` forwards a copy of a capsule towards a node using default routes, decrementing the remaining resource limit in the process. Custom routing services typically express their routes in terms of default routes, since this is a robust way to navigate the network. `deliverToApp()` invokes an upcall on a local application to deliver a capsule; the application must make a copy of the capsule if it needs to be referenced after the upcall returns. If neither of these functions are called before the forwarding routine completes, the capsule is discarded. `log()` signals informational or error messages to the node. This is currently handled by writing them to the console to facilitate debugging, though it would be straightforward to extend this processing to generate an ICMP-like message.

There are also a small number of errors that can occur when calling these methods during forwarding. When one occurs, an error is delivered as an exception to the capsule forwarding routine making the call. This allows it a chance to recover. The exceptions are listed in Table 3.2. If the exception is not fully handled and propagates beyond the forwarding

Method	Description
`int getSrc()`	Get source address
`int getDst()`	Get destination address
`void setDst(int address)`	Set destination address
`int getResources()`	Get remaining resources
`void prime(Capsule parent)`	Transfer resources to fresh capsule
`int getPrevious()`	Get previous active node address
`byte[] getCapsuleID()`	Get capsule type
`byte[] getGroupID()`	Get code group type
`byte[] getProtocolID()`	Get protocol type

Table 3.3: Capsule API

routine, then it is handled as a `log()` call and capsule processing is cleanly terminated.

Method	Description
`short getSrcPort()`	Get source port
`void setSrcPort(short port)`	Set source port
`short getDstPort()`	Get destination port
`void setDstPort(short port)`	Set destination port
`ByteArray getData()`	Get payload
`void setData(ByteArray data)`	Set payload
`DataCapsule(short sp, short dp, int da, ByteArray p)`	Constructor

Table 3.4: DataCapsule API

3.1.2 Capsule

Subclasses of `Capsule` are used to control how packets are processed within the network. The network programmer must write a subclass for each different type of capsule. Objects of the subclass are then used to represent and manipulate the corresponding type of capsules as they are forwarded through each active node.

The abstract base `Capsule` class defines the minimal set of features that all capsules must possess. Capsule header fields are manipulated with the methods listed in Table 3.3.

The source and destination addresses are manipulated as 32 bit integers, with the class `NodeAddress` providing routines to format them in IPv4-style notation. Only the destination address can be updated. The source address is set to the address of the address of the node at which the capsule is injected into the network, and is null for capsules that have not been sent.

The `getResources()` method returns the remaining resources available to the capsule. Resource limits are initialized by an application when a capsule is created and before it enters the network layer. They are decremented by active nodes before a capsule sends a copy of itself or another capsule, or places an object in the soft-store. In the case that one capsule creates another within the network layer, the `prime()` method is used to initialize

Method	Description
`void evaluate(Node n)`	Perform forwarding
`Xdr encode()`	Encode capsule to representation
`Xdr decode()`	Decode capsule from representation
`int length()`	Length of representation
`<default constructor>`	Call superclass
`byte[] mid()`	Return capsule type
`byte[] gid()`	Return code group type
`byte[] pid()`	Return protocol type

Table 3.5: Capsule Subclass Framework

the resource limit of the new capsule, consuming a resource unit from the capsule being forwarded in the process.

The remaining methods return code distribution information. `getPrevious()` returns the address of the previous active node encountered by this capsule. `getCapsuleID()`, `getGroupID()` and `getProtocolID()` return the types corresponding to the capsule, code group, and protocol, respectively. Types are represented as arrays of bytes, and the class `TypeID` provides routines to manipulate them.

As well as the minimal `Capsule` baseclass, ANTS provides a slightly fuller-featured capsule class that will often be convenient for network programmers to use directly or to subclass. The `DataCapsule` class provides UDP-like service between applications. It provides the additional methods listed in Table 3.4. Source and destination applications at nodes are identified with port addresses, which are analogous to TCP/UDP port numbers. They are manipulated with the `getSrcPort()`, `setSrcPort()`, `getDstPort()` and `setDstPort()` methods. A payload is carried as an uninterpreted array of bytes. It is manipulated with the `getData()` and `setData()` methods. New capsules of type `DataCapsule` are constructed by specifying the payload and source and destination addresses. Within the network, forwarding follows default routes until the destination node is reached, at which point the capsule is delivered to the indicated application.

New subclasses of `Capsule` written by the service developer must override the capsule framework methods listed in Table 3.5. The active node runtime depends on these methods to manipulate all types of capsule.

Forwarding is customized by overriding the `evaluate()` method. It is invoked at every node the capsule visits and passed the local node as a parameter in order to access node services. It can handle errors that arise by catching exceptions, and it must terminate within the processing time permitted by the active node. While it is running, the active node guarantees that the concurrent execution of other capsules will not be observed. This might otherwise occur, for example, through changes in the contents of the soft-store. Besides the node API, the only methods `evaluate()` may access are those of other capsule subclasses in the same code group.

New types of capsule will typically include instance variables that represent the additional header fields used within the network to implement the new service. The `encode()`, `decode()` and `length()` methods convert these instance variables to and from an exter-

Method	Description
`void startProtocolDefn()`	Begin new protocol
`void endProtocolDefn()`	End current protocol
`void startGroupDefn()`	Begin new code group
`void endGroupDefn()`	End current code group
`void addCapsule(String name)`	Add capsule to current group
`void addHelperClass(String name)`	Add non-capsule to current group

Table 3.6: Protocol API

nal representation. This is needed to transmit them across the communication channels that link nodes. `encode()` initializes a linear representation from a capsule object for network transmission, `decode()` reverses the process, and `length()` indicates the buffer requirements. Each method is written in a stereotyped fashion using the `Xdr` class, which implements an external data representation library. Unfortunately, the task of writing this code is error-prone. This is because unless the methods agree exactly in all classes along the implementation hierarchy the contents of instance variables will be jumbled as the capsule crosses the network. A better solution for future versions of the toolkit is to generate these methods automatically at nodes from an annotated class description.

Finally, some standardized constructors and initialization methods are needed. This code provides no input from the network programmer, but rather is boilerplate that could not readily be provided otherwise. A default constructor (having no arguments) is needed to provide a template for decoding, and should simply call the superclass constructor. Static initializers and related methods should link the capsule with its types, since the fingerprint process means that they cannot be embedded in the code source directly.

Subclasses of capsule can also provide methods and constructors for the convenience of applications. Thus code that manipulates capsules falls into two distinct categories: code that supports forwarding within the network; and code that supports applications. As well as being logically distinct, these types of code are subject to different requirements. In particular, code that forwards capsules must be transferred across the network by the code distribution protocol and restricted in its operation by the node security mechanisms, while application code runs locally and with application privilege. Unfortunately, the current implementation forces the programmer to combine these two roles in a single capsule class, effectively making application code subject to network restrictions. It is not clear how best to improve the situation, since artificially dividing the code into separate classes would result in duplication and potential consistency problems. In future versions of the toolkit, an automated mechanism for producing both types of code from a common code base would be useful.

3.1.3 Protocol

To define a new service, network programmers must write a protocol class as well as writing capsule classes. Subclasses of the `Protocol` class are used to specify the organization of capsule and other classes into the code groups and protocol structures that are needed within the network.

Method	Description
`void attachNode(Node n)`	Connect to local node
`Node getNode()`	Get connected node
`short getPort()`	Get connected port
`int getDefaultResources()`	Get resource limit
`void setDefaultResources(int l)`	Set resource limit
`void register(Protocol p)`	Register protocol
`void unregister(Protocol p)`	Unregister protocol
`void send(Capsule c)`	Send capsule via node with default resources
`void send(Capsule c, int l)`	Send capsule via node
`void receive(Capsule c)`	Receive capsule from node

Table 3.7: Application API

Each new protocol must provide a default constructor that groups capsules as appropriate. The methods listed in Table 3.6 are provided for this purpose. The method `addCapsule()` is used to add a given capsule class or base class to the code group that is being defined. Helper classes, which are not subclasses of `Capsule` but are needed inside the network to help to manage application-defined objects in the soft-store, can also be added with the `addHelperClass()` method.

3.1.4 Application

Applications are the independent entities that make use of ANTS network services. They are constructed by subclassing the `Application` class. This is a container for end-system processing that provides a small API for connecting to the local node, registering protocols, injecting capsules into the network and receiving capsules from the network. An application runs within the same address space as the local node.

The `Application` class provides access to the network via the methods listed in Table 3.7. The first set of methods is used to connect to the local node and access the node and application port addresses, as well as establish the default resource limit that is stamped on each outgoing capsule.

To use a new service, application code first creates an instance of the corresponding protocol and registers it with the local node using the `register()` method. This informs the network of the protocol definition, and allows the local node to obtain the associated capsule code from the local filesystem to bootstrap the code distribution process. Once complete, applications can send and receive capsules belonging to the new service. The application should unregister the protocol when it is finished.

The `send()` method is used to inject a capsule into the network via the local node, stamping it with a resource limit and setting the source address in the process. If no resource limit is specified, an application-controlled default value is used. Once sent, the capsule is the property of the active node, and should not be referenced by the application again. Inside the active node, it is forwarded as appropriate by calling its `evaluate()` method. A capsule is removed from the network with the `receive()` method. This method is invoked as an upcall. It is overridden by the application and called by the node when a capsule is available. Because this method is run with the node thread and is synchronized to prevent further capsules from being received, it must complete quickly, queuing the capsule for later processing if necessary. It must also copy the capsule if it will be referenced at a later time, since the capsule remains the property of the active node after the call completes.

Since this research is focused on the structure of network services rather than the structure of applications, I have provided only a simple and low-level interface between applications and the network. In addition to the buffering and control semantics described above, applications support a single port and application code is run in the same protection domain as the active node. In future versions of the toolkit, a higher-level interface that includes multiple ports and buffer decoupling would be convenient for network developers.

3.1.5 Extension

Extensions allow a node administrator to augment the built-in services of a node with privileged code or large components that are not readily transferred using the code distribution protocol. In ANTS, extensions model the heterogeneity of the network. They are installed independent of the deployment of a new service, and represent capabilities that may or may not be present at a given active node. For example, Lehman implemented a `GroupsManager` extension to support group communication. Other candidates include richer schedulers, such as those which would support the IETF notions of Integrated Services and Differentiated Services, as well as various compression and transcoding libraries, such as those used in an active services prototype [Amir *et al.*, 1998].

New extensions are developed by subclassing the `Extension` class to provide whatever methods are appropriate. The extension is then installed on a particular active node as part of local configuration. Since extension code is privileged within the active node, the developer must take care to separate public methods for use by capsules from other methods. Because the node implements security mechanisms that restrict capsule behavior, extensions need to instruct the node to grant capsules access to their capabilities. Two methods on the node assist extension developers in this task. Both are typically called from extension code during system initialization.

- `attachExtension(Extension e)`, which informs the node of the given extension. Capsule forwarding routines are then able to locate the extension with the node method `findExtension()` and use its services.
- `exportClass(Class cl)`, which instructs the node to grant capsules access to additional classes. This allows the API of the new extensions to use helper classes, such as associated data structures.

3.1.6 Channel

Channels provide an interface to a link layer, allowing individual nodes to be connected into a network via point-to-point or shared medium technologies. New types of channel, corresponding to new link layers, can be developed by subclassing the abstract class `Channel`. The `UDPChannel` is used to model hosts connected by UDP as a single shared medium channel. This arrangement provides a flexible means of constructing active networks on

Method	Description
`int getAddress()`	Get interface address
`int getBandwidth()`	Get bandwidth (bps)
`int getLatency()`	Get latency (ms)
`int getMTU()`	Get maximum packet size (bytes)
`int getBER()`	Get bit error rate (reciprocal)

Table 3.8: Channel API

top of the Internet. Small networks can be constructed by running one node per router and connecting the nodes with UDP channels, since UDP spans hosts across the network. Larger networks can be emulated by running many nodes per machine and connecting them with UDP channels, since each host can support many UDP endpoints.

Each type of channel provides the methods listed in Table 3.8 to allow service developers to explore the local environment. Note that many of these characteristics are not well defined for virtual channels such as UDP, since they can vary considerably for different capsules, nor for situations where active nodes are separated by many non-active nodes, since local channel information may not be indicative of the compound channel between active nodes. For these reasons, service developers should treat channel characteristics as hints.

3.1.7 Configuration Management

As a practical matter, experimenting with an active network requires that a network topology be constructed. ANTS provides tools and conventions to automate this process.

The `ConfigurationManager` tool can be used to create and initialize the local machine portion of an active network. It requires a description of the network in the form of a configuration file, containing one or more lines for each network entity (node, channel, application, and extension) in a simple textual format. This simplifies management, since one file can be used to specify the entire topology.

To permit the generic handling of network entities, each of the `Node`, `Channel`, `Application` and `Extension` classes is derived from the abstract class `Entity`. Subclasses must adhere to a number of conventions. Two key conventions are:

- The `setArgs(KeyArgs args)` method is called by the `ConfigurationManager` to allow the entity to process command line arguments that are passed from the configuration file.
- The `start()` method is called by the `ConfigurationManager` to complete initialization and allow the entity to commence operation. It will typically create a thread to manage ongoing processing.

In addition, the `Entity` class also provides methods that its subclasses can use to signal informational, warning, and error messages to a common point of collection, filtering and display.

3.2 A Simple Example

To provide a concrete example of the toolkit in use, I describe how it can be used to introduce a path Maximum Transmission Unit (MTU) discovery service. Path MTU discovery is an important service because packet size is known to have a significant effect on performance: if packets are smaller than necessary then performance suffers due to per packet costs; if packets are larger than the channel frame sizes then performance suffers due to fragmentation costs [Kent and Mogul, 1987].

In the Internet today, path MTU discovery is performed in an *ad hoc* manner by triggering error messages and adapting [McCann *et al.*, 1996, Mogul and Deering, 1990]. In an active network, path MTU discovery can be performed by supplementing this mechanism with a service that queries the attributes of network paths directly. This requires only that a new kind of capsule and protocol be defined, and then an application can make use of the service at any time. The source code for these classes (taken from the ANTS distribution and reformatted for presentation) is presented below.

3.2.1 `PMTUCapsule` Class

The purpose of the `PMTUCapsule` class is to define a new kind of capsule that records the smallest MTU encountered so far (representing the current path MTU) as it travels through the network. This information can then be passed to a remote application and conveyed back to the sender via an appropriate channel, for example, the acknowledgment stream in the case of a TCP connection. Since the path MTU discovery capsule will determine the path characteristics only where it encounters active nodes, the resulting path MTU should be treated as an estimate. This does not complicate the use of the information because the dynamic nature of Internet paths means that path MTUs must always be treated as an estimate for the current path. To the extent that active nodes are likely to be found in the lower bandwidth region of the network and these regions typically have smaller MTU restrictions, the estimate is likely to be useful. Some techniques, such as checking the outgoing interface MTU as well as the incoming one, can improve the quality of the estimate, but are omitted from this example for simplicity.

The source code for this class is shown in Figure 3-2. It extends the class `DataCapsule` in order to inherit the behavior of capsules carrying a data payload. The bulk of the additional functionality is captured in the `evaluate()` method. This method is invoked at every active node the capsule passes through, and is passed the local node object as its single parameter in order to access local services. The method updates the current path MTU estimate by comparing it against the MTU of the incoming interface. The estimate is carried with the capsule and accessed by the instance variable `pmtu`, by comparing it against the MTU of the incoming interface. The logic of this comparison is slightly complicated by the need to handle the case in which the capsule has not yet been sent across a channel.

Because this capsule class includes an additional header field that is carried with it across the network, as is typical of new capsule types, the `encode()`, `decode()`, and `length()` methods are overridden. They include the `pmtu` field in the encoding and decoding process.

Most of the remaining code is boilerplate. A default constructor is required for the system to manipulate this type of packet. The static fields are used to initialize the capsule, code

```
package apps;
import ants.*;

public class PMTUCapsule extends DataCapsule {
  final private static byte[] MID = findMID("apps.PMTUCapsule");
  protected byte[] mid() { return MID; }

  final private static byte[] GID = findGID("apps.PMTUCapsule");
  protected byte[] gid() { return GID; }

  final private static byte[] PID = findPID("apps.PMTUCapsule");
  protected byte[] pid() { return PID; }


  public int pmtu = 64*1024-1;  // begin with the largest possible datagram size

  public int length() { return super.length() + Xdr.INT; }

  public Xdr encode() {
    Xdr xdr = super.encode();
    xdr.PUT(pmtu);
    return xdr;
  }

  public Xdr decode() {
    Xdr xdr = super.decode();
    pmtu = xdr.INT();
    return xdr;
  }

  public boolean evaluate(Node n) {

    // check the incoming channel and perhaps reduce our estimate
    ChannelObject in = getChannel();
    if (in != null) {
      int last = in.getMTU();
      if (pmtu > last) pmtu = last;
    }

    // and continue towards the destination
    if (n.getAddress() != getDst()) {
      return n.routeForNode(this, getDst());
    } else {
      return n.deliverToApp(this, getDstPort());
    }
  }

  public PMTUCapsule() { }
  public PMTUCapsule(short dpt, short spt, int da, ByteArray p) { super(dpt, spt, da, p); }
}
```

Figure 3-2: Source code for the class `PMTUCapsule`

```
package apps;
import ants.*;

public class PMTUProtocol extends Protocol {

  public PMTUProtocol() throws Exception {
    startProtocolDefn();

    startGroupDefn();
    addCapsule("apps.PMTUCapsule");
    endGroupDefn();

    endProtocolDefn();
  }
}
```

Figure 3-3: Source code for the class `PMTUProtocol`

group and protocol types with their fingerprint-based identifiers when the code is loaded within the network.

3.2.2 `PMTUProtocol` Class

The purpose of a protocol class is to organize capsule classes into code groups so that they can be manipulated by applications and nodes within the network. In the case of the path MTU discovery service, this is straightforward because there is a single capsule class. This class is wrapped in a single code group that forms the entire protocol, as shown in Figure 3-3.

3.2.3 `PMTUApplication` Class

An application running on an active node can use the path MTU discovery service at any time. This is accomplished in three steps:

- Obtain the code that implements the `PMTUCapsule` and `PMTUProtocol` classes from a directory service.
- Create an instance of `PMTUProtocol` and register it with the local active node, causing the service code to be loaded into the local node.
- The application is then free to create capsules that are instances of the class `PMTUCapsule`, and send them into the network where they will receive the path MTU service.

The source code for a simple application that does this is shown in Figure 3-4. It is shown for completeness, though it consists mainly of Java GUI code to send a capsule in response to a button click, and to display the path MTU when a capsule is received. In a real application, path MTU information would also have to be propagated from the receiver back to the sender.

```
package apps;
import ants.*;
import utils.*;
import java.awt.*;

public class PMTUApplication extends Application {
  final public static String[] defaults = {};
  int target;
  Button send;
  Label receive;

  synchronized public void receive(Capsule cap) {
    super.receive(cap);
    if (cap instanceof PMTUCapsule) {
      PMTUCapsule pcap = (PMTUCapsule)cap;
      receive.setText("PMTU is "+ pcap.pmtu +" bytes");
    }
  }

  public boolean handleEvent(Event evt) {
    if (evt.id == Event.ACTION_EVENT && evt.target == send) {
      send(new PMTUCapsule(port, port, target, null));
      return true;
    } else
      return super.handleEvent(evt);
  }

  public void setArgs(KeyArgs k) throws Exception {
    k.merge(defaults);
    for (int i = 0; i < k.length(); i++) {
      if (k.key(i).equals("-target")) {
        target = NodeAddress.fromString(k.arg(i));
        k.strike(i);
      }
    }
    super.setArgs(k);
  }

  public void start() throws Exception {
    getNode().register(new PMTUProtocol());
    setLayout(new BorderLayout());
    resize(400, 200);
    send = new Button("Send PMTU to "+ NodeAddress.toString(target));
    add("South", send);
    receive = new Label("", Label.CENTER);
    add("Center", receive);
    pack();
    show();
  }

  public PMTUApplication() throws Exception { super(); }
```

Figure 3-4: Source code for the class `PMTUApplication`

3.3 Major Implementation Components

In this section, I switch gears and describe the underlying implementation. Recall that the toolkit is written entirely in Java, without native methods and using only the standard Java libraries. Each node is run in a single Java virtual machine along with its locally connected applications. Nodes communicate with each other between virtual machines using the communication services provided by Java.

Rather than divide the discussion along class lines, I have grouped it by the major tasks that arose in the implementation, with the expectation that this is likely to provide the most insight into how to construct an active node. I further focus on the design issues that were not obvious (such as the node security mechanisms) and omit the remainder (such as the external data representation library).

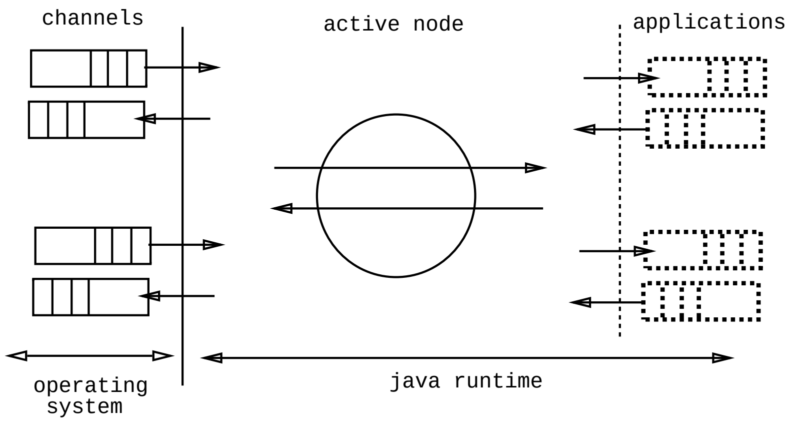

Figure 3-5: Control structure and buffering in the active node

3.3.1 Control Structure

A control structure knits the key classes together into a running active node that forwards capsules. Its design must accept packets from multiple channels and applications, forward them without allowing concurrent activity to expose intermediate states, and deliver packets to multiple channels and applications. It should accomplish this without starving any channel or application, and with a minimum of overhead. The design is further constrained to be that of a software-based implementation running on workstation or PC hardware.

Given that capsule processing routines are analogous to IP processing (in that they are intended to run to completion at the forwarding rate) a simple approach is to model the ANTS design on that of a Unix-based router. The relevant features of this design are that: buffering is used to decouple rates between channels and the node, and between the node and applications; and scheduling is trivially performed once per packet.

In the ANTS toolkit, this scheme is approximated in a lightweight fashion as shown in Figure 3-5. Since the underlying operating system provides buffering to match network and host processing rates, no additional buffering is provided as part of channels within the Java runtime. This reduces the complexity of the implementation considerably, at the cost of the inability to tune buffer sizes and observe queue overflow locally[3]. This tradeoff seems reasonable for a prototype, though in the longer term the ability to observe congestion related losses will be valuable for innovative protocols.

Since this research is focused on the structure of the active node rather than that of applications, applications are provided with a low-level interface in which buffering is their responsibility. Packets are delivered to the application with the node thread (that is, as an upcall) and sent by stealing the application thread. This design requires that the node trust its applications but does allow for efficient operation. Normally, buffering and a control handoff would be used to cross this boundary. Instead, an upcall allows the application to avoid a control handoff if the receive processing completes quickly. Similarly, the copying of capsules can be avoided by the application if it is known that the service being used is compatible with "move" semantics.

To schedule packets, a separate thread is run per channel and per application, with capsule processing serialized by synchronizing it on the node[4]. To reduce overhead, this is done once per capsule, rather than once per node API call. The channel threads loop forever receiving and then forwarding packets, blocking when no input is available. Thus it is code in the channel thread that receives packets, evaluates them in the context of the node, and handles any errors that arise. In our prototype, errors simply cause capsule processing to be terminated, though it would be straightforward to extend this to an error message scheme analogous to ICMP. The application threads are under the control of the application programmer, and interact with the node only when they are used to send a packet.

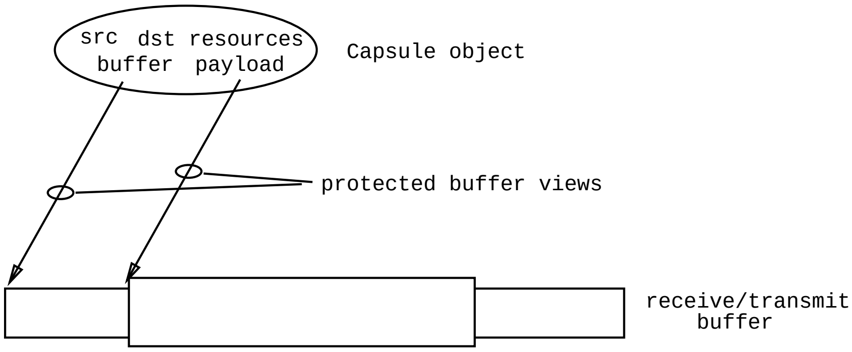

Figure 3-6: Collapsed receive, `Capsule` and transmit representations

3.3.2 Data Paths

Experience with network protocol implementations provides evidence that the organization of the data paths is important in order to achieve high throughput. Copying across protection boundaries is generally an expensive operation. In Java, allocation is also potentially

[3]In Unix, the system call `setsockopt()` allows control of buffer size, and non-blocking output allows output queue overflow to be observed. These features are not available in JDK1.1.

[4]An obvious alternative is to poll for input. This is not readily supported in Java because there is no notion of non-blocking input.

expensive because it requires that object memory be initialized (by zeroing) and may invoke the garbage collector.

In the case of the active node, the data paths receive capsules from channels and applications and process them within the node. There they can be sent out on other channels, delivered to applications, entered into the soft-store, as well as any combination of these operations, including multicasting. The design of the data path is more complicated than a typical network implementation, for example, TCP/IP within a BSD-based kernel, for two reasons:

- Capsules are manipulated as objects within the node, rather than as linear buffers. Some copying may be required to convert between these forms, depending on the underlying representation of Java objects and in the case that fields change size, as is typically the case when the payload is substituted. Further, different parts of the capsule are subject to different protection requirements. The `evaluate()` routine, for example, must not be able to change the value of the resource field, but the node must be able to decrement this field as resources are consumed.

- Different types of packet are represented with different types of capsule. This makes cheap allocation schemes more complex. Further, allocation ultimately depends on the capsule forwarding routines themselves. For example, one program may wish to preserve a copy of the capsule in the soft-store as well as forward it, while another may not. Because these effects are not known ahead of time, static allocation schemes are made more complex.

The ANTS data path design uses two strategies to minimize memory copying and allocation.

To remove unnecessary copying, the receive, object and transmit representations of capsules are largely combined into a single overlapping data structure. An example is shown in Figure 3-6. The `Capsule` object is kept relatively small by referencing, rather than including by value, large fields such as the payload. A special class, `ByteArray`, is used to express these references by exposing only a portion of the receive buffer used to decode the capsule and protecting the remainder[5]. The same buffer is later used to encode the capsule for transmission. Transmit buffer copying is minimized because only the changes caused by processing need to be copied from `Capsule` object back to the buffer. In the common case that the payload is at the same buffer position and has not been modified it need not be copied.

To remove unnecessary allocations, the design is specialized for two common cases: a packet being received from a channel and sent one or more times to another channel or application, but without being copied into the soft-store; and a packet being sent once by an application, again without being copied into the soft-store. These cases allow allocation and initialization of large buffers to be avoided by reusing one buffer for multiple capsules. In the first case, the receive buffer can be used to transmit the capsule and then to receive subsequent capsules because no copies of the previous packet are kept. In the second case, a single transmit buffer can be used, again because no copies are kept.

Besides the buffer objects, capsule objects must also be allocated and initialized. To balance efficiency and complexity, these remaining capsule objects are managed per packet in a

[5]Unfortunately, without runtime modification, this requires a set of array bounds checks in addition to those that Java already performs.

straightforward manner. As one strategy to reduce the number of objects that need to be allocated per capsule, primitive types are used for header fields where possible. For example, resource limits and node and port addresses are manipulated as integers, with utility classes such as `NodeAddress` providing methods to manipulate them as a special datatype[6].

3.3.3 Soft-Store

The design of the soft-store consists of an object-based API and an allocation mechanism that combines usage patterns and aging. (A protocol-based protection mechanism is also implemented in the soft-store, but described later as part of the node security design.)

One difficulty was to implement the combined allocation mechanism efficiently. This was done by aging information in seconds, at a much coarser granularity than the packet rate. Given this, aging can proceed as a background process that sweeps the soft-store, and so ensures that stale information is removed eventually, while caching policies can be enforced at the packet rate, and so ensure that fresh storage is available immediately when it is needed. In the current implementation, the caching policy is least-recently-used across the information stored by all protocols.

In order to support a high-level view of network programming, the soft-store is designed around an interface that manipulates generic objects. Java encourages this style of programming with safe dynamic casts and the use of generic objects in its standard libraries. This generality makes the soft-store powerful for the network programmer. For example, lists of capsules can be stored as easily as routing information. By the same token, it complicates node resource management in three respects that are only partly addressed by the design.

- The soft-store interacts with the design of the data paths because buffers cannot be reused while the corresponding capsule is held in the store. However, it is difficult to efficiently discern whether a buffer has entered the soft-store via a graph of objects. The conservative strategy used in ANTS is that capsule buffers are not reused if an object whose type is not explicitly known to permit buffer reuse is entered into the soft-store.
- The store is managed by the number of objects, irrespective of their size. This is a consequence of the lack of Java facilities for determining the size of an object.
- Object are evicted in a straightforward manner by removing the object reference from the store. However, this will not free actual memory until other references to the object via the soft-store have been removed and the garbage collector runs. Again, without runtime support this is difficult to remedy.

The net result of these complications is that the soft-store has only approximate control of the underlying memory pool, a defect that could be exploited to attack the node. However, I have deferred refining the soft-store to address these complications until there is greater experience with network programming. On the one hand, it is clear that with runtime support for object sizes and a restricted API (such as one that uses a fixed set of concrete

[6]Arguably, exposing the implementation in this fashion is a mistake. But the performance implications and lack of an alternative, such as `typedef`, make the choice difficult.

types that do not contain references) all of these issues can be addressed. On the other hand, a more flexible API may prove useful, in which case strategies for supporting it safely should be explored. For example, API modifications that limit or expose references or specify what should happen when one object of a graph is evicted may be able to combine flexibility with safety.

3.3.4 Code Distribution

The implementation of the code distribution system is comprised of a demand loading protocol, a code cache, and a secure dynamic loading system. I describe each in turn.

To provide as modular an implementation as possible, the demand loading protocol (class `DLProtocol`) is modeled after a regularly loaded protocol. It is unusual in three respects that relate to its function:

- To trigger the protocol in response to unrecognized packets, the channel is modified to intercept such packets and encapsulate them within special demand loading capsules. These capsules (of class `DLBootstrapCapsule`) are then passed to the node as regular capsules and result in a demand loading exchange. In this manner, the channel reception and decoding process always appears to succeed to the node, with the result that the demand loading knowledge is localized.
- Because code distribution is used to bootstrap the customization process, the demand loading capsule types must be universally known. In theory, the fingerprint process provides such an identifier. In practice, the process is tied too heavily to the source code (which depends on that of the node) and compilation process (which can produce slightly differing bytecode on different runs). This conflict is resolved with the notion of built-in protocols (class `BuiltinProtocol`) that are never loaded across the network and whose types are constructed from hard-wired version strings.
- The implementation is privileged in ways that regular protocols are not. First, the protocol interacts directly with internal node state to manage the code cache. Second, the protocol is permitted to manufacture new resources within the network layer as they are needed to carry out the demand loading exchange. Third, when the missing code is loaded, the associated capsules are run recursively, from within the running of the demand loading protocol. Calling across protocols is not otherwise allowed, since it poses security hazards. In all three cases, `package` level protection in Java is used to grant restricted access.

The code cache structure itself is fairly straightforward. It is separated from the soft-store, and specialized to store and lookup capsule, code group and protocol information. The contents of the cache are managed in a least-recently-used fashion.

Finally, a secure dynamic loading system handles the loading and linking of new protocol code, interpreting calls from capsule code to other classes to restrict their reach. Calls against the node API or to permitted utility classes (for example, `java.util.Vector`) are unambiguous and are allowed. A more subtle issue is that of supporting calls between capsule code. These occur when one type of capsule creates another type within the network. These calls are expressed in the bytecode with local names (the Java class names from the originating host) rather than global names (the capsule types). To interpret local names correctly and not permit unintended access, linkage between loaded code is confined to a single protocol. This scheme is implemented (class `DLClassLoader`) by customizing the class loading abilities of Java.

3.3.5 Default Routing

The current implementation of default routing forwards capsules along shortest path routes. These routes are generated by a distributed process that provides essentially the functionality of the original Routing Information Protocol (RIP) [Hedrick, 1988] used in the Internet. That is, update messages are exchanged between neighboring nodes to support a distance vector calculation. These messages are sent periodically and when triggered by changes in routes. The distance vector calculation includes heuristics (split horizon processing with poison reverse) that increase the rate of convergence of the calculation and its stability.

The route maintenance is modeled in the node as one protocol, class `RouteProtocol`, that is built-in in the same sense as demand loading. It relies on privileged access to manipulate routing information that is then used by the node, as well as to run a thread that periodically exchanges route update capsules with neighboring nodes.

The implementation itself is unremarkable. Of greater interest is a design issue that was raised in the process: should default routing be part of the infrastructure, or provided as an introduced service? All other things being equal, an introduced service would highlight the value of the architecture. Despite this preference, I opted to build default routing into the infrastructure as for three reasons:

- Default routing will often be used by applications that simply wish to exchange data without customized processing. As such it should be both efficient and convenient to use. A built-in service can be exposed directly as part of the node API to achieve these goals.
- Default routing will often be used as the basis for constructing special-purpose routes. As such, it is essentially a bootstrapping service that must always be available. An introduced service that relies on information in the soft-store may not always be able to provide reasonable routing information. Conversely, with a built-in default routing service, introduced services that construct their own routes can fall back to default routes if their routes are temporarily unavailable.
- Given that alternative routing services are a potential use of an active network, the best means of expressing them is not *a priori* obvious. What is clear is that a straightforward translation of traditional routing protocols, either distance vector protocols such as RIP or link state protocols such as OSPF, will not work. This is because such protocols rely on features such as timers, or assumptions such as that all nodes support the protocol, that ANTS does not provide or make, respectively.

3.3.6 Node Security

Code that runs within the active node falls into different protection domains: one for the active node runtime, and one for each loaded protocol. Calls between these domains must be restricted, and the resource usage of individual domains must be limited.

To accomplish these goals without sacrificing performance, the features of Java are used to implement multiple protection domains within a single virtual machine, to the extent that this is possible. In general, a combination of mechanisms is used to restrict capsule activity, while the active node runtime is not limited in any way.

As described for code distribution, calls between capsule code and other protocols or the active node runtime are restricted at link-time by using the `ClassLoader` infrastructure. This strategy is supplemented in two ways. First, a dynamic checking mechanism is used at a finer granularity for those classes that must be accessible to capsules but also contain sensitive methods that should not be called by capsules. For example, capsules can call node API methods, but not the node constructor. To intercept calls that are not permitted, the stack inspection mechanisms of Java are used to verify that the call has passed through a mediating class, such as a trusted extension, rather than come directly from capsule code. This is similar to the privilege mechanism being introduced as part of the JDK1.2 security architecture [Gong *et al.*, 1997, Gong and Schemers, 1998]. Second, capsule fields that cannot be manipulated directly by forwarding routines are protected with methods. For example, the source address is private and manipulated with an accessor. Raw access to the capsule buffer itself is restricted with protected buffer ranges, which were described as part of the data path design.

The resources consumed during capsule processing are limited in two ways. Accidental long-running computations are broken with a coarse watchdog timer. Protection at a finer granularity is advisable, but requires loadtime or runtime modifications. Node storage is tracked by using a context record that is maintained by the node while the forwarding routine is running. This prevents capsules from lying about the protocol to which they belong. It is used by the appropriate node API calls to locate and decrement the capsule resource limit field, and to partition the contents of the soft-store. Other memory resources, such as stack and heap space, are not protected in the prototype.

Finally, in addition to these mechanisms the integrity of the node runtime is based on the assumption that bytecode is well-formed in a number of ways beyond what is required for type safety. These properties could be checked at load-time during the bytecode verification phase, but this cannot easily be accomplished without modifying the Java runtime.

- Until they are generated by the node itself, the `encode`, `decode`, and `length` methods must be well-formed to ensure that they will not tamper with the runtime or cause capsule invariants to be violated. In particular, they must correspond to each other exactly, begin with a call to the superclass method, and call no other methods except those in the Xdr library.

- Some boilerplate is needed so that capsules can be manipulated by the active node. The type of the capsule must be statically initialized to the correct value and not written subsequently. The default constructor must consist of a single call to the superclass constructor that passes the type of the capsule.

- Various exceptions are used by the node in response to error conditions to signal the capsule code that it must terminate, for example, when the capsule code has been running too long or attempts to access a class for which it lacks permission. These indications can be used as an opportunity to leave data structures in a consistent state, but must not be blocked[7].

3.4 Aspects Not Addressed

In the current version of the ANTS toolkit, no attempt was made to implement several aspects of the architecture since their omission seems unlikely to affect its evaluation. This section documents those aspects, how their implementation might be handled in future releases, and how their absence impacts the usefulness of the toolkit.

3.4.1 Non-Active Nodes

The ANTS architecture models the network as a graph of active and non-active nodes in which the customized forwarding routines are run only at active nodes. Since the focus of this research is on the active nodes and the non-active ones are essentially transparent because they perform default IP forwarding, the toolkit does not implement them. In a future version of the toolkit, it would be possible to reuse existing IP routers as non-active nodes by combining the IP packet and capsule formats as described in Chapter 2. In the meantime, to be faithful to the architecture, network programmers must be careful to ensure that the customized forwarding routines do not rely on a fully-active network by failing to maintain a valid destination address at all times.

3.4.2 Node Security Policy

The ANTS architecture relies on a security policy to ensure that overall network resources are consumed in a reasonable manner. Service code is typically accepted for execution if it has been certified (by using a digital signature) by a trusted authority. Services that are not accepted receive default forwarding. In effect, the node is not active for them.

The toolkit, on the other hand, does not implement this kind of security policy. It can be modeled as having a null security policy that accepts all new services. In the same manner as non-active nodes, this decision does not affect the evaluation in terms of the kinds of new services that can be expressed. In practice, services that are not accepted may generate an overhead at otherwise active nodes, as demultiplexing based on their capsule types must be performed before their service is mapped to default IP forwarding.

3.4.3 Code Representation

Since the assumption of small service descriptions underlies the design of the code distribution protocol, compact code representations are important. The code distribution protocol transfers forwarding routines as the Java classfiles of the corresponding capsule subclasses. While bytecode representations are typically compact, the resulting transfer format is larger than necessary for several reasons:

[7] Java provides no sure way for one thread to control another. Signals are delivered to a running thread as exceptions and so can be caught.

- It is not compressed but amenable to compression. The use of gzip, for example, reduced code group size by at least 50% for the services presented in Chapter 4. A supplementary dictionary may lead to further gains because there is significant repetition between capsule classes, as they all have at least one set of interface classes and methods in common: the node API.

- It contains code that could be generated automatically at each node. Boilerplate such as default constructors can be generated with no extra information beyond that already in the classfile, while encoding and decoding methods need only knowledge of the new header fields. Removing this code reduced code group size by at least 30% for the example services.

- It can contain code that is not needed within the network, such as convenience constructors for use by applications. This is a consequence of the design of the toolkit. The amount of this code depends on the service developer, but was typically less than 10% for the example services.

Code size has not proven to be an issue in experiments with the toolkit to date. However, services that are developed in a production active network may be more complicated, and consequently larger. If and when size becomes an issue, the preceding observations can be used to obtain a more compact encoding in exchange for additional code processing.

3.4.4 Granularity of Capsule Processing

In the ANTS architecture, forwarding decisions are expressed in terms of customized processing routines. While these routines must capture all of the forwarding decisions that can affect the handling of a particular capsule, many forwarding decisions do not need to be recomputed for each capsule since they vary at a much coarser granularity. Forwarding decisions that vary with the capabilities of the local active node (such as the size of the cache or available computation) effectively need to be made only once, the first time the code is run at a particular node, since they will not vary afterwards. Other forwarding decisions will vary more often, but still infrequently compared with the packet rate. For example, when the forwarding of one type of capsule depends on information that is maintained in the soft-store by another type of capsule and changes only slowly (such as a custom route that is refreshed every 30 seconds), repeated soft-store lookups are wasteful.

In the current implementation, essentially all decisions are recomputed each time that a capsule is forwarded. As a special case, the outcome of the search for node extension components is saved as a static field. This design seems appropriate for a prototype. It provides a straightforward programming interface with which to explore the structure of new services. It effects the evaluation of the architecture only in that the local node performance will be underestimated.

In future versions of the toolkit, a more promising approach may be to use a dynamic binding mechanism based on events, for example, similar to the one used in [Pardyak and Bershad, 1996]. This would allow changes that occur as less than the capsule forwarding rate, for example, the presence of node extensions or information in the soft-store, to control which version of the forwarding method is invoked for a given type. The implementation may be efficient, since it overloads the existing demultiplexing stage to capture the dispatch decisions that now tend to occur within the `evaluate()` method. Unfortunately this approach was not straightforward to use with Java, since it lacks methods as first class objects[8].

Optimizing two special cases should also be worthwhile. First, switching between customized and default forwarding at a node is likely to be a common case. Built-in support may allow a capsule to receive default forwarding more efficiently, by omitting protection mechanisms that would otherwise be necessary. Second, many routines may simply forward capsules unless they are at the node given by the destination address (which of course can change en route). If capsules of this kind can be readily discerned, for example, from a bit in the type field, then they too can be handled with an efficient default forwarding mechanism when appropriate. This is reminiscent of the distinction between Hop-by-Hop and Destination options in IPv6 [Deering and Hinden, 1995].

3.5 Experience With Java

As a high level language that supports mobile code and emphasizes security, Java has supported the rapid prototyping and exploration of the design space of an active node. I close this chapter by describing how well or poorly aspects of Java have met the needs of the ANTS toolkit.

At the time of design, and still as of the time of writing, Java is arguably the language that most closely matches the combined mobility, security and efficiency needs of an active node. It allows compiled representations (carried as bytecodes) that originate from an untrusted party to be safely yet efficiently executed within the context of a trusted virtual machine. The key innovative feature of Java in this respect is bytecode verification. It enables a rapid load-time analysis to guarantee properties such as type-safety and lack of stack underflow or overflow. Together, these properties allow a much more efficient execution of the bytecodes than guarded interpretation, since many runtime checks can be omitted. Complementary infrastructure (particularly a controlled linking process, stack introspection, and checks build into the system libraries) allow the scope of untrusted code to be further restricted. As an added bonus, the availability of high-level language features such as automatic storage management and exceptions has allowed a simple yet powerful programming interface to be provided.

Conversely, without Java the simultaneous investigation of performance and security issues would have been greatly complicated. Binary code representations offer greater runtime efficiency, but are not yet compatible with security goals. Proof-carrying code, for example, is a promising technology, but is not mature enough to be of use. Software-based fault isolation is a more mature technology, but offers a form of safety that is less well-suited to frequent protection domain crossings than type-safety. The design of PAN [Nygren, 1999], a binary-based active node that investigates performance but sacrifices security, illustrates this tension. At the other extreme, an implementation based on interpretation would have met security but not performance goals. It would have been too far removed from the long

[8]Subsequent to the current implementation, JDK1.1 emerged, complete with a reflection package in which methods can be manipulated as first class objects. This facilitates the described approach, yet incurs a performance penalty that may negate its advantages.

terms goal of introducing services whose performance is comparable to that of native code to argue the performance implications with any conviction.

These advantages have come along with several disadvantages. The level of node performance has been relatively poor due to the design of existing runtimes and their operational constraints, especially the limited mixing of untrusted mobile with trusted stationary code, and lack of kernel-based runtimes. This situation is temporary and will change as new runtimes and improved just-in-time compilers emerge. More interesting are disadvantages that stem from the design of the existing Java virtual machine and its standard libraries.

Some performance overheads are the indirect result of language features that are not necessary for the active node. For example, Java emphasizes a programming style based on multiple threads and synchronization between them. Instead, a simpler polling and event-driven style would have been sufficient for the design and reduced runtime overhead. While Java does not require that multi-threading features be used, their availability has influenced the design of standard libraries and runtime such that their costs cannot readily be avoided. For example, there are no non-blocking network interfaces, which precludes a polling-style design in favor of multi-threaded one. Similarly, the object-oriented emphasis tends to result in more expensive implementations than would otherwise be needed because it is often promoted to the exclusion of other forms of abstraction. For example, the reuse of primitive types (such as integers for addresses) requires an object wrapper rather than a more efficient aliasing (such as `typedef` in C).

There is also a mismatch between the capabilities offered by Java and those required for systems programming. The layer of abstraction that Java imposes for portability makes it difficult to control memory and processing resources at a fine granularity. For example, object size and structure cannot readily be discovered, processing can be interrupted by garbage collection, and the lack of first class methods makes it difficult to efficiently rearrange bindings between events and code. There is also no easy way for trusted code to circumvent type safety mechanisms in situations where they result in unnecessary overhead. Conversions between different representations that can be accomplished with a cast in C requires object allocation, initialization and copying in Java. Other systems projects have extended high-level languages with comparable facilities, for example, SPIN introduced the `view` construct into Modula-3 [Hsieh *et al.*, 1996]. Another useful mechanism would be runtime support for array subsetting, which would enable regions of a buffer to be protected efficiently. These mismatches do not indicate tasks that cannot be accomplished in Java, but rather tasks that are not well supported. For example, introspection methods in JDK1.1 can be used to support first class methods, but the cost is high. Similarly, object size can be inferred by introspection, but again the cost is too high for this method to be used in practice.

Finally, while Java has directly contributed to security goals, it has fallen short of the ability to support multiple protection domains in a single virtual machine. It is difficult to separate effects in one protection domain from those in another since stack and heap memory pools are shared across all domains. Similarly, it is difficult for one domain to exert control over another, since there are no signaling mechanisms (such as thread methods or exceptions) that cannot be subverted. These difficulties are the focus of other research projects; see for example [Hawblitzel *et al.*, 1998]. An interesting approach not suggested elsewhere is to achieve security goals through an extensible bytecode verification process. For example, untrusted code could be examined at load-time to ensure that it does not catch various thread exceptions and so can be controlled by another thread without runtime overhead.

Chapter 4

Introducing Network Services

To demonstrate how the ANTS architecture can be used to introduce new services, this chapter describes two case studies: multicast and Web caching.

The case studies are complementary. Multicast represents a widely-accepted change in network service that has a relatively long history. Studying multicast thus looks backwards to shed light on whether the architecture can introduce the kind of services that have proven valuable. On the other hand, Web caching represents the impact that changing application patterns are having on the network today. The best way to incorporate caching into the network infrastructure is the subject of much ongoing research (and debate). Studying Web caching thus looks forward to shed light on whether the architecture is able to address the kinds of new network service that are required today.

Both services are discussed at some length, since they are intended to be realistic service candidates, rather than toy examples. I do not present new solutions to these particular problems. The goal of each study is to demonstrate how the active network approach can be applied. That is, I wish to show that ANTS facilitates the construction and deployment of new services, not that multicasting and Web caching should be tackled in a novel manner. Before presenting the two cases, I discuss network programming issues with an eye towards understanding how services should (and should not!) be written.

4.1 Programming with Capsules

Protocol design is a complex task. Processing that occurs within the network must provide good performance in the common case that packets are delivered in sequence and via the same route. The same processing must be compatible with correct application behavior in the cases that: packets are lost, corrupted, reordered or duplicated; nodes or links fail; and network routes vary. Paxon provides more information on the type and frequency of end-to-end packet dynamics that must be accommodated [Paxon, 1997].

By itself, an active network does not necessarily make the task of protocol design any easier. The freedom that the capsule model allows is useful for experimentation, but provides scant guidance to network programmers. It would be straightforward, for example, to construct services that rely on fixed topologies, or that only function correctly when the network is reliable.

The following guidelines have proven valuable for constructing modular, robust and efficient services. They are essentially the result of well-known protocol design principles applied to the capsule programming model. Adhering to them eliminates known design failures, and the principles can be seen at work in other protocols that employ additional processing at nodes within the network, for example, Snoop TCP [Balakrishnan *et al.*, 1996] and RSVP [Braden *et al.*, 1997].

4.1.1 Network State is Soft-State

Many new services will maintain state at nodes internal to the network, for example, custom routing data. This state should be treated as *soft-state* [Clark, 1988], since it can be lost due to partial failures and this should not result in permanent service disruption. To optimize performance, service designers should assume that in the common case information placed in the soft-store will remain there until the coarse grain aging clock associated with the information fires. For the purpose of correctness, however, service designers should assume that the soft-store provides no guarantee that information deposited by one capsule will be accessible by a subsequent capsule, and that if information is lost no indication of failure will be provided. This is because, aside from partial failures, periods of excessive storage demand may cause the node to cycle through the available storage relatively quickly.

A side-effect of this guideline is to avoid high-level conflicts between network-resident processing and the end-to-end argument [Saltzer *et al.*, 1984]. The latter makes the case that functions integral to the correctness of an application must be implemented on an end-system to end-system basis; they can additionally be implemented within the network as a performance optimization. Treating capsule data as soft-state causes network-resident processing to fall into this optimization category. Applications must be prepared to regenerate such data, for example, with periodic refresh messages, or forego the subsequent processing.

A further aspect of soft-state is naming. The soft-store also separates state based on protocols. This means that service designers need not worry about namespace collisions between the state of different services. At a finer granularity, however, the service developer is responsible for separating the state of concurrent sessions by using whatever form of identifier is appropriate. For example, at any given instant, a multicast service might support a number of groups, each of which is an independent session that should not share state with other sessions. This can be accomplished by using the multicast address as a prefix key for soft-store operations. On the other hand, by its nature a Web caching service should be designed to share state between requests.

4.1.2 ALF Techniques Reduce Complexity

One view of the process of designing a service is that some fraction of the processing that would otherwise take place at the application endpoints is moved into the network. To facilitate this structuring, capsule data carried between nodes should be decomposed into units that can be manipulated independently by application code.

This principle, known as Application Level Framing (ALF) [Clark and Tennenhouse, 1990], is applied to achieve efficient performance at end-systems by preserving data boundaries meaningful to the application in the lower-level framing. Effectively, the semantics of the application must be taken into account in the design of its network protocols so that the roles of application and network are matched and the resulting processing is efficient.

The same arguments apply to the processing of application data at intermediate nodes. Here, there is an even greater need to process packets independently and possibly out of order since they cannot reasonably be reassembled on a hop-by-hop basis. By extension of this reasoning, a useful strategy is to make processing at intermediate nodes idempotent, so that duplication does not introduce error.

4.1.3 Design for Asymmetric and Dynamic Paths

Network paths should be modeled as both asymmetric and dynamic so that services do not fail when these situations arise.

A route is asymmetric when the path from destination back to source is not the reverse of the path from source to destination. Asymmetric routes are now common because of "early-exit" routing policies. To accommodate them, service developers should not assume that soft-state placed along a forward path will be encountered along a reverse path.

A route is dynamic in the sense that while it might appear stable it can change at any time. This has consequences for service developers. Any process that constructs routes or measures their characteristics should treat the result as a hint and repeat the process periodically in order to accommodate changes. More basically, no service should have node addresses embedded in its code, since this would limit it to particular topologies. Instead, addresses should be passed to capsules as input parameters.

4.2 Case Study I — Multicast

I describe multicast experiments with ANTS along three lines: a problem statement; the design of the basic service in ANTS; and extensions that exploit the flexibility of an active network.

I draw two conclusions from these experiments. First, despite the complexity of PIM, it is possible to construct a similar service in ANTS. This demonstrates that the node API is sufficient for real-world tasks. Second, it became apparent that there are many variations of the service that produce different and useful effects for clients. This suggests that the active network approach will be valuable because, in contrast with a fixed standard, one variation is as easy to use as another.

4.2.1 Problem and Relevance

The problem addressed by multicast is how to efficiently send one message to multiple receivers, which are traditionally identified by a group address. This service is widely applicable, for example, for conferencing and replicating information, and has significant performance advantages over multiple unicasting.

Early work on multicast was pioneered by Deering [Deering, 1989]. Most of the innovation has been in the routing mechanisms that establish the dissemination trees over which data is forwarded. The first routing protocols, such as DVMRP [Waitzman *et al.*, 1988], were targeted at situations in which the members of a multicast group were densely distributed within a contained region. They worked by pruning the flood of the region. To cater for wide-area groups, new protocols in which distribution trees were grown explicitly, rather than pruned from a flood, were explored [Deering *et al.*, 1994]. This reduced the scope of multicast messages, but resulted in a significant amount of routing state, since a distribution tree was maintained for each combination of group and sender. Routing protocols such as Core-Based Trees (CBT) [Ballardie *et al.*, 1993] reduce this state by sharing a single tree across all senders.

Today, Sparse Mode PIM [Deering *et al.*, 1998] (hereafter referred to as simply "PIM") continues this evolution and is arguably the leading candidate for wide-area multicast routing. PIM is a good test of the ANTS architecture for reasons beyond the choice of a service of widespread utility that is difficult to deploy in the Internet. Multicast has been the subject of much research, and after several years of effort PIM is a relatively mature protocol that is used by a number of ISPs. As such it is also relatively complex. This, combined with the fact that it does not map to the ANTS architecture in an obvious manner, makes it a challenging test case.

I briefly describe PIM for those readers not already familiar with it. To receive PIM multicast messages, hosts explicitly join a group via their Designated Router. This router sends periodic join messages to a rendezvous point for the group, causing a shared distribution tree that is rooted at the rendezvous point to be constructed within the network. Sources send to the group by encapsulating their messages and sending them to the rendezvous point. There they are decapsulated and forwarded down the shared tree. Prune messages can be used to leave a group.

Receivers (and the rendezvous point) can also initiate source-specific trees that are rooted at a given source rather than the rendezvous point, if they are deemed worthwhile. This is typically done to reduce overhead when a source is sending with a high data-rate. Source-specific joins travel towards the indicated source, and construct additional tree information in the network. Within the network, the PIM forwarding algorithm understands how to use both kinds of forwarding state to switch between shared and source-specific trees. To prevent a source from sending messages via the rendezvous point when the shared tree is no longer being used, the rendezvous point sends stop messages to that source. Further features and details are described in [Deering *et al.*, 1998].

Derived from class JoinPruneCapsule:

```
int last;                              // previous PIM node
int holdtime;                          // route data period
int group;                             // group address
boolean ackReq;                        // capsule needs ack
boolean ack;                           // capsule is ack
boolean join;                          // join or prune
int S;                                 // "source" (RP)
boolean WC;                            // wildcard (any source) route
boolean RPT;                           // shared (RP) tree

public boolean evaluate(Node n) {
  Integer I = new Integer(last);         // look up routing data
  Integer G = new Integer(group);
  RouteEntry r = RouteEntry.get(n, new Integer(RouteEntry.SOURCE_ANY), G);

  last = n.getAddress();                 // update incoming address
  if (ackReq) {                          // send an ack if needed
    ackReq = false;
    ack = true;
    setDst(I.intValue());                // back to where we came from
    n.routeForNode(this, I.intValue());  // there is goes
    ack = false;
  } else if (ack) {                      // or process ack data
    if (n.getAddress() == getDst()) {    // have we reached previous?
      if (WC && r != null) {
        r.iif = I;                       // update our route entry
        r.prune(I);
      }
      return true;
    } else
      return n.routeForNode(this, getDst());
  }

  if (join && WC && RPT) {               // a join for the shared tree
    if (r == null) {                     // create a fresh entry
      r = RouteEntry.New(n, new Integer(RouteEntry.SOURCE_ANY), G,
                         new Integer(S), holdtime);
      if (n.getAddress() != S) {         // initialize and continue
        ackReq = true;
        setDst(S);
        n.routeForNode(this, S);
      }
    } else if (n.getAddress() != S && (r.oif.isEmpty() ||
                n.time() - r.lastPeriodic > JOIN_PRUNE_PERIOD)) {
      setDst(S);                        // send upstream join  periodically
      n.routeForNode(this, S);
    }

    long timeout = holdtime*1000 + n.time();
    r.join(I, timeout);                  // update our route entry
  } else
     [ omitted ] // other cases not covered here!
  return true;
}
```

Figure 4-1: Pseudo-code for joining the shared tree

4.2.2 Service Design

One of the first observations to make is that there is a mismatch between the assumptions implicit in the design of PIM and ANTS. PIM assumes that all routers in a region will be multicast-capable, while ANTS allows for networks in which only a subset of the routers can be re-programmed. PIM also uses periodic timers to maintain routing information, whereas ANTS does not provide a timer facility for use within the network.

To design an ANTS multicast service, I focussed on resolving these issues to implement the core PIM mechanism specified in [Deering *et al.*, 1998]: the formation of shared and shortest-path trees; cutover between them; and forwarding along them. Other aspects of PIM (namely extensions to multi-access media, the mapping of groups to rendezvous points for bootstrapping, and interoperation with other multicast routing protocols or administrative domains) were omitted. While these aspects are necessary for a complete implementation, they are mostly separable mechanisms that do not contribute significantly to understanding whether ANTS can successfully support multicast routing. Further, I made no effort to allow the ANTS multicast service to interoperate with existing PIM implementations by using identical packet formats. This is because the question being studied is whether a general-purpose system such as ANTS could have obviated the need for a special-case multicast implementation in the first place. For convenience, however, IP multicast addresses are used to identify ANTS multicast groups.

The resulting service is comprised of three co-operating capsule types. Each of these roughly corresponds to a PIM message, and each is processed by nodes according to the PIM specification. All three capsules are grouped into a single code group because of the interactions that PIM requires within the network. The role of these capsules is summarized below, then each is described in detail.

- `JoinPrune` capsules are sent by hosts periodically (and by routers when triggered) to maintain tree forwarding state at active nodes within the network.
- `Data` capsules are sent by a source to deliver information to the group.
- `RegisterStop` capsules are sent to a data source by the rendezvous point (the root of shared tree) to signal the source that the rendezvous point has joined the shortest path tree.

To describe the network processing of each of these capsules more fully, consider the following scenario. The focus of the discussion is on design considerations and the differences between an ANTS-style PIM and regular PIM, rather than trying to explain PIM itself.

Before a node can receive data that is multicast to a group, it must join the group. This is one of the tasks accomplished by the `JoinPrune` capsule. Capsules of this type are sent periodically by each receiver to the rendezvous point for the group, setting up reverse path forwarding state for the shared tree in active nodes along the way. Pseudo-code for this kind of `JoinPrune` processing is shown in Figure 4-1. (The complete set of `JoinPrune` processing is more involved because it includes other cases, which are omitted here for simplicity. See the ANTS distribution for details.) The path of these capsules is shown in Figure 4-2. Large arrows over links represent the path of capsules, and small arrows near nodes represent forwarding state.

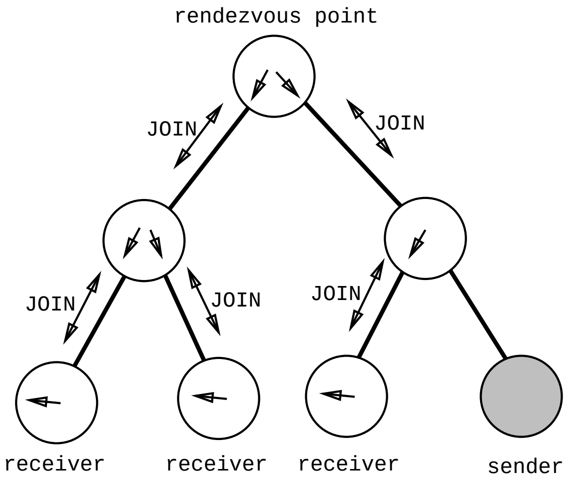

Figure 4-2: Path of `JoinPrune` (marked as join) capsules as receivers join the shared tree.

ANTS `JoinPrune` implements all of the major PIM features: shared trees based on the rendezvous point and shortest-path trees for specific sources. It differs from its PIM counterpart in three respects. The first is a consequence of a separation of function that is not present in the toolkit. By Internet convention, routers participate in route construction while hosts only forward packets. Therefore, to handle group membership, routers act as agents for their local hosts and communicate with them using the Internet Group Management Protocol (IGMP) [Fenner, 1997]. In ANTS, each host manages its own group membership operations directly. In PIM parlance it is its own Designated Router.

The strategy for reducing the overhead of `JoinPrune` messages also differs. In PIM, periodic timers are used to exchange aggregated group membership information between neighboring routers. In ANTS, no built-in timer facilities are available. Instead, `JoinPrune` capsules are periodically sent by hosts for each group to which they belong. As these capsules pass through each active node they are filtered so that only one capsule per interval is sent over a given link for a given group. This is readily accomplished by maintaining information in the soft-store that is checked before the capsule is forwarded; capsules above the specified rate are simply discarded.

A third difference is that the reverse path forwarding state kept at nodes has a different form. In PIM, forwarding state is expressed in terms of input and output interfaces. This is not sufficient in ANTS because there is no guarantee that the next node encountered on an outgoing interface will be an active node. To accommodate this, forwarding state is expressed in terms of node addresses. As the `JoinPrune` capsules are routed towards the rendezvous point, they record the address of the last encountered active node in a header field. Acknowledgements carry address data in the reverse direction to complete the link. The address data is then recorded in the soft-store of the active nodes, allowing capsules to be sent through the non-active nodes in-between. In effect, a tunnel that is analogous to those used in the MBONE is established dynamically, rather than being configured manually. (This "dynamic tunnel" method might be applied to improve PIM by restricting multicast forwarding state to those nodes where messages must be replicated. A similar mechanism is explored in [Tian and Neufeld, 1998].)

Derived from class MulticastDataCapsule:

```
int group;                          // group address
boolean register;                   // "encapsulated" register

public boolean evaluate(Node n) {
  Integer G = new Integer(group);
  Integer S = new Integer(getSrc()); // look up routing data
  Integer I = new Integer(last);
  RouteEntry r = RouteEntry.get(n, new Integer(RouteEntry.SOURCE_ANY), G);

  if (getSrc() == n.getAddress()) {  // initialize at first node
    if (r == null) {                 // make new route entry
      r = RouteEntry.New(n, S, G, new Integer(getDst()), DATA_TIMEOUT);
      r.regSuppTimeout = 0;          // without suppression
    } else
      r.refresh(n, DATA_TIMEOUT);    // kick this soft-state

    if (r.regSuppTimeout < n.time()) {
      register = true;               // unicast to the RP
      n.routeForNode(this,getDst());
    } else {
      [ omitted ] // send is on source-specific trees
    }
  } else if (register) {             // forward a register towards RP
    if (n.getAddress() != getDst())
      return n.routeForNode(this, getDst());
    else {                           // turn it round at RP
      register = false;
      if (r == null) {               // if no route data, send stop
        RegisterStopCapsule c = new RegisterStopCapsule(group, getSrc());
        c.prime(this);
        n.routeForNode(c, getSrc());
      } else {                       // otherwise, forward, check rate
        forwardPacket(this, r.oif, n);
        if (updateThruput(S,G,RP_THRESHOLD_COUNT,RP_THRESHOLD_TIMEOUT,n)) {
          [ omitted ] // initiate source-specific routing
        }
      }
    }
  } else {                           // forward a native packet
    if (r != null && r.iif != null && r.iif.equals(I)) {
      if (forwardPacket(this, r.oif, n)) {
        if (updateThruput(S,G,RP_THRESHOLD_COUNT,RP_THRESHOLD_TIMEOUT,n)) {
          [ omitted ] // initiate source-specific routing
        }
      }
    }
  }
  return true;
}
```

Figure 4-3: Pseudo-code for sending data on the shared tree

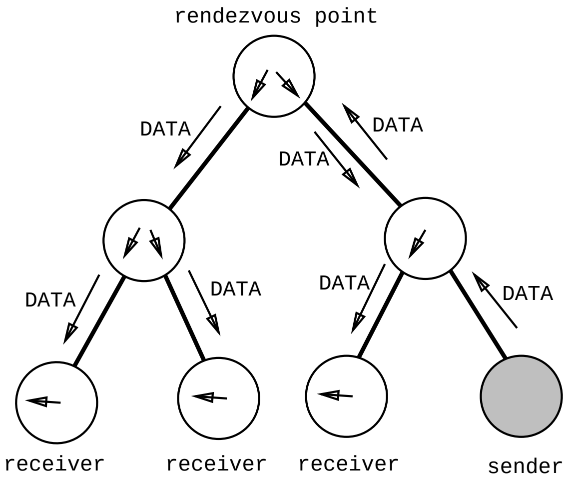

Figure 4-4: Path of `Data` capsules after the receivers join the shared tree.

Once the `JoinPrune` capsules have propagated through the network, the corresponding group members are in a position to receive multicast data. This is sent with the `Data` capsule, which routes itself to follow the established forwarding state. Senders can inject `Data` capsules into the network at any time and regardless of whether they have joined the group. Pseudo-code for `Data` capsule processing, again only for the shared tree case, is shown in Figure 4-3. The path of a `Data` capsule is shown in Figure 4-4.

The full version of the ANTS code implements the major PIM features: shared and source-specific path forwarding, cutover between the different types of path forwarding with triggered route updates, and Register messages and their suppression. It too differs from PIM multicast data in several respects.

In PIM, when a source multicasts data that is to be delivered via the shared tree, forwarding proceeds in two steps. The multicast data is first encapsulated at the local router in a Register message that is sent directly to the rendezvous point. There it is decapsulated to reveal the group address and sent down the distribution tree using forwarding state. The distinction between encapsulated and native multicast data is useful in the Internet to shield hosts from routing messages. In ANTS, this distinction is unnecessary; recall that IGMP is not used. ANTS is therefore able to combine the roles of the PIM Register message and regular multicast data packets without an encapsulation and decapsulation stage.

A second difference is the implementation of the rendezvous point itself. In PIM, it exists as an endpoint within the network that acts as an independent entity to the extent that it originates messages as well as receives them. The ANTS model does not allow the creation of a long-lived process at a remote node. Instead, the required activity is generated by the multicast data stream itself. For example, in PIM the rendezvous point joins the source-specific tree for sources that are sending data at a high rate. This avoids decapsulation and shortcuts paths that would needlessly loop through the rendezvous point. In ANTS, this join is triggered directly by the forwarding of `Data` capsules that exceed a rate limit. The rate information is maintained in the soft-store and updated as `Data` capsules are forwarded. As an added bonus, the same mechanism is used at receivers to control when they join the shortest-path tree.

Derived from class RegisterStopCapsule:

```
int group;                                  // group address

public boolean evaluate(Node n) {
  if (n.getAddress() != getDst())           // continue to destination
    return n.routeForNode(this, getDst());

  Object[] key = {new Integer(ROUTE_ENTRY),   // look up route entry
                  new Integer(RouteEntry.SOURCE_ANY),
                  new Integer(group)};
  RouteEntry r = (RouteEntry)n.getCache().get(key);

  if (r != null && r.timeout > n.time())      // kick suppression timer
    r.regSuppTimeout = n.time() + REGISTER_SUPP_TIMEOUT * 1000;

  return true;
}
```

Figure 4-5: Pseudo-code for processing `RegisterStop` capsules

The final type of capsule is `RegisterStop`. Pseudo-code for the processing of this capsule is shown in Figure 4-5. It is used after the rendezvous point joins the shortest path tree (which is not pictured). Once the rendezvous point begins receiving multicast data via a shortest path tree, it must periodically notify the corresponding local router. In PIM, this kicks a timer that is used to prevent multicast data from being encapsulated in a Register message. In ANTS, this message is generated from within multicast data forwarding, as described above. To approximate a periodic message, information is placed in the soft-store at the rendezvous point for aging. This data is checked during forwarding via the rendezvous point, and when it is no longer present, a `RegisterStop` capsule is sent to the source. The path of this message is shown in Figure 4-6.

4.2.3 Possible Extensions

One of the strengths of an active network is that services can be readily refined as application needs evolve. Reliable multicast protocols, such as SRM [Floyd *et al.*, 1997] and LBRM [Holbrook *et al.*, 1995], provide a topical example of this process in action. It has proven difficult to scale the size of the multicast group with the existing forwarding model, which sends a copy of a multicast message to all recipients. A significant number of proposals are now exploring network support for reliable multicast, for example, [Papadopoulos *et al.*, 1998], [Li and Cheriton, 1998] and [Levine and Garcia-Luna-Aceves, 1997].

In this section, I consider two hypothetical extensions that are intended to convey some of the directions in which an ANTS multicast service could be readily expanded. Lehman has also explored a variety of reliable multicast protocols in the context of ANTS [Lehman *et al.*, 1998].

One extension allows a receiver to send a message to other receivers in its distribution subtree. This mechanism is referred to as subcasting, and provides a basis for local recovery schemes. The other discards duplicate messages traveling up the distribution tree. This

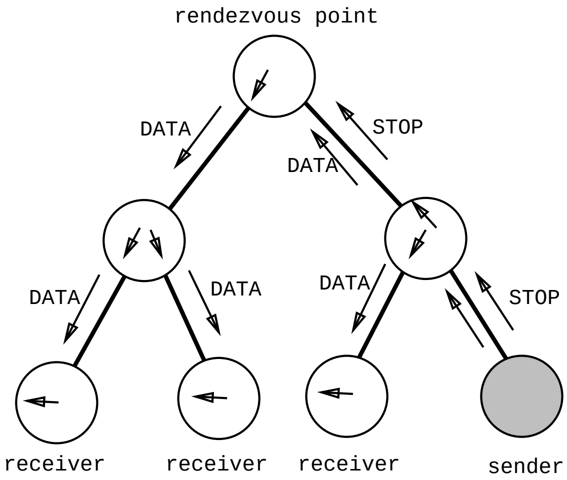

Figure 4-6: Path of `RegisterStop` (marked as stop) and `Data` capsules after the rendezvous point joins the sender-specific tree.

mechanism is complementary to multicast forwarding, and provides a basis for preventing NACK implosion (the overload that occurs when many receivers simultaneously notify one sender of loss) without introducing delay. It is referred to as suppression. For simplicity, both mechanisms are considered only in the context of a single shared distribution tree rooted at the data source.

Subcasting

The subcasting primitive first proceeds up the distribution tree, towards the source, a specified number of hops, and then forwards data down that portion of the overall tree.

The path of a `Subcast` capsule is shown in Figure 4-7. Before the capsule is turned, it is forwarded towards the source and counts the number of multicast nodes along the way. Default routing can be used for this purpose because the distribution tree is built in terms of reverse shortest paths. When the capsule has passed the specified number of multicast nodes (indicated when the turn count reaches zero), a `Data` capsule is created to deliver the data to the subtree routed at the current multicast node.

This mechanism requires no additional soft-state and does not slow regular multicast forwarding.

Suppression

The suppression primitive sends a capsule up the distribution tree, towards the source, discarding duplicate messages. Ideally the source will receive only one copy of a message even if it is sent by all group members.

The path of `Suppress` capsules is shown in Figure 4-8. The forwarding routine uses an identifier, such as the multicast sequence number in the case of NACK suppression, to

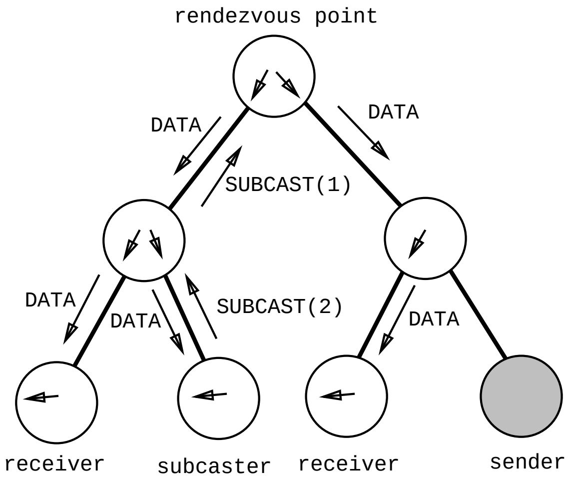

Figure 4-7: Path of a two level `Subcast` capsule and subsequent `Data` capsules.

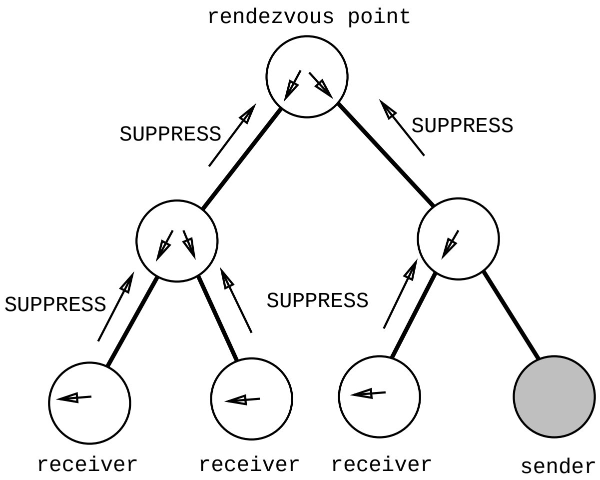

Figure 4-8: Path of `Suppress` capsules sent from all receivers.

match and discard duplicate messages. At every node, information in the soft-store is consulted to determine if the capsule is the first of its kind to have passed through this node. If so it is forwarded up the distribution tree. As before, default routing via shortest paths can be used directly. If it is not the first capsule of its kind, then it is simply discarded as a duplicate. An alternative might be to pass only some fraction of the duplicates, or only one duplicate per interval; both parameters could be specified as part of the `Suppress` capsule if they vary per capsule.

This service makes limited use of the soft-store and does not slow regular multicast forwarding.

4.3 Case Study II — Web Caching

As an example of how active network capabilities can be applied today, I consider the problem of Web caching that was raised in the introduction. As for the first study, the discussion is divided along three lines: a problem statement, the design of the basic service in ANTS, and extensions that exploit the flexibility of an active network.

I draw two conclusions from these experiments. First, ANTS is well-suited to draw embedded network resources, such as Web caches, into the chain of processing between client and server. It provides far greater flexibility than the configuration of proxies or use of firewall-style transparency. Second, there are many variations of the service that produce useful effects by using embedded network resources in different ways. Again, this suggests that the active network approach will be valuable in practice.

4.3.1 Problem and Relevance

With the dramatic increase in Web traffic, caching mechanisms are being explored to boost performance. They do this in three respects: the latency of client requests is reduced; the load on servers is reduced; and the use of wide-area bandwidth is reduced. Early efforts improved client behavior to avoid redundant transfers (client-side caching) and redirected queries through a local agent to improve sharing (proxy caching).

To improve caching further, it is necessary to share information between caches at different locations in the network. The key problems are: how to locate a copy of a document by searching through the caches, and how to propagate documents to caches for future queries. Both can be thought of in terms of routing Web requests and responses through a shared network of caches. Since performance is related to the underlying network topology, it is natural to consider combining cache lookup with packet routing. This approach is being explored by Legedza [Legedza and Guttag, 1998]. It contrasts with existing systems that construct an overlay of cache nodes. It has the potential to improve performance by stripping away the overhead of overlays and retaining packets in the network layer. This simple strategy may also result in globally cost-effective cache search paths in the same sense that multicast routes based on shortest paths (like PIM) are known to be effective in practice [Doar and Leslie, 1993].

The problem that I tackle here is how to control the overall cache lookup and replacement path with a new network service that combines routing and caching. Besides being topical, this problem is interesting because it is not well-suited to a pure ANTS service. This is because cache nodes require a large amount of storage (typically provided by disks) and the ability to support reliable connections, neither or which fit the notion of an active node. Instead, I assume that cache nodes will be incorporated into the network infrastructure at selected points, such as at access links, ISPs, and exchange points. This is already happening today. The problem that must be tackled, then, is to design a service that takes advantage of these caches as part of a new network service.

4.3.2 Service Design

To provide a caching service, two key pieces of functionality must be designed. There must be some way for Web requests to locate objects in caches that are a short distance from the path they follow towards the server. In the reverse direction, there must be a means for Web responses to transfer objects to these same caches so that they are available to satisfy future requests.

The design of these mechanisms should meet two goals. First, the impact on network performance must be minimal. This can be achieved by specializing network processing only for the request and response packets that establish the data transfer connections but not for the data transfer itself, that is, only for the signaling packets. The data transfer itself then proceeds at full speed, without additional processing inside the network. This strategy contrasts with existing products, in which backwards-compatibility requires that all packets be processed (to spoof either end of the connection) at each modified router. Second, the impact on the reliability of the Web must be minimal. Failed caches and active nodes should not result in persistent failures to retrieve a Web object. This can be achieved by using default routing (which will route around failed nodes) and having cache/active node pairs act in concert so that it appears that either both fail or none do.

In ANTS, the resulting service is realized with four co-operating capsule types, each of which forms its own code group:

- `Bind` capsules are exchanged between the cache and its associated active node to synchronize their states.
- `Query` capsules are sent from clients towards the server. Along the way they may encounter caches that have a copy of the requested document, in which case they will be diverted.
- `Redirect` capsules initiate a connection between the server (or an upstream cache) and a downstream cache (or client) for the purpose of transferring a document and redirecting subsequent requests.
- `Activate` capsules terminate a connection between a downstream cache (or client) and server (or an upstream cache) once the transfer of a document has been completed and redirection state has been established.

In addition, regular packets, which do not require special network handling, are exchanged between the downstream cache and client to transfer the Web document itself.

To describe the network processing of each of these capsules more fully, consider the following scenario for document retrieval. It is based on the organization of caching functionality within the network into caches and active nodes. Each cache node is associated with active nodes in the region of the network that it serves. Each corresponding active node maintains a rough table of cache contents in its soft-store. It uses this table to decide whether to divert Web requests to the cache node from the shortest path that they would otherwise follow.

From class BindCapsule:

```
public boolean bound;           // latched the active node
public long epoch;              // web cache generation
public int interval;            // binding timeout

public boolean evaluate(Node n) {
  // continue until we reach the active node or the cache
  if (n.getAddress() != getDst()) return n.routeForNode(this, getDst());

  if (!bound) {
    // bind node to cache by entering its details in the soft-store
    Object[] binding = new Object[2];
    binding[0] = new Long(epoch);
    binding[1] = new Integer(getSrc());
    n.getCache().put("CACHE", binding, 3*interval);

    // now alter capsule and return to cache for synchronization
    setDst(getSrc());
    bound = true;
    if (n.getAddress() != getDst()) return n.routeForNode(this, getDst());
  }

  // we are back at the cache
  return n.deliverToApp(this, dpt);
}
```

Figure 4-9: Pseudo-code for processing `Bind` capsules

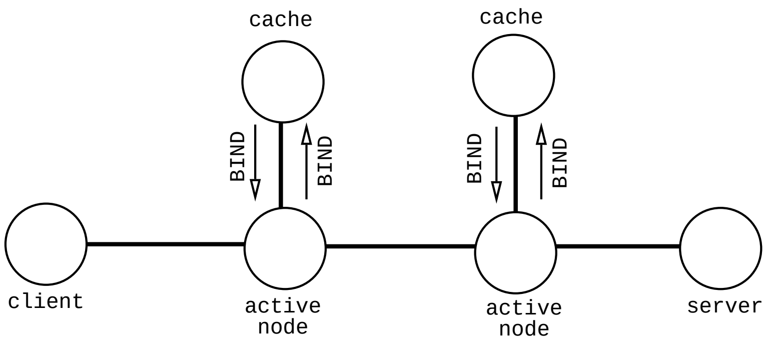

Figure 4-10: Path of `Bind` capsules.

From class QueryCapsule:

```
public int last;                              // previous cache
public int found;                             // cache with hit
public long objectID;                         // requested object

public boolean evaluate(Node n) {
  if (found != 0) {                           // divert to a found cache
    if (n.getAddress() == found) {
      return n.deliverToApp(this, dpt);
    } else {
      return n.routeForNode(this, found);
    }
  }

  // otherwise, are we at a cache -- is there a binding?
  Object[] record = (Object[])n.getCache().get("CACHE");
  if (record != null) {

    // if a cache, is there a redirect record?
    Object[] redirect = (Object[])n.getCache().get(objectID);
    if (redirect != null) {

      // if a redirect too, does it match the binding?
      Long bindEpoch = (Long)record[0];
      Long redirectEpoch = (Long)redirect[0];
      if (bindEpoch.longValue() == redirectEpoch.longValue()) {
        found = ((Integer)record[1]).intValue();
        if (n.getAddress() == found) {
          return n.deliverToApp(this, dpt);
        } else {
          return n.routeForNode(this, found);
        }
      } else {
         n.getCache().remove(objectID);       // clear out old data
      }
    }
    last = n.getAddress();                    // note cache was here
  }

  if (n.getAddress() == getDst()) {           // now continue to server
    return n.deliverToApp(this, dpt);
  } else {
    return n.routeForNode(this, getDst());
  }
}
```

Figure 4-11: Pseudo-code for processing `Query` capsules

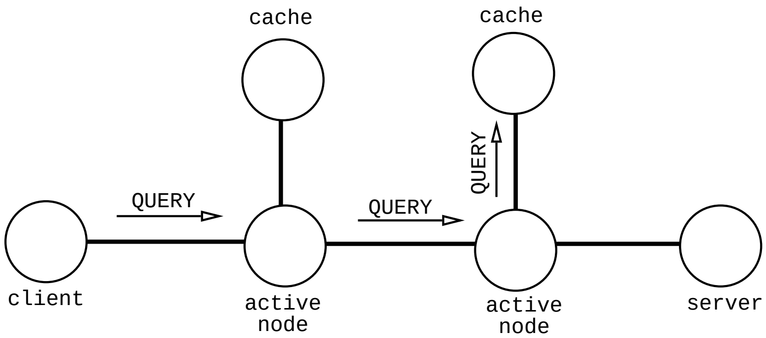

Figure 4-12: Path of a Query capsule that hits in a network cache.

Before a cache can be used by the ANTS Web caching service, it must establish contact with nearby (and possibly co-located) active nodes. This is accomplished with the Bind capsule. Its purpose is to bind the cache and active node into a single unit of operation. In the prototype ANTS service, one associated cache is statically configured, though it would be straightforward to extend this to multiple active nodes and automatic configuration via limited broadcast.

The path of the Bind capsule is shown in Figure 4-10. Pseudo-code for the its processing is shown in Figure 4-9. The capsule is sent periodically by the cache to the active node and then returned to the cache. Binding of cache to active node is accomplished by checking for the return of the Bind capsules. If several consecutive capsules are not received, then the cache can assume that the active node has failed. Binding of the active node to the cache is accomplished by placing the address of the cache in the soft-store with a short expiration interval. If the cache fails, then this state will be removed by the node, and the service code will discover that caching is not currently available. To ensure that the cache and active node can maintain consistent state when the other fails and restarts, the cache passes an epoch identifier (defined to be the time of the most recent restart) with the Bind capsule.

Once the caches are up and running, they are used to satisfy Web requests by Query capsules. These capsules are sent by the client to open a connection to receive the requested document, that is, they add processing to a TCP-SYN packet. The path of a Query capsule that hits in a cache is shown in Figure 4-12. Pseudo-code for its processing is shown in Figure 4-11. It heads towards the server by default, and checks along the way for active nodes with associated caches, as indicated by information in the soft-store. There are three outcomes of this check. If the node has no associated cache, the Query continues towards the server. If there is an associated cache but the requested document does not hit in the table of contents at the active node, then the Query continues towards the server, noting the address of the active node for subsequent cache loads. Finally, if there is a cache hit in the table of contents, then the Query is diverted to the cache. The hit test includes a check that the active node and cache belong to the same epoch. If not, then old data has been found, and it is removed. Note that this process flushes the active node table of contents gradually when the cache is restarted, rather than requiring any one capsule to perform a large amount of work. In all of these cases, the Query capsule eventually leaves the network when it reaches either the server of hits in a cache.

The cache or server that receives the Query must load downstream caches (if any have

From class RedirectCapsule:

```
public long objectID;                           // requested object
public int redirect;                            // redirecting node
public int requestor;                           // original client

public boolean evaluate(Node n) {

  // have we reached the redirecting node?
  if ((redirect == 0) &&  (n.getAddress() == getDst())) {
    Object[] record = (Object[])n.getCache().get("CACHE");

    redirect = n.getAddress();                  // if so, lock
    if (record != null) {
      setDst(((Integer)record[1]).intValue());  // and head to external cache
    } else {
      setDst(getSrc());                         // failure, return to server?
    }
  }

  // continue to redirect, external cache, or server
  if (n.getAddress() != getDst()) {
    return n.routeForNode(this, getDst());
  } else {
    return n.deliverToApp(this, dpt);
  }
}
```

Figure 4-13: Pseudo-code for processing `Redirect` capsules

been encountered) as well as see that a copy of the document is returned to the client. These tasks are accomplished as a series of cache transfers, culminating in the last cache to client transfer. Each transfer is accomplished by augmenting the signaling of a regular data transfer (a TCP connection) with the `Redirect` and `Activate` capsules. Pseudo-code for their processing is shown in Figures 4-13 and 4-14. The path of these capsules, responding to a request hit within the network, is shown in two parts in Figures 4-15 and 4-16. The first shows a cache to cache transfer, and the second the final cache to client transfer. In a high performance implementation, these transfers would be overlapped to reduce latency.

The `Redirect` capsule is used to establish a connection between caches, or between the last cache and the client, for the transfer of a document. For cache-to-cache transfers, it is designed to augment the role of a TCP-SYN packet, which initiates a connection. For the final cache-to-client transfer, it is designed to augment the role of a TCP-SYN-ACK packet, which accepts a connection initiated by the corresponding `Query` acting in the role of a TCP-SYN[1]. Unlike a regular packet, it is routed via the associated active nodes that were recorded in the `Query`. This allows those nodes to observe the cache hit and ensures that the cache pairings are still valid. Data transfer then proceeds as normal, directly between

[1]Observant readers will notice that this requires that the client be capable of accepting a connection from a third party. This is analogous to TCP connections to an anycast address. See [Partridge *et al.*, 1993] for details.

From class ActivateCapsule:

```
public long objectID;                       // requested object
public long epoch;                          // cache generation
public int redirect;                        // cache to redirect
public int activate;                        // cache we activated

public boolean evaluate(Node n) {

  // have we reached the node to activate?
  if ((activate == 0) &&  (n.getAddress() == getDst())) {
    Object[] record = (Object[])n.getCache().get("CACHE");

    activate = n.getAddress();              // if so, lock
    setDst(redirect);

    if (record != null) {
      long bound = ((Long)record[0]).longValue();
      if (bound == epoch) {
        Object[] redirect = new Object[2];  // put it in the cache
        redirect[0] = (Long)record[0];
        n.getCache().put(objectID, redirect, 100);
      } else {
        return true;                        // unexpected change, die quietly
      }
    }
  }

  // continue to node or upstream cache/server
  if (n.getAddress() != getDst()) {
    return n.routeForNode(this, getDst());
  } else {
    return n.deliverToApp(this, dpt);
  }
}
```

Figure 4-14: Pseudo-code for processing `Activate` capsules

the sender and receiver without special network handling. Similarly, the `Activate` capsule is used to terminate these connections after they have been used for data transfer. It is designed to augment the role of a TCP-FIN packet. Like the `Redirect` capsule, it is routed via the corresponding active nodes to allow them to observe the cache load and update their table of contents.

4.3.3 Possible Extensions

While this basic service has a number of attractive properties, a key strength of using an active network is that variations can be constructed to make use of the same caching infrastructure in different ways. Here, I consider two hypothetical variations that employ different search and replacement policies that are appropriate to different kinds of applications. The first variation extends the capabilities of clients, the second of servers. Many

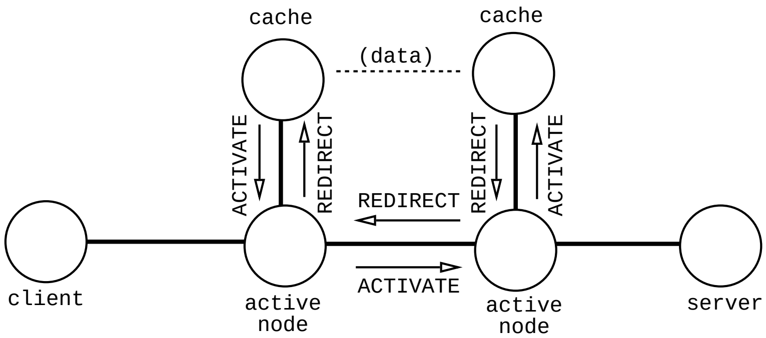

Figure 4-15: Path of capsules that load a downstream cache.

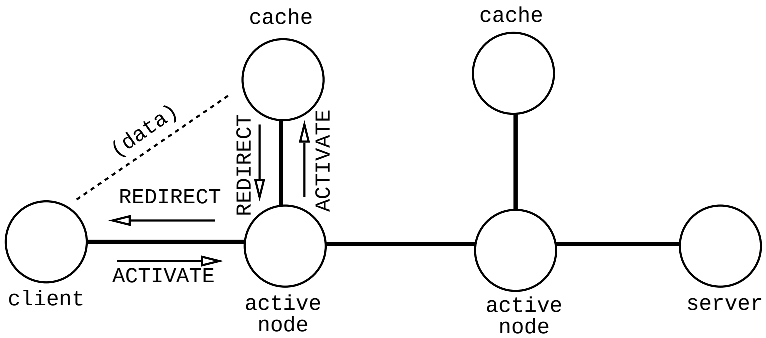

Figure 4-16: Path of capsules that return a document to the client.

other variations are possible.

Caching of Timely Data

Today's Web caches do not store dynamically generated data because the response may differ each time. The situation is slightly different for continuously varying quantities, such as stock quotes. This data is dynamic in the sense that it changes over time, but for many applications it would be sufficient to observe it with a specified granularity. If stock quotes (or other requests that exhibit these properties) become popular, then the ability to cache them within the network at all may provide a significant benefit to heavily loaded servers.

It is straightforward to take advantage of timely data by constructing a service that searches for cached objects of a specified currency. This requires that the cache table of contents maintained at active nodes include a modification timestamp that can be used to assess freshness. Then, a new request for timely data, a `TimedQuery` would include both an object name and a currency. It would invoke a modified check at active nodes for both availability and currency, but otherwise proceed as previously.

Popular versus Unpopular Objects

Since maximum Web cache hit rates are typically no better that 50%, it is important to make informed decisions about whether to cache a document to make effective use of cache space. Bhattacharjee explores one approach: placing documents in every nth cache and making n small caches appear to be a single large cache [Bhattacharjee *et al.*, 1998]. A different approach is to try to and assess the popularity of data and discard that not worth caching.

In a solution deployed pointwise, each cache must make its own decisions as to whether to cache a document based on the request and response sequences it observes. With a new caching service, additional information provided by the server or upstream caches — which have observed previous aggregated request sequences — can assist in this decision. For example, caches might only store objects that are marked by the server as popular.

It is straightforward to introduce this distinction into the caching mechanism. If a server simply classifies objects as popular and unpopular, then popular requests are handled as before. Unpopular requests are handled by having the server contact the client directly. No new service is needed. A slightly more sophisticated version might involve the server marking responses with the last retrieval time of the object. Since caches and active nodes will have varying storage resources (depending where they are in the network) the decision as to whether a document is "popular enough" to store may depend on both the object and the cache. A new service could propagate responses back through caches to the client, while only storing an object if the last retrieval time is hot enough for the cache.

Chapter 5

Evaluating ANTS

In this chapter, I present an evaluation of the ANTS architecture that is intended to shed light on the hypothesis of this thesis: that an active network can be used to rapidly and automatically introduce new services at a reasonable cost in terms of performance and security. I find the hypothesis to be valid. ANTS is capable of introducing a variety of new services. The architecture is able to support these new services efficiently, with modest overheads compared to IP. And the security afforded by the architecture is comparable to that of the Internet.

The basis for this evaluation is experience gained by constructing new services, such as those presented in Chapters 3 and 4, and experiments that characterize these new services running under the ANTS toolkit. The focus of this evaluation is on the tradeoffs between flexibility, security and performance. I use the Internet as a baseline for comparison. I describe ANTS performance first, then its security properties and finally the expressiveness of the capsule programming model. Three kinds of performance are discussed: local node forwarding performance, code distribution performance, and overall application performance.

5.1 Node Performance

In this section, I characterize node performance both quantitatively and qualitatively. I focus on forwarding performance for the common case in which code has already been loaded. The subsequent section examines code distribution performance.

Measurements of the active node report on a Java-based implementation running on a Unix-based operating system and commodity hardware. They show that the prototype is able to run at rates in excess of 1700 capsules/second for small packets and 16 Mbps for large packets, and is largely limited by Java. An analysis of capsule forwarding compares the intrinsic runtime overhead of the ANTS architecture to that of IP. It shows that ANTS requires very few additional processing steps.

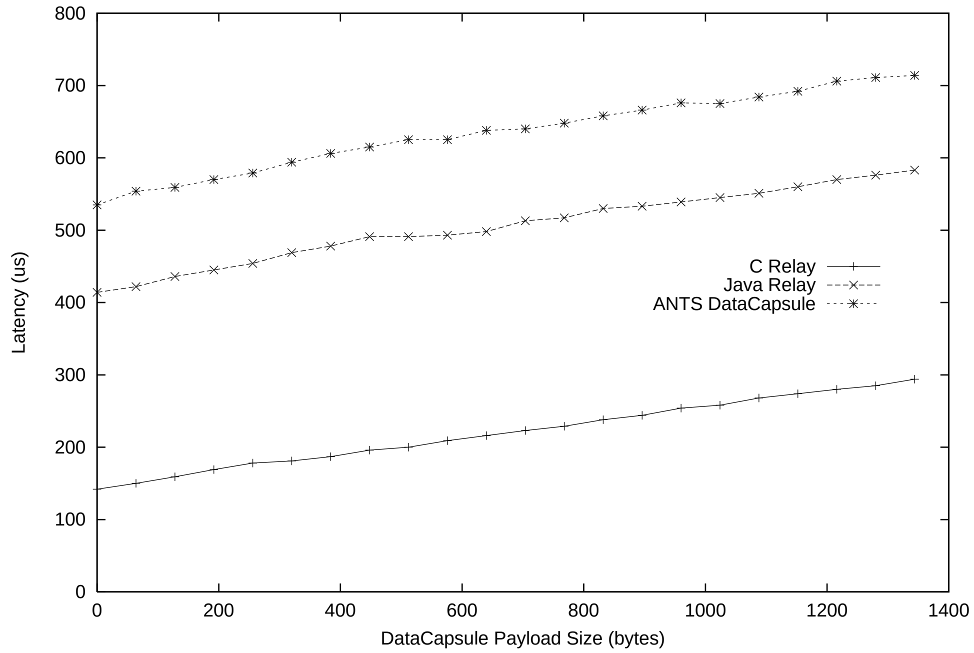

Figure 5-1: Median Latency of an ANTS node as a function of Capsule Payload Size

5.1.1 Experimental Setup

All results reported in this section were measured on a Sun Ultrasparc 1 (167 MHz) platform running Solaris 2.6, a commercial version of Unix, and connected to other ANTS nodes by a switched 100 Mbps Ethernet. The Java runtime used was an early access release of Sun's upcoming JDK 1.2. This runtime includes a just-in-time compiler.

For each experiment, streams of capsules of type `DataCapsule` were sent through the active node. This capsule implements UDP-like functionality within the ANTS framework. It is intended to be the equivalent of the "null RPC" that is often used to assess the costs of an implementation. Latency was measured across the complete active node by using a passive `tcpdump`-based monitor that records processor cycle counters on packet arrival within a modified Linux kernel. Throughput was measured with a test harness that gradually increased the load while monitoring the output of the node to determine when it had saturated.

To place the ANTS results in context, the performance of a Java relay and C relay running on the same hardware and operating system were measured. Both relays are short programs (of less than 100 lines) that simply receive and transmit packets, uninterpreted and with no additional processing. The Java relay shares the same basic receive and transmit loops as the ANTS node. It is used to gauge the combined performance of the Java runtime and measurement platform. The C relay is a straightforward socket-based program that runs at user-level. It is used to gauge the performance of the measurement platform.

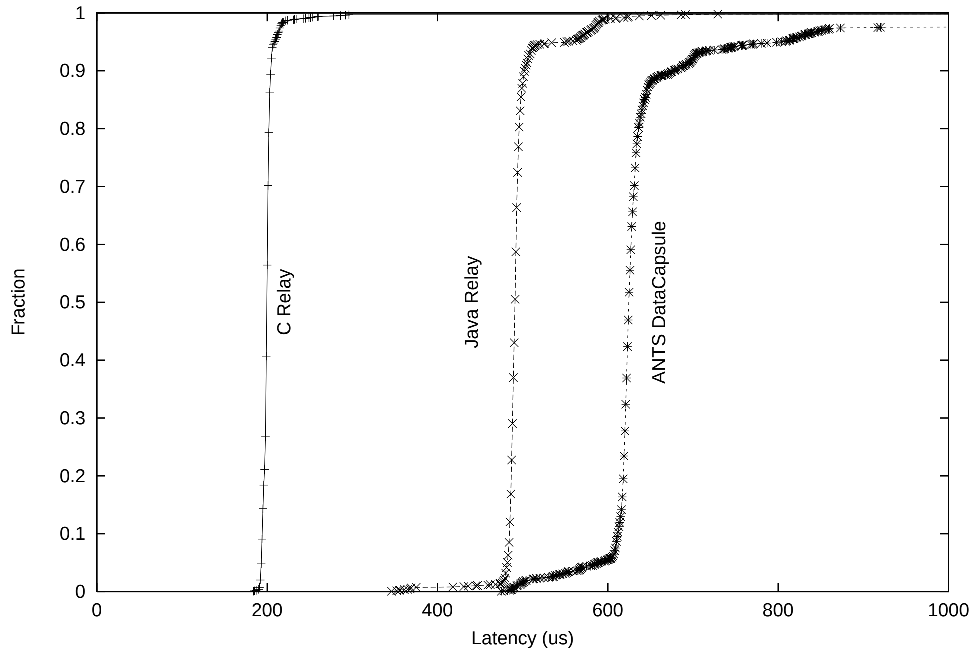

Figure 5-2: Cumulative PDF of the Latency of an ANTS node for 512 byte DataCapsule payloads.

5.1.2 Latency

The median latency of an active node forwarding capsules is shown as a function of the `DataCapsule` payload size in Figure 5-1. The three lines on the graph correspond, from bottom, to the C relay, the Java relay, and the ANTS node. It is evident that the Java relay adds a constant overhead of around 250us per capsule, while the ANTS node adds closer to 400us per capsule. All lines increase from left to right, as data transfer costs increase with the size of the capsule. Importantly, all lines have the same slope. This confirms that the data-dependent costs of the active node are of the same form as the minimal costs for the platform. No hidden costs that are a function of the packet length have emerged in the implementation, and the ANTS node adds a modest overhead to the Java relay.

To examine the latency behavior more closely for a single point on the first graph, Figure 5-2 shows the cumulative distribution of latency measurements for the three systems when the DataCapsule payload is 512 bytes. Again, all three distributions have a similar form, where the majority of capsules are processed in essentially a constant amount of time. Because the measured platform is based on a general purpose operating system, the distributions all have long tails, with some capsules taking up to 100ms to process when other system activity interrupts forwarding. The C Relay, Java Relay, and ANTS distributions end with an increasingly gradual step because the longer and more involved the forwarding routine, the more likely it is to be affected by system activity. The softening of the beginning of the step for the Java Relay and ANTS distributions has not been attributed to a particular aspect of the implementation.

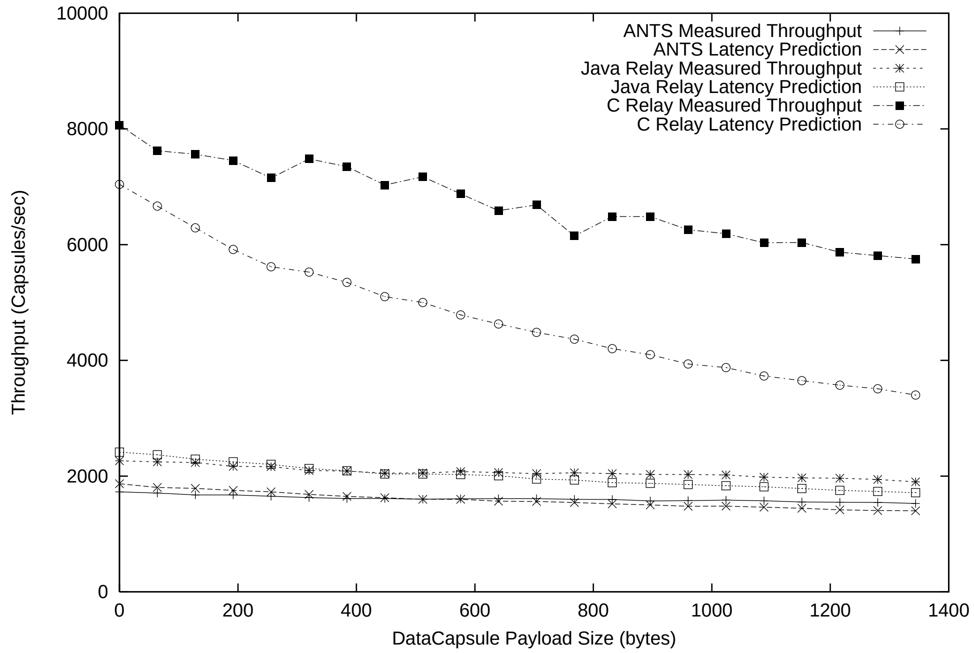

Figure 5-3: Throughput of an ANTS node as a function of Capsule Size

5.1.3 Throughput

The throughput of the ANTS node, along with the Java and C relays is shown in Figure 5-3 in capsules/sec and in Figure 5-4 in Mbps. For comparison purposes, the latency data is also shown, inverted to convert it into a rate measure. At close to 1600 capsules/sec and 6.5 Mbps for 512 byte payloads, the ANTS system is usable for experimenting with distributed applications at various points in the network, for example, across wireless links, modems, and access links through T1 (1.5 Mbps) speeds. Importantly, it captures most of the potential of a Java process running on the platform, indicating that the ANTS processing is not having a serious adverse effect on performance. It is clear, however, that the system falls well short of the performance potential of the platform, which is 3 to 4 times better, though this gap should close as the implementation of Java runtimes and compilers improves.

The graphs show that the node throughput is essentially computationally rather than memory bandwidth limited. The difference between the measured throughput and that predicted from latency arises because of the parallelism that occurs when transfers from the network interface card to main memory proceed at the same time as capsule processing. This difference increases with payload size, as the network card can perform more work in parallel. It is pronounced for the C relay where the computational component is small, and barely visible for the Java relay and ANTS node, where the computational component is larger relative to the data transfer work.

A potential concern is that throughput will degrade due to memory caching effects when many protocols are being used concurrently. To test for this, I cycled up to 1000 different capsule types through the node in a pattern not amenable to caching, and measured the throughput. No statistically significant degradation was observed, most likely because the implementation is at user-level, though such effects are likely to be of concern in faster implementations.

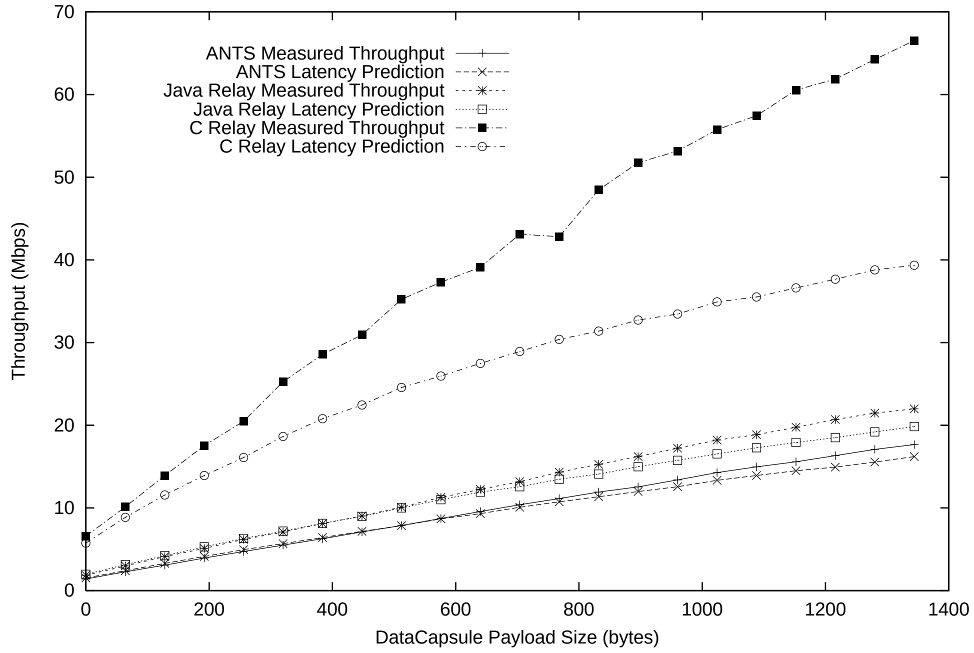

Figure 5-4: Throughput of an ANTS node as a function of Capsule Size

From class UDPChannel:

```
public Capsule receive() throws Exception {
  do {                      // loop until we get a capsule
    DatagramPacket p = new DatagramPacket(rxbuf.buf, rxbuf.len);
    try {
      dsock.receive(p);     // receive message from network interface
      if (p.getLength() <= 0) continue;
      // call step 2 onwards: check format and convert to capsule
      return decode(new Xdr(rxbuf, 0, p.getLength()));
    } catch (IOException e) {
      continue;             // interrupted
    }
  } while (true);
}
```

Figure 5-5: Reception of a Packet Buffer (Step 1)

From class Channel:
```
public Capsule decode(Xdr xdr) throws Exception {
  // check ANEP version and parse fields
  if (xdr.BYTE() != ANEPVERSION) throw new Exception("Wrong ANEP version");
  byte flags = xdr.BYTE();
  short typeID = xdr.SHORT();
  if (typeID != ANTSTYPE) { [[ processing for non-ants messages omitted ]] }

  // skip forwards and check ANTS version
  short headerLen = xdr.SHORT();
  xdr.index += (headerLen-2) * Xdr.INT + Xdr.SHORT;
  if (xdr.INT() != ANTSVERSION) throw new Exception("Wrong ANTS version");

  // call step 3: map type to forwarding method or invoke code loading
  Method m = codecache.fastLookup(xdr.buf, xdr.index);
  if (m == null || m.cl == null)
    return new DLBootstrapCapsule(m, xdr, client.getAddress());

  // call step 4: convert message buffer to capsule object
  Capsule c = Capsule.create(m.cl, xdr);
  c.chan = this;
  return c;
}
```

Figure 5-6: Receive Header Processing with ANEP (Step 2)

From class CodeCache:
```
public synchronized Method fastLookup(byte[] buf, int off) {
  // compute hashcode in same manner as TypeID class
  int hcode =
    ((buf[off+0]+buf[off+4] + buf[off+8] + buf[off+12]) << 24)+
    ((buf[off+1]+buf[off+5] + buf[off+9] + buf[off+13]) << 16)+
    ((buf[off+2]+buf[off+6] + buf[off+10] + buf[off+14]) << 8)+
    (buf[off+3]+buf[off+7] + buf[off+11] + buf[off+15]);
  int bin = (hcode & 0x7FFFFFFF) % size;

outer:  // look for a match in the right bin
  for (Method m = buckets[bin] ; m != null ; m = m.next) {
    if (m.hash == hcode) {
      byte[] bits = m.methodID.unwrap();
      for (int i = 0; i < 16; i++)
        if (bits[i] != buf[i+off]) continue outer;

      // found it, update LRU linked list
      movetofront(m.group);
      return m;
    }
  }
  return null;
}
```

Figure 5-7: Demultiplexing by Capsule Type (Step 3)

From class Capsule:
```
public Xdr decode() {
  Xdr xdr = new Xdr(this.xdr, this.xdr.index);

  // version handled, skip over type too
  xdr.index += Xdr.TYPEID;

  // get other fields
  src = xdr.INT();
  dst = xdr.INT();
  previous = xdr.INT();
  resources = xdr.INT();
  return xdr;
}
```

From class DataCapsule:
```
public Xdr decode() {
  // first, call base class
  Xdr xdr = super.decode();

  // now get our fields
  spt = xdr.SHORT();
  dpt = xdr.SHORT();
  data = xdr.BYTEARRAY();
  return xdr;
}
```

From class Capsule:
```
public static Capsule create(Class cl, Xdr xdr) {
  Capsule c = null;

  // allocate new memory
  try {
    c = (Capsule)(cl.newInstance());
  } catch (Exception e) {
    Entity.error("Class.newInstance failed for "+ cl);
  }

  // initialize it from message buffer
  c.xdr = xdr;
  c.decode();
  return c;
}
```

Figure 5-8: Decoding of a Capsule after Reception (Step 4)

From class DataCapsule:

```
public boolean evaluate(Node n) {
  if (n.getAddress() ==  getDst()) {
    return n.deliverToApp(this, dpt);
  } else {
    // call step 6 onwards: route and send
    return n.routeForNode(this, getDst());
  }
}
```

From class ChannelThread:

```
public void run() {
  Node engine = source.client;
  Capsule c = null;

  while (true) {
    try {
      // clear this each loop in case receive throws
      c = null;

      // steps 1 to 4: block while we receive and decode from the channel
      c = source.receive();

      // make it sequential, assume no recursion
      synchronized (engine) {
        try {

          // save context information on the side
          engine.context = c;
          c.evaluate(engine);
        } finally {
          engine.context = null;
        }
      }

      if (c.saveBuffer) source.rxbuf = new Xdr(source.rxbuf.length());
    } catch (Exception e) {
      // log a problem but continue unheeded
      source.log(Entity.L[0], "terminated " + c + "with " + e);
      e.printStackTrace();
    }
  }
}
```

Figure 5-9: Default Forwarding in the Capsule Framework (Step 5)

From class Node:

```
public boolean routeForNode(Capsule c, int dst) {
  // look up route in hashtable
  RouteEntry r = routes.get(dst);

  try {
    // call step 7 onwards:  header, encode and transmit
    return link.send(c, r.addr);
  } catch (NullPointerException e) {
    throw new NoSuchRouteException();
  }
}
```

Figure 5-10: Route Lookup (Step 6)

From class Capsule:

```
public Xdr encode() {
  Xdr xdr = new Xdr(this.xdr, this.xdr.index);
  xdr.index = 0;
  xdr.PUT(this.mid(), 0, Xdr.TYPEID);
  xdr.PUT(src);
  xdr.PUT(dst);
  xdr.PUT(previous);
  xdr.PUT(resources);
  return xdr;
}
```

From class DataCapsule:

```
public Xdr encode() {
  Xdr xdr = super.encode();
  xdr.PUT(spt);
  xdr.PUT(dpt);
  xdr.PUT(data);
  return xdr;
}
```

From class Channel:

```
 public final boolean send(Capsule cap, ChannelAddress dst) {
  // convert object to message, after calling step 7: header processing
  Xdr xdr = encode(cap);

  // call step 9: transmit
  if (xdr.offset != 0) {
    byte[] bytes = new byte[xdr.len];
    System.arraycopy(xdr.buf, xdr.offset, bytes, 0, xdr.len);
    return send(bytes, xdr.len, dst);
  }
  else
    return send(xdr.buf, xdr.len, dst);
}
```

Figure 5-11: Encoding of a Capsule for Transmission (Step 7)

From class Channel:

```
public Xdr encode(Capsule cap) {
  // source, etc. must be set already
  if (cap.src == 0) throw new SecurityException();

  // previous is updated for use by the code loading
  cap.previous = clientAddress;

  // resources decremented here just before we leave
  if (--cap.resources <= 0) throw new ResourceLimitException();

  Xdr xdr;
  if (cap.xdr != null) {
    // just update potentially changed header bits
    xdr = cap.xdr;
    xdr.index = Xdr.INT;
    short headerLen = xdr.SHORT();
    short packetLen = (short)(headerLen * Xdr.INT + Xdr.INT + cap.length());
    xdr.len = packetLen;
    xdr.PUT(packetLen);
    while (xdr.index < headerLen * Xdr.INT) {
      short optType = (short)(xdr.SHORT() & 0x3FFF);
      short optLen = xdr.SHORT();
      if (optType == DESTOPTTYPE) {
        xdr.index += Xdr.INT;   // Skip over scheme identifier
        xdr.PUT(cap.getDst());
        break;
      } else {
        xdr.index += (optLen - 1) * Xdr.INT;
      }
    }
    xdr.index = headerLen * Xdr.INT + Xdr.INT;
  }
  else {
    [[ processing for first-time capsules, e.g., from applications, omitted ]]
  }

  // resume step 7: write rest of capsule
  cap.xdr = xdr;
  cap.encode();

  return xdr;
}
```

Figure 5-12: Transmit Header Processing with ANEP (Step 8)

From class UDPChannel:
```
public boolean send(byte[] buf, int length, ChannelAddress dst) {
  // check validity of address
  try {
    UDPChannelAddress a = (UDPChannelAddress)dst;
  } catch (Exception e) {
    return false;
  }

  // send buffer as datagram
  DatagramPacket fr = new DatagramPacket(buf, length, a.address, a.port);
  try {
    dsock.send(fr);
  } catch (IOException e) {
    return false;
  }
  return true;
}
```

Figure 5-13: Transmission of a Packet Buffer (Step 9)

Operation	IP?	Time (%)
Packet Receive	yes	29
Header Processing	yes	5
Type Lookup	no	3
Capsule Decode	no	18
DataCapsule Evaluate	no	2
Route Lookup	yes	5
Capsule Encode	no	14
Header Processing	yes	7
Packet Transmit	yes	13
Other		4

Table 5.1: Profile of `DataCapsule` Processing Steps within an ANTS node

5.1.4 Profile

The baseline performance measurements show the overall runtime costs of an active node. This results from a number of processing steps, some of which are analogous to steps used in IP forwarding, and some of which are required to support the capsule programming model. To understand the key steps and their performance implications, I profiled an ANTS node.

The common-case steps for forwarding capsules of type `DataCapsule` are described below. The code that implements them is shown in Figures 5-5 through 5-13 to define forwarding in detail. This code is taken directly from the ANTS distribution and reformatted for presentation. I compare it qualitatively with the IP forwarding steps as specified in [Baker, 1995] by looking at the rough structure of common case IP processing (excluding IP options, ICMP and multicast datagrams) and contrasting it with the equivalent forwarding written as a new service in ANTS.

1. A message is received from the incoming network interface via the operating system. IP executing at user level would perform substantially the same processing.

2. The structure of the message is checked to ensure that it is a valid capsule. The checks are simple, and IP performs analogous checks.

3. The capsule type is mapped to the appropriate forwarding routine. This is done with a hash table lookup. There is no corresponding requirement for IP.

4. The capsule is converted from an external packet representation to an internal object representation. This step requires object allocation and member initialization by copying. It is an artifact of the current Java-based runtime that is not intrinsic to ANTS would be performed at compile time with a cast in a C-based implementation. IP implemented in C requires no runtime overhead overhead for this step.

5. A forwarding routine is invoked. To implement IP-style service, the `DataCapsule` forwarding routine simply invokes default forwarding via the active node. While default forwarding is the same as IP forwarding, expressing it as a service within ANTS generates some overhead to set up the call in a safe and generic manner.

6. A routing table lookup is performed. In the prototype, a simple hash table lookup is used, while in IP and a production ANTS system a more expensive longest matching prefix operation is needed. Note that the greater relative cost of this step in practice will result in a more favorable comparison for ANTS.

7. The capsule is converted from an internal object representation to external packet representation. Again, this step is a consequence of the Java-based runtime. It is not intrinsic to ANTS and is not necessary in a C-based implementation such as is used for IP.

8. Some header fields are updated, for example, by decrementing the resource limit and setting the previous node address. The updates are simple, and IP requires analogous updates, though fewer in number.

9. A message is sent to the outgoing network interface via the operating system. IP executing at user level would perform substantially the same processing.

As well as these per packet tasks, there are maintenance tasks that must be performed at a coarser granularity. An IP router must periodically process and exchange routing packets. An active node must in addition age the soft-store and code cache, and run the code distribution protocol when it is needed. These maintenance tasks do not noticeably impact forwarding performance because they occur relatively infrequently; code distribution overhead is potentially more variable and is discussed in the next section.

Table 5.1 shows the fraction of forwarding time spent in each of these steps. It is based on profile data collected directly by the Java runtime. The "IP" column indicates whether substantially the same processing step is required as part of IP forwarding. The measurements show that ANTS requires few expensive steps over IP-style functionality. Much of the forwarding time is spent receiving and transmitting packets buffers, tasks which are common to ANTS and IP. Of the remainder, only the encoding and decoding steps are expensive, which is an artifact of the Java-based implementation strategy. In ANTS, they require copying because of type safety restrictions, while in C a cast that has no runtime

Protocol	Capsule	Latency (us)	Slowdown
PIM	MulticastData	670	1.25
	JoinPrune	975	1.82
	RegisterStop	560	1.05
WebCache	Query	615	1.15
	Bind	685	1.28
	Redirect	600	1.12
	Activate	620	1.16

Table 5.2: Latency for different types of Capsules, expressed as an absolute time and Slowdown factor relative to DataCapsule forwarding

cost is performed. Overhead unique to the ANTS framework is small — 5% for type dispatch and the safe evaluation framework — as a result of the simple forwarding model. From a computational perspective, the profile data supports the argument that the overhead of active node can be kept low in those situations where a software-based implementation is viable.

5.1.5 Service Latency

The remaining effect on node performance is the overhead of running the service code itself. The cost of this depends, of course, on the complexity of the service. To avoid interfering with the capsules of other services, the time to forward the capsules of different services must be within the capabilities of the node and not greatly exceed the default forwarding time. This is facilitated by the node API operations, all of which complete rapidly and locally without depending on scheduling or hidden I/O, for example, by blocking due to multiple threads or disk writes.

To gauge the cost of typical services, I measured the performance of capsules that provide the multicast and Web caching services described in Chapter 4. I focus on the latency of a single representative path through the forwarding code. Throughput cannot directly be measured for all of the capsules, such as route construction capsules that are rate-limited, but can be inferred from latency. Table 5.2 lists these measurements. They suggest that the overhead of user-defined processing routines can be low, in most cases adding no more than 25% to the total processing time.

5.2 Code Distribution

The performance of the code distribution system is important because ANTS is built on the assumption that occasional bursts of code distribution activity will not interfere with the continuous provision of a service, as perceived by distributed applications that are using the network. To assess the validity of this assumption, two questions must be answered:

- What does it cost to distribute code along a network path?
- How often does this occur?

Protocol	Capsule	Code Size (bytes)	Load Time (us)	Load Time (capsules)
PIM	MulticastData JoinPrune RegisterStop	11849	17100	26 18 31
WebCache	Query	1844	4700	8
	Bind	1821	4750	7
	Redirect	1843	4850	8
	Activate	1991	4950	8

Table 5.3: Code Size and Code Loading Latency for different types of Capsule

Protocol	Capsule	A (us)	B (us)	C (us)	D (us)	Total (us)
PIM	MulticastData JoinPrune RegisterStop	700	1200	400 x 13	9750	17100
WebCache	Query	600	1250	450	2350	4700
	Bind	750	850	600	2550	4750
	Redirect	700	1250	450	2450	4850
	Activate	700	1250	500	2500	4950

Table 5.4: Breakdown of Load Latency Components

To answer the first question, I measured the code distribution costs of the prototype, and used the results to construct approximate analytical models that predict code loading performance along entire network paths. I find that, for small code sizes, code distribution costs are low in terms of loss, bandwidth, and latency. Code distribution is unlikely to adversely affect applications. To place the models in context, I compare them with static code loading and an alternative scheme that uses TCP connections and code servers. This demonstrates that the properties of the *demand pull* scheme provide a number of benefits.

The answer to the second question depends on the traffic characteristics of a large ANTS network, which are presently unknowable. This leads me to consider the impact of different operating regimes. I conclude that code distribution activity is likely to be limited enough to make the use of new services feasible for most situations of interest.

5.2.1 Measured Local Costs

Table 5.3 shows the performance of the code loading system between a pair of adjacent nodes for the multicast and Web caching services. The size of the code group and the additional latency attributable to the load operation are given. The measurements are taken under light load, with negligible propagation delay and ample bandwidth between nodes. No cross traffic is present since its effect is mainly additive: capsule forwarding and different types of code loading can occur concurrently and in a manner that is fully interleaved at the capsule level.

The loading times are positively correlated with code group size and small in absolute terms,

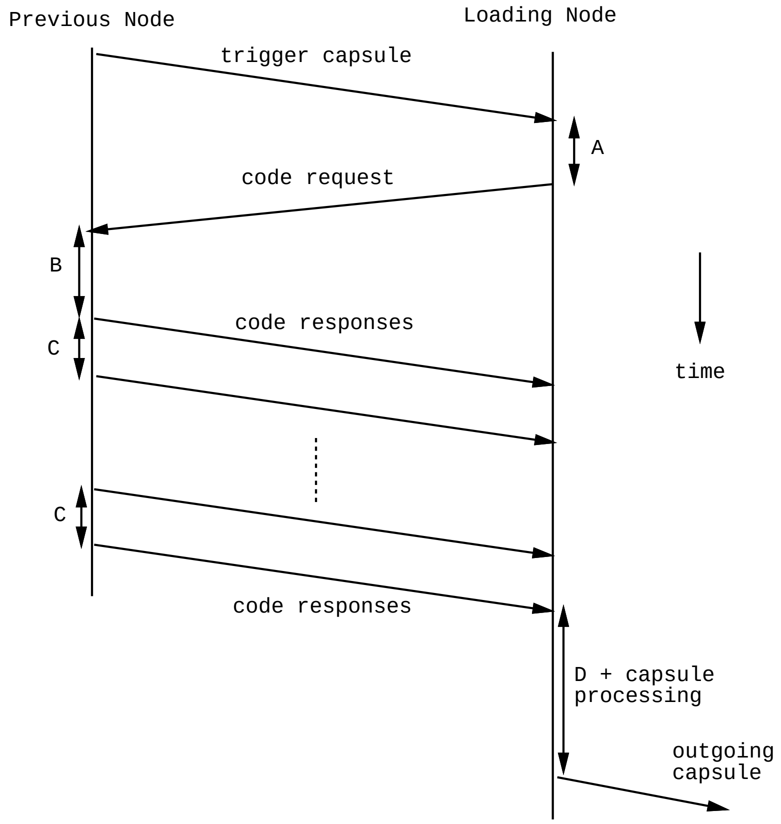

Figure 5-14: Components of a Code Loading Event

though they can be more than an order of magnitude greater than the corresponding capsule forwarding times. The largest load time occurs for the multicast service. This is because the PIM definition leads to a single large code group, since `MulticastData` capsules can create `JoinPrune` and `RegisterStop` capsules within the network. In contrast, the Web caching service allows the different forwarding routines to be carried separately and has much smaller times.

To better understand these times, I separated the total load time into its components. These are defined as shown in Figure 5-14: an initial delay before the load request capsule is sent (A), the delay until it has been processed and the first response appears on the wire (B), the delay between subsequent responses (C), and the final delay while the loaded code is reassembled before execution (D).

These results are given in Table 5.4. They suggest that the loading delay stems mainly from two sources, both of which can be addressed in an improved implementation. More than half of the load time is spent reassembling and checking the code before execution. This is due in part to a Java implementation of the MD5 algorithm and buffer copying. The delay to send out the first capsule is noticeable and also larger than necessary because the code is not kept in a format that is ready for transfer. Further improvements can likely come from a more compact code transfer format. This is because the current format is larger than necessary (see Section 3.4 for details) and costs such as fingerprint verification and the generation of response capsules are proportional to code transfer size.

5.2.2 Approximate Global Costs

The overall costs of code distribution seen by an application are those of loading a new service along an entire network path. To help understand these costs, I constructed approximate analytic models for the additional loss, bandwidth consumption and latency seen by capsules that require code distribution. These models compare ANTS to a conventional network, which has no additional code distribution costs because services are statically deployed.

A model of code distribution along a network path first requires models of a network path and code to be distributed. I begin by postulating the following:

- A path between two applications that is made up of A equal length "hops" between adjacent active nodes. There can be an unspecified number of non-active nodes too, but they do not affect the model. For a relatively long path between MIT and Berkeley, there are approximately 20 routers, so in the case that the path is fully active, $A \approx 20$.
- A one way path delay of D, which summarizes the latency contributions of propagation, transmission, and congestion delay. For a relatively long path between MIT and Berkeley, $D \approx 50$ ms.
- A single code group that can be transferred with N capsules, each containing 1 KB of code, the fragment size used in the prototype. I consider $N = 16$ to be a large group. This is larger than the code size of PIM, the largest of the services I have studied with ANTS.

I first consider capsules that are lost due to code distribution failures. For a network in which there is sufficient memory to cache service code while it is being used, losses must stem from congestion or bit errors. I assume that the probability of a single capsule being lost in this fashion is L, lumped at a single bottleneck node along the path. Measured loss rates vary considerably in the Internet depending on factors such as time-of-day. Paxon found average losses of $L \approx 5\%$ and a single bottleneck in the vast majority of transfers [Paxon, 1997]. For this simple model to be adequate for each capsule, code distribution must not itself congest the network. For the prototype, this is the case only when N is small so that router buffers are able to absorb code transfer bursts. (As an indication of "small", the dynamics of TCP result in bursts of approximately the bandwidth-delay product, which for cross-country trips at T1 rates is approximately 16 KB.)

In ANTS, the loss of any code distribution capsule will cause the loss of the capsule waiting for the code. There are $N + 2$ capsules in a complete load exchange, counting the original capsule and the load request as well as the responses. The probability that a full load succeeds without the loss of any of these capsules is:

$$Probability_{Full\ Load\ Succeeds} = (1 - L)^{(N+2)} \tag{5.1}$$

This probability is a lower bound on the probability that subsequent loads succeed, since the scheme can make incremental progress. It can thus be inverted to provide a weak upper bound on the expected number of capsules that must be sent by an application to complete the load and result in a capsule being delivered to the destination. This number grows exponentially with the size of the code group, and for $L \approx 5\%$ reaches two when $N \approx 12$. This result reflects that fact that the demand pull protocol is effective for small code group sizes, and would require further support mechanisms to handle large code sizes without interfering with loss.

The bandwidth consumed by code distribution along a path is a function of the size of the code group. I ignore loss since the number of lost capsules will be small compared to the total (because L is much closer to 0 than 1) and losses contribute only a small amount to the total since both schemes make incremental progress. All N code capsules and one small load request capsule must be sent across each hop:

$$\begin{aligned} Bandwidth_{\text{ANTS}} &= N + 1 \ capsules \\ &\approx N \quad KB \end{aligned} \tag{5.2}$$

This simple result shows that, assuming a capsule follows a shortest path route, ANTS consumes close to the minimal bandwidth of one copy of the code per hop.

The latency that code distribution adds to a successful transit of the network by a capsule is also a function of code size. Again, I conservatively ignore the affect of prior code distribution losses, since these can only reduce the latency of a successful loads because the scheme makes incremental progress. To charge for the processing costs associated with code distribution, I assume that there is a delay of approximately 1 ms per code loading capsule, and that these delays combine sequentially for each capsule involved in the code load sequence. This is roughly the measured behavior of the prototype.

As well as the capsule processing costs, one round trip latency of $2D/A$ between each pair of active nodes is needed to send the code request and return the first response. Combining these factors, the total added latency for the entire path is thus:

$$Latency_{\text{ANTS}} = A((N+2) + \frac{2D}{A}) = A(N+2) + 2D \quad ms \tag{5.3}$$

This result is linear in path length, code size and round-trip delay. For the cross-country path ($A = 20$ and $D = 50$), and a large code group ($N = 16$), ANTS code loading adds 460 ms. This is well within the normal transit time variations measured in the Internet and reported in [Paxon, 1997] as between 0.1 and 1 seconds. This implies that occasional delays within the network due to code distribution activity will not be identified as such by applications.

5.2.3 Comparison with an Alternative

To help place the ANTS scheme in context, I construct similar models for an alternative code distribution scheme based on TCP connections to a well-known code server. I refer to this scheme as SERVER. It is motivated by active network systems such as [Decasper and Plattner, 1998] and [Nygren, 1999], and is interesting primarily in the ways that it contrasts with ANTS.

With SERVER, by definition, capsules are no more likely to be lost if they require code loading than not. This is because the code transfer is reliable, so that compared to ANTS, losses that would be handled by applications are handled by the code server within the network. Thus instead of directly seeing increased loss, applications will perceive increased latency and competition for bandwidth.

To model these effects, I assume the same kind of network path as before with a single code server that lies directly on the path at its midpoint. This is the optimal location for a single server solution assuming shortest paths. Again, I ignore loss because its impact is small.

For SERVER, bandwidth is consumed by the same kind of capsules as for ANTS, plus two additional capsules for TCP connection establishment, two for teardown, and N for data acknowledgements. This results in the transfer of $2N + 5$ capsules per load. All of the additional capsules are small, so that the volume of the transfer is still approximately N KB. However, since the code server is in the middle of the path, these code loading capsules will transit some hops multiple times. The sum over active nodes of the number of code loading capsules times the number of nodes traversed yields the total number of capsule-hops required to transfer the code along the entire path. Splitting the sum into two parts (since the code server is at the midpoint) and dividing by the number of nodes yields the bandwidth consumed per hop:

$$\begin{aligned} Bandwidth_{\text{SERVER}} &= \frac{2}{A} \times \sum_{i=1}^{i=A/2} i \times (2N+5) \\ &= \frac{(2N+5) \times (A+2)}{4} \; capsules \\ &\approx \frac{N \times (A+2)}{4} \quad KB \end{aligned} \tag{5.4}$$

Unlike ANTS, this result is linear in the number of active nodes. For a cross-country path ($A = 20$), the bandwidth multiplying factor of SERVER over ANTS is 5.5. This illustrates the advantage of hop-by-hop transfer of code along the path that the capsule follows compared to loading from a fixed location. Schemes in which loading occurs from many locations can of course be developed, but have hidden costs such as distributed directory support and are still likely to be more expensive than ANTS unless directory nodes are as abundant as active nodes.

For SERVER, additional latency comes from the longer path between the code server and active node and the dynamics of slow-start, which spreads packets over time. I model TCP dynamics simply, conservatively ignoring loss and assuming that the processing time for TCP messages is zero so that their contribution to the total delay comes only from round-trip times. I still charge code capsule processing times because in the prototype much of their cost occurs after all code capsules have been received.

Compared to ANTS, SERVER requires one extra round trip for connection establishment, but not for teardown since it can proceed in parallel with capsule processing. Slow-start doubles its window every round-trip time (RTT) before congestion is detected. The best time to send P packets is thus $floor(\lg P) \times RTT$. Combining this with ANTS costs and summing over the different RTTs along the path, which are multiples of D/A:

$$
\begin{aligned}
Latency_{\text{SERVER}} &= A(N+2) + 2\sum_{i=1}^{i=A/2} \frac{4iD}{A} + \frac{iD \times floor(\lg N)}{A} \\
&= A(N+2) + 2D + AD + \frac{D}{4} \times floor(\lg N) \times (A+2) \\
&= Latency_{\text{ANTS}} + AD + \frac{D}{4} \times floor(\lg N) \times (A+2) \quad ms
\end{aligned}
\tag{5.5}
$$

This form shows the contribution of increased path length and slow-start clearly. Recall that for the cross-country path ($A = 20$ and $D = 50$), and a large code group ($N = 16$), ANTS code loading took 460 ms. Loading along longer paths with SERVER increases this latency by 1000 ms. Slow-start increases the latency by a further 1120 ms. This gives a total of 2560 ms, which is beyond the range of normal variation and will likely be perceived by applications as loss.

5.2.4 Frequency of Code Distribution

The overall effects of code distribution on the network depend on its frequency as well as the cost of loading individual network paths. The frequency of loading depends in turn on the traffic patterns of a large ANTS network, which I consider in terms of two operating regimes.

I expect that, while many services will be used for experimentation over time, few services will account for the vast majority of capsules at any given time, in the same way that most traffic today is HTTP, SMTP and FTP. Assuming that node caches are large enough to hold the "working set" of network services, commonly-used services will be loaded into the network infrastructure once and stay there for their useful lifetime. In this operating regime, code loading activity is rare and its affect on the network as a whole negligible.

Alternatively, active networks such as ANTS allow custom services to be constructed as needed. In an operating regime where there are many custom services (which by their nature are not shared by different applications) a greater amount of code loading is needed. In the worst case, a custom service must be loaded once at the beginning of each application session. (Again, I assume that network has been reasonably dimensioned so that the "working set" of services fits in node caches.)

In this extreme, the impact of code loading depends on the session size. As a simple model, I assume that loading a new service causes each node to process on the order of ten capsules, totalling 10 KB of data, and that each of these code capsules requires approximately the same amount of processing as a capsule for which the service code has already been loaded. This is roughly the behavior of the prototype.

Despite these worst case assumptions, it appears feasible to deploy new ANTS services with minimal overhead even for moderately sized sessions. For example, code loading would consume 1% of the bandwidth and 1% of the node processing time (effectively slowing the whole network by around 1%) when the session size was 1000 packets of 1 KB each. This might correspond to the transfer of a 1 MB video clip.

5.3 Security

If not well-designed, an active network might suffer from protection, resource management and other types of security problems as a result of allowing untrusted code to execute in sensitive contexts. To evaluate ANTS in this respect, I characterize the threats that new services present, how they are handled, and how they compare with the behavior of the standard Internet protocols. I find that ANTS raises few additional security concerns compared to the Internet. Most threats can be handled by active nodes locally and without trusting service code or external parties, with the exception that global resource consumption must be certified as reasonable by a central authority.

5.3.1 Service Protection Threats

Protection threats are those that directly affect the execution of a service within the network so that it is no longer isolated from other code and hence no longer guaranteed to behave correctly. Compared to the Internet, additional protection threats in ANTS must stem from the transfer and execution of code within the network. There are several potential ways that services can affect the behavior of each other in unintended ways that cannot occur in the Internet except in response to a faulty implementation:

- the node may be corrupted by service code
- service code may be spoofed
- service state held at a node may be corrupted by other service code
- capsules using the new service may be manipulated by other service code

The first threat, corrupting the active node itself, is met through the use of safe evaluation techniques for executing service code. While the ANTS toolkit relies on the properties of Java, other implementations could be based on proof-carrying code (PCC) or software-based fault isolation (SFI). The net result is that the sound implementation of the node operating system implies that it cannot be crashed or otherwise corrupted by arbitrary service code.

Code spoofing, the second threat, is met through the use of fingerprint-based names for capsule types. ANTS uses a code hierarchy of forwarding routine, code group and protocol. A belief in the one-wayness of the fingerprint function, which is based on MD5 [Rivest, 1992] in the prototype, implies that none of these code units can be spoofed for a given capsule. This means that there is no ambiguity as to which forwarding routine to run, and that new forwarding routines cannot be added to an existing protocol to, for example, capture private information.

A third means of interfering with a service is to corrupt the state that it maintains at an active node. This is prevented in ANTS by building on the fingerprint-based capsule types to separate the state of different services. The only state retained at a node after a capsule is forwarded is that placed in the soft-store. Access to the soft-store is partitioned by protocol type, which guarantees that state maintained on behalf on one service will not be manipulated by code corresponding to another service.

A forth threat is that capsules using a new service may be manipulated by other service code. This is prevented by supplementing the previous mechanisms with a restricted active node API. There are no calls exported by this API that allow one capsule to control the forwarding of subsequent ones, and all state that can indirectly influence forwarding is carefully managed by the node. Data kept in the soft-store is partitioned by service for protection, as described above, and data shared between services such as the default routing table is treated as read-only. This means that it is not possible to construct a service that, for example, searches the network and discards capsules belonging to another service. Nor is it possible to construct a service that, for example, alters the default routes that will be followed by capsules belonging to another service.

In sum, ANTS provides measures that defeat all of these kind of threats. This implies that the protection provided by ANTS is equivalent to the protection afforded to protocols in the Internet today. (Arguably, ANTS could provide better protection by addressing human errors. For example, bugs in many protocol implementations can be caught by one small core that relies on safe evaluation techniques, while the use of protocol types that are based on fingerprints of the corresponding code prevents versioning errors. But I ignore such factors for a rough analysis.) Further, this protection is provided despite the fact that ANTS services are implemented with mobile and untrusted code, rather than static trusted code. No central trusted authority is needed to obtain the required level of protection, which eliminates accidental as well as malicious attacks.

In the larger context, however, Internet security is considered poor, and is in the process of being extended. Both ANTS and the Internet protect different protocols from interfering, but not different users of a protocol from interfering with each other. This kind of protection can be provided using measures such as authentication and encryption. In the Internet, the IETF IPSEC working group is adding these features to IP. Similar measures could be added to the ANTS architecture by extending the node API with security calls that services could combine to meet their own security needs. For example, in the case of Web caching, capsules that bind a cache to an active node should be authenticated, since allowing arbitrary users to send these capsules has the potential to disrupt the caching service. It is necessary, however, that end-to-end encryption and authentication be composed with a service rather than simply layered below it in the protocol stack. This is because if a packet is completely encrypted, the ANTS header fields would be inaccessible within the network. The same constraint applies to value-added processing in the Internet today, for example, Snoop-TCP and firewalls that need to examine TCP header fields.

5.3.2 Resource Management Threats

In contrast with protection threats, resource management threats affect the performance of a service, rather than its correctness, by consuming shared resources in an unreasonable fashion. By "unreasonable" I mean in a manner that can starve another service, rather than in a strictly balanced manner according to some notion of fairness.

In the Internet, the resources that are consumed as a packet is forwarded from source to destination are relatively well understood in terms of a static model that strictly limits bandwidth, memory and processing time. In an active network, resource consumption is driven by programs. It can be much more dynamic in nature, and must be restricted in its form to ensure that the effects of one service on others or a region of the network are reasonable.

In ANTS, one service can potentially interfere with the performance of another in three ways. The first two of these threats are prevented in the Internet today by the design and correct implementation of IP and other network protocols.

- a capsule may consume a large, possibly infinite, amount of resources at a single node;
- a capsule and other capsules it creates within the network may consume a large, possibly infinite, amount of resources across multiple nodes; and
- a service may send a large, possibly infinite, number of capsules across a region of the network.

The first threat, too many resources used at a node during the forwarding of a capsule, is detected and handled by each ANTS node. This is relatively straightforward because it is a local matter. Long running forwarding routines are broken with a watchdog timer. Access to memory and bandwidth is also limited by using the `resources` header field, which prevents a capsule from sending too many outgoing capsules and placing too many objects in the soft-store. Other resources, such as the stack, can be similarly limited by checks in the runtime or at loadtime, though this is not done in the prototype. It would also be straightforward to enforce different local limit schemes than are used in the prototype without trusting the service code, for example, a set maximum bandwidth and soft-store allowance. Thus while the design of an active node may pose engineering issues, local resource management concerns are not complicated beyond those raised by the IP model.

The second threat, bounding resource consumption across a group of nodes, is more difficult to address. In both the Internet and ANTS, packets and capsules would consume unbounded resources if blindly forwarded around a routing loop. In IP, this is prevented by using a TTL field (or Hop Count in IPv6). In ANTS, the same mechanism is used to prevent an analogous class of infinite loop program errors.

This mechanism, however, does not prevent capsules from consuming an excessive (though bounded) amount of node resources if directed to do so by a poorly designed service, for example, by ping-ponging between two nodes until the resource limit is exhausted. I have argued that the resource consumption of one capsule is "reasonable" if it visits each node no more than k times, assuming there are no loops in the standard routing tables, and where k is a small number such as one or two. This definition readily allows multicast services and accommodates Internet protocols, but disallows loops and flooding behavior such as ICMP smurf attacks. I refer to protocols that satisfy this definition for some value of k as $k - bound$.

To ensure that services satisfy this definition for small values of k, nodes rely on a security policy in which only service code that is certified by a trusted authority with a digital signature is freely executed. This mechanism is weaker than the other mechanisms of the node operating system in that it depends on external parties. Nonetheless, it is arguably at least as strict as that used in the Internet today and yet provides a system that is capable of evolving much more rapidly than is possible today.

It is at least as strict as standardization (in the case that the certification authority is the equivalent of the IETF) because a single implementation is approved. This implementation can be tested, and bugs that arise when implementing a standard are avoided. It can proceed more rapidly because certification differs from standardization in that it does not seek to define a single preferred behavior. Rather, it seeks to establish that a service makes reasonable use of overall network resources, regardless of whether it is considered an effective means of accomplishing a particular task. Further, the ability to run new but not yet certified services at a reduced level of performance simultaneously allows for rapid experimentation.

The final threat, that heavy use of one service may starve another, exists in both ANTS and the Internet. It is not well addressed in the Internet today because drop-tail or FIFO routers, which provide a poor separation of traffic flows, are prevalent. This area is the subject of much research, and scheduling algorithms such as RED and fair queueing that provide a greater degree of isolation are in the process of being developed and deployed. This type of threat is no greater in ANTS than the Internet, and the same measures are equally applicable.

5.4 Expressiveness

The example services explored in Chapter 4 demonstrate that ANTS provides a kind of flexibility that is not readily available in the Internet. To evaluate the expressiveness of the system, I characterize the kinds of service that ANTS can readily introduce, and those that are not well-suited to the capsule model.

I find that ANTS is able to introduce a wide variety of services that are difficult to handle in the Internet today. Rather than list potential new services that have been mentioned during the course of the dissertation so far, I have chosen to present a number of design patterns. These patterns give a good idea of the kind of potential services because they abstract the programming forms that have proven useful across a range of services. In describing the kind of services that cannot readily be introduced, I use examples to suggest how complementary techniques can act in synergy with ANTS.

5.4.1 Service Design Patterns

Over the course of experimenting with multicast, Web caching and other services, a number of programming forms came up repeatedly. These forms avoid design pitfalls and fulfill their function relatively efficiently. They tend not to be straightforward mappings of distributed graph algorithms, such as breadth first search, because of the need to accommodate constraints such as unreliable transfer and storage combined with a mix of active and non-active nodes.

Custom Routing and Forwarding

A strategy to forward capsules along customized routes is to use one type of capsule to establish the route and another to follow it. The capsule that establishes the route can express it in terms of default routes and maintain it in the soft-store. The capsule that follows the route then simply looks up the route in the soft-store and proceeds along it to the next active node. The PIM service utilizes this pattern and extends it to multicast.

This strategy is simple to understand and code. It is efficient since the rates of route construction and use are decoupled. Typically, the routing capsule will be sent only periodically to maintain the contents of the soft-store, allowing the overhead of route construction to be amortized over a flow of packets. The resulting size of the code is also small since the routing and forwarding capsules can be placed in separate code groups. Further, with a standard form of routing record and identifier, for example, Multi-Protocol Label Swapping, this strategy is amenable to high-end routers, where forwarding is performed with hardware support.

Forwarding with Filtering

A number of effects can be achieved by having capsules discard themselves during forwarding depending on the contents of the soft-store. The state that is maintained can be considered a filter function to which many capsules are input but only few are output. As with all services based on the soft-store, the filter function is not guaranteed and can retain memory only over a short period of time. Soft-state failures will result in fewer capsules being discarded than expected.

Some examples of filter function classes and their effects are:

- Discard based on number. Passing only the first capsule per application token (such as a sequence number) produces route flooding patterns such as NACK suppression.
- Discard based on time. Passing only x capsules every y seconds was used to reduce the overhead of PIM route maintenance by limiting the aggregate rate of packet streams.
- Discard based on loss. If one part of an application unit such as an MPEG frame is lost due to congestion, then the remainder can also be dropped. More interesting alternatives that require somewhat more computation are priority queuing and drop priority.

Forwarding with Additional Processing

Capsules traveling through the network can be made to undergo additional processing at a particular location. This pattern is demonstrated as part of the Web caching service of Chapter 4, in which cache processing is applied to Web requests at selected nodes.

To tie additional processing into the regular capsule processing in a robust manner the properties of the soft-store are exploited as follows. The process controlling the additional processing (for example, the Web cache) periodically refreshes activation information in the soft-store of the active node at which capsules are to be intercepted for additional processing. Capsules being forwarded through the active node check this information, and divert themselves for the additional processing if needed. Since the soft-store ages information, capsules will only retrieve the information needed to divert themselves if the controlling process is functioning, and otherwise will continue regular processing. The controlling process can also get an indication that the active node is running by relaying capsules through it in the same manner as the Bind capsule of the Web caching service.

This pattern can be used to tie heavyweight capabilities at nodes within the network infrastructure, representing transcoding, compression, and caching engines, into the chain of capsule processing. It is more powerful than the node extensions available in the ANTS toolkit, since the additional capabilities can reside at nearby nodes and capsules diverted to reach it. This pattern can also be implemented efficiently (though it is not in the prototype) by using the observation that the activation switch essentially re-binds the forwarding routine that corresponds to a given capsule type, and thus occurs on a much coarser granularity than the packet rate.

Optional Processing and Piggybacked Events

As capsules travel through the network they can be subject to processing that is not always required, and can trigger other events by their presence. An example of the former is that the contents of a capsule might be encrypted if it is about to leave a trusted administrative domain or fragmented if it is larger than the next link MTU. An example of the latter is that if routing state is found to be incomplete during forwarding then route construction messages might be triggered.

Such changes can often be signaled with special-purpose header fields on the corresponding capsule. This approach is both robust and efficient.

It is robust because the changes are associated with individual capsules and as such can adapt on a per capsule basis. Alternatives that try to amortize changes over all of the capsules of a flow are fragile because it is difficult to estimate how long path characteristics will be stable. Alternatives such as signaling events with additional capsules may result in intermediate states being visible, such as an event whose trigger is lost, or contribute to instability in the same manner that source quench packets have proved to be of dubious utility in times of congestion.

It is efficient because no additional capsules are created and the space consumed by header fields is small. With an improved implementation, it would be possible to eliminate this overhead completely by encoding this state as part of the capsule type.

5.4.2 Complementary Services

The example services and preceding design patterns show that ANTS provides sufficient flexibility to express a wide variety of new network services. By design, however, it is not well suited to all kinds of services. In particular, the three classes of service described below are difficult to model and help to characterize the solution space of the approach. These services are best deployed by complementary means.

Imposing Policy on Existing Services

Since the processing that can be applied to a capsule is determined by its type when it is sent into the network, it is not possible to deploy processing that augments an existing service in a fully backwards-compatible manner. Instead the ANTS approach focuses on innovation, where users request the new service directly. Firewall processing, for example, cannot be deployed in a straightforward manner because it will not be applied to capsules unless they request it, defeating the purpose of the firewall. This restriction is deliberate, since the remote upgrading of processing in a backwards-compatible manner has obvious security ramifications. Supporting these services would complicate the basic model with requirements for naming different flows and authorizing users to manipulate them.

Instead, these kind of services are handled quite effectively by the existing pointwise models, for example, Cisco IOS upgrades, since they are often intended to impose policy at a particular location in the network, rather than processing along a path. The models can be easily combined with the addition of appropriate node API calls that include authorization checks. This combination might also be useful to permit select nodes to accept old-style packet formats for backwards-compatibility and map them into new format capsules for upgraded service.

Heavyweight Tasks

Customized forwarding routines are intended to be compact and execute at the packet rate. Services that require large amounts of code or processing are therefore not well supported. While there is no hard and fast line between these regimes, I expect that tasks such as transcoding (between different audio and video formats) and compression are beyond the capabilities of the code distribution system. Further, other heavyweight services that include reliable transport or stable storage cannot be expressed within the ANTS architecture because the node lacks the necessary API calls.

Instead, heavyweight services are modeled in the ANTS architecture as capabilities that are built in to an active node as part of the heterogeneity of the network infrastructure. For example, a router may or may not have an attached Web cache. This view does not mean that heavyweight components within the network infrastructure cannot evolve, but rather than they cannot be extended at the capsule level.

The ANTS model works in synergy with this evolution outside of the capsule model because once the capabilities of a node are changed by the local administrator all users have immediate access via new ANTS services. This model of operation is supported by the ability as part of the node API to query if a given extension is present or not. Unlike transparent upgrades to the Internet, a new ANTS service can use the capabilities present at a node to determine how it should be forwarded, rather than accept some default configuration. Further, nearby nodes supporting heavyweight capabilities can be combined into the chain of processing as was demonstrated by the Web caching example. Thus the ANTS approach works well with a network infrastructure in which components are embedded to support value-added services: it provides the computational glue that is able to combine the capabilities of the embedded components in a flexible manner.

Widely-Held Assumptions

Finally, a number of aspects of the architecture cannot easily be changed because they are widely shared across the network. This includes assumptions that are hard-wired into the capsule format (such as addresses, types, and resource limits) as well as capabilities required by all active nodes (such as default routing and code distribution).

The chief benefit of ANTS compared to the Internet is that the number and scope of these assumptions is greatly reduced. While it would perhaps be preferable to have an architecture that encoded no assumptions, this position is untenable. It would result in every node shouldering the burden normally borne by the sum of network nodes, for example, the discovery and maintenance of a routing database. It would result in the repeated construction of commonly used mechanisms, such as code caching to improve performance. And it would defeat properties that bound the program form of services for the overall benefit of the network, for example, resource limits that prevent infinite loops, and a naming scheme that efficiently separates service state.

Changing widely shared assumptions requires versioning (and hence an extended transition period) if a backwards-compatible upgrade cannot be designed. This is exactly what happens today for IP, and a version field (analogous to the IP version field) exists at the beginning of the ANTS header fields for this purpose. In some cases, leeway is available without the need for versioning. For example, default routing might be upgraded by extending the node API, which is effectively a version change, but one that can be compatible with nodes that are not upgraded. Similarly, with a careful design of the node API, new code distribution services might be synthesized by transferring the code that implements them.

An interesting case in point is that of addresses, since one of the necessary changes to the Internet is an increase in the size of the address space as part of IPv6. This task can readily be accomplished in ANTS with a source routing mechanism — expressed in terms of existing addresses for compatibility — that provides the appearance of a deeper hierarchy of addressing. This is what was proposed for PIP [Francis, 1993, Francis and Gondivan, 1994], a candidate for IPv6, and effectively what happens with the NAT (Network Address Translation) boxes that are being used to increase the address space in the Internet today [Egevang and Francis, 1994][1]. Conversely, an ANTS network cannot easily accommodate the wholesale change to IPv6 addresses in all capsule formats, since it cannot change the behavior of all nodes at the same time. Of course, IPv6 is proving to be extremely difficult to deploy in the Internet for precisely this reason, requiring complex transition plans to a new address space rather than a natural evolution of the existing one. Arguably, approaches that can be more effectively localized (and so are more amenable to introduction in ANTS, would be more successful.

[1]See also Internet Drafts, *Traditional IP Network Address Translator (Traditional NAT)*, P. Srisuresh and K. Egevang, July 1998, for updated usage, and *Network Address Translation – Protocol Translation (NAT-PT)*, P. Srisuresh and G. Tsirtsis, August 1998, for the application of NAT to IPv6 transition plans.

5.5 Application Performance

A compelling reason to use new network services is to improve performance, and so it is natural to ask whether ANTS can be used to improve the overall performance seen by applications. I consider this question in three parts:

- Are there new network services that can improve performance?
- Can ANTS introduce these types of services?
- Does the benefit of using these services outweigh the cost of using ANTS?

That the answer to the first question is "yes" is evidenced by several examples. First, Shenker recently showed that drop priority within the network could increase the performance of layered video applications (as measured by a utility function) beyond what could be achieved by a purely end-system approach [Bajaj *et al.*, 1998]. An upper bound of around 36% improvement between drop priority and the optimal (and currently unknown) end-to-end scheme was demonstrated, with gains compared to the current end-to-end approach (RLM [McCanne *et al.*, 1996]) of around 50%.

Second, many reliable multicast protocols rely on processing within the network to improve the performance of local error recovery [Papadopoulos *et al.*, 1998, Lehman *et al.*, 1998, Levine and Garcia-Luna-Aceves, 1997]. For example, ARM results reported in [Lehman *et al.*, 1998] demonstrate a recovery latency of one round-trip time across a range of group sizes for which SRM [Floyd *et al.*, 1997] (a widely used end-to-end scheme) recovery latency is consistently larger by up to a factor of two. The total number of hops that NACK and repair packets traverse to recover a single loss also grows more slowly than SRM as group size increases, and differs by roughly a factor of two for moderately sized groups (of 25 or more members).

Third, the service that delivers timely data such as stock quotes with a specified currency (described in Chapter 4) extends the range of today's Web caches. When timely data is popular, the benefits of network-based caching can be significant even for tight currencies. Simulations results reported in [Legedza *et al.*, 1998] describe scenarios in which server and router load is reduced by up to 50% and client latency by 20%.

As a final example, the simple PMTU service presented as an example in Chapter 3 can improve performance. Without it, discovery is more likely to result in the generation of an ICMP error message, with the subsequent loss of a data packet and round-trip of delay. Since the service can be piggybacked on the exchange of other packets, it incurs no additional cost in terms of either bandwidth or delay. Further, path MTU discovery can occur at connection setup time when using the service, but would typically occur afterwards without it (since a large TCP segment size must be negotiated). This means that it can save further packets.

Note that these examples necessarily ignore the issue of mechanistic complexity. That is, routers that support more services may be more complex to construct, raising the possibility of constructing a faster router that supports fewer services. This effect is dependent on many factors and not readily quantified. I argue that the increased computational resources that are present in an active network compared to a conventional one do not preclude a useful performance comparison. This is because without an active network there is no clean and robust way to take advantage of additional resources.

The answer to the second question is also "yes." Of the services described, two are already implemented within the toolkit: ARM and the PMTU example. (Note that ARM bears no relation to the PIM service. ARM adds reliable multicast primitives on top of an unreliable multicast foundation, while PIM creates the unreliable multicast foundation.) Of the remaining two, stock quote caching was described in Chapter 5 and is readily expressible in terms of the capsule and active node model. Drop priority requires access to congestion information at a node. It should be straightforward to express, but cannot be implemented in the ANTS toolkit because of implementation artifacts that prevent congestion loss from being observed. Aside from this defect, the ANTS programming model is able to introduce these kind of services without difficulty.

Finally, the answer to the third question is "yes" across a wide range of network topologies and usage patterns. The overheads of using an ANTS service stem from two sources: capsule processing and code loading. Both can be small for networks in which a software-based router implementation is viable and services are popular or sessions are moderately sized.

The measurements presented in this chapter suggest that ANTS nodes can be profitably deployed in the Internet today without reducing capacity in bandwidth-limited situations, for example, across wireless, modem, and access links up through T1 speeds. Such nodes represent an appreciable fraction of the IP routers today (since there are relatively many low performance nodes and relatively few high performance ones) and those where active network services are most likely to improve performance.

The analysis of the code distribution system suggests that code groups for new services can be kept compact, at less than 16 KB. This in turn means that the loss, bandwidth and latency overheads of loading a network path can be kept low enough that they are not perceived by applications using the new service. The overall effect of code loading on the network is also likely to be negligible because useful services will be shared between many applications. Even with worst case assumptions, many new services can be supported with minimal overhead when the session size is moderate.

Chapter 6

Related Work

This chapter compares ANTS with other approaches that provide a means of introducing new services or otherwise increase the flexibility of the network.

6.1 Configurable Protocol Suites

A number of protocols provide mechanisms for supporting a space of possible processing, rather than a single fixed point. This capability can be seen to a limited extent in the IP options [Postel, 1981b], header extensions in IPv6 [Deering and Hinden, 1995], TCP options [Postel, 1981c], and HTTP extensions[1]. I refer to them as configurable protocols rather than programmable protocols, since they support some amount of flexibility but do not involve the transfer and execution of code.

Some of the more flexible configurable protocol suites are described below. They are relevant to this research in that they demonstrate the kind of flexibility that can be offered without requiring an active network, and hence the kind of flexibility that an active network must improve upon to be worthwhile. In general, they do offer greater flexibility that the existing Internet, but appear limited in their abilities to introduce new services when compared to active networks.

6.1.1 The x-kernel

The x-kernel provided a collection of protocol elements (called micro-protocols) along with a generic mechanism for composing them based on layering [Hutchinson and Peterson, 1991]. It demonstrated that larger protocols, such as remote procedure call (RPC), could be expressed in terms of many smaller components with the layering model while still resulting in a high performance implementation [O'Malley and Peterson, 1992].

While the thrust of the system was to explore protocol implementation techniques, it provided dynamic composition of micro-protocols on a per packet basis. This allowed new

[1] Internet Draft, *HTTP Extension Framework for Mandatory and Optional Extensions*, H. Frystyk Nielsen, P. Leach and S. Lawrence, August 1998.

protocols to be expressed and introduced. Another feature of interest was the existence of virtual protocols, which were included or not depending on criteria such as the length of the message to be sent. In terms of programming language mechanism, the virtual protocols model "if" statements.

6.1.2 Protocol Boosters

Protocol boosters also leverage the layering model to alter processing within the network [Feldmeier *et al.*, 1998]. The modularity of a layered protocol organization allows sub-layers to be inserted between two points in a manner that is transparent to the remainder of the protocol stack. Protocol boosters use this leeway to adapt processing to local conditions, thereby improving performance. For example, video transmitted over a lossy link may be augmented or "boosted" by a forward error correction (FEC) module.

The emphasis of the protocol booster effort is on the performance increases that are attainable by adapting network processing to both the application and environment. Little work has been aimed at understanding how to adapt network processing in a generic manner (that does not rely on prior knowledge of the topology or application). An ANTS active network can realize the same gains with a similar pattern of organization, and in addition provides a programming model for specifying how the network should be adapted.

6.1.3 PIP, SPip and SPIP

Instead of using layering as a composition mechanism, PIP and its variants [Francis, 1993, Francis and Gondivan, 1994] exploit the flexibility of source routing. Essentially this requires an expanded packet header and a slightly different forwarding mechanism that tracks the currently active destination in the list of source routes.

PIP was proposed as a candidate during the design of IPv6 because source routing is able to provide the semantics of a larger address space by using addresses hierarchically. The same mechanism is able to support a variety of routing services, including mobility, core-based trees and provider selection.

Experience with PIP provides at least two useful comments in the context of ANTS. First, the same source routing strategy can be expressed in the capsule programming model and so used to implement new services; ANTS can in addition implement network processing, such as conditional discard, that cannot be expressed in PIP. Second, the assumption that the DNS provides information on not only the address of a host, but how to reach it, is implicit in the design of PIP [Francis, 1994]. This observation can be applied to ANTS in the sense that DNS information about what service to use to contact a host would be valuable for service introduction.

6.2 Programmable Networks

Several programmable network efforts, described below, allow the behavior of the network to be upgraded to some extent. These systems are typically intended for use by network administrators or third party developers, and the flexibility they provide tends to be used in a relatively static manner that does not depend on mobile code.

6.2.1 Programmable Routers

Increasingly, network devices such as firewalls, routers and remote access servers are becoming programmable. Several systems illustrate this trend:

- Cisco's IOS (Internet Operating System) standardizes the developer API and feature set of their routers, and so provides the basis for software releases that include intelligent network services[2].
- Microsoft's Routing and Remote Access (RRAS) Service[3] allows third parties to develop internetworking products using Windows NT servers.
- NetBoost recently announced a programmable network application engine[4] with an open API and hardware support for packet processing operations. It is intended to allow third parties to develop network services such as virus scanning.

Compared with ANTS, these products provide pointwise service upgrades, but are not effective at managing services over a region of the network. One interesting aspect is the decomposition of packet processing tasks into hardware and software, since the systems demonstrate different levels of programmability and performance that are practical for different purposes.

6.2.2 Open Signaling

In the telecommunications community, the Advanced Intelligent Network (AIN) has enabled the relatively rapid introduction of value-added services, such as 1-800 numbers and call forwarding, by standardizing interfaces to switches and other network equipment. Research in the Open Signaling community is aimed at pushing further in this direction by standardizing a programmable interface to the network control plane. For example, see "The Tempest" [van der Merwe *et al.*, 1998].

These architectures are specialized to telecommunications tasks, in which there is a strong separation between signaling and data transfer tasks, and it is not clear how to best apply them to the concept of an active network based on the Internet architecture.

6.3 Active Networks

There are a significant number of recent or ongoing research efforts that are exploring active network approaches. Here, I describe those relevant to network architecture, rather

[2]See http://www.cisco.com/warp/public/732/ for product information.
[3]*Routing and Remote Access Service for Windows NT Server*, MSDN Online Library
[4]*A New Breed*, White Paper, http://www.netboost.com

than those that explore applications or technologies to aid in the construction of an active network.

For the most part, these projects demonstrate different approaches than ANTS, with different strengths and weaknesses. ANTS is one of the more aggressive of these systems in that it targets a relatively wide range of applications.

I have organized the projects in roughly chronological order, beginning with two projects the pre-date that active network approach but are clearly similar in spirit.

6.3.1 Softnet

The seminal programmable network is perhaps Softnet [Zander and Forchheimer, 1983], an experimental, distributed packet radio network constructed in Sweden in the early 1980s. The goal of Softnet was to allow users to define their own high level services (such as datagrams, virtual calls, file transfers and mailboxes) as well as to allow changes at the lower level of link access protocols. Nodes of the network were dual processor (6809) systems, with one processor dedicated to the link, and the other to user tasks. Packets in Softnet are considered to be programs of a network language. They are written in a multi-threaded dialect of FORTH, and are interpreted at nodes as soon as they arrive. Each packet is run concurrently in a separate thread to prevent unwanted interactions between users. A standard set of functions allows control of the node hardware, allowing packets to retransmit themselves to other nodes, store programs in remote nodes, and synchronize with other packets.

In terms of this work, Softnet is an intriguing example of a real programmable network, and one with an architecture that is quite similar to the capsule approach. Softnet inspired a user community and workshops, but unfortunately fell into disuse with little known about its successes, failures and lessons for future programmable networks. It is tempting to speculate that it was not more widely adopted because of difficulties with safety and efficiency (given the state of active technologies in the early 1980s) and with reliability (given the many engineering guidelines such as the end-to-end argument that have since emerged).

6.3.2 Messengers

The messenger paradigm for communication [Tschudin, 1993] replaces the exchange of packets that are interpreted according to a fixed protocol with the exchange of messages that instruct the receiving host how to proceed. The intent is to investigate an alternative structuring of communicating systems such as networking protocols, distributed operating systems, and intelligent agents.

In a prototype system, messengers are written in a postscript-like language called m0 (developed for the purpose) and passed across unreliable channels (framed by Ethernet or UDP) [Tschudin, 1994]. On reception, messengers are interpreted to determine not only their immediate processing, but also the continuing actions that constitute the remote portion of a protocol. Each messenger is interpreted concurrently in a separate thread. It is run in a standard environment with operators to access shared dictionaries, send new messengers or create local processes, and synchronize with other messengers (via a process queue abstraction).

As with Softnet, the messenger system is similar to the capsule approach. It demonstrates that communications protocols can be successfully structured as migrating computations by implementing a sliding window protocol. Further, it shows that with care the resulting programs can be compact enough to be transferred as datagrams. For example, a network discovery program is given in a terse 200 bytes. Unlike the system proposed here, performance was not a goal of the prototype system, and so the impact of messenger-style protocol implementations on overall application performance is not known. The overhead of code distribution is also unknown, though provision is made to refer to other programs installed at hosts rather than to always carry complete programs.

6.3.3 ACTIVE IP

The ACTIVE IP system was a proof-of-concept active network intended mainly for network probing and discovery tasks. It allowed network users to tag their IP packets with fragments of Tcl code. This code was carried by using the existing IP options mechanism. In this manner, packet processing at enabled nodes was customized in terms of a set of available primitives. The system was implemented by extending and embedding a Tcl interpreter in the IP layer of the Linux operating system kernel. Further details can be found in [Wetherall and Tennenhouse, 1996].

As a predecessor to ANTS, ACTIVE IP demonstrated the feasibility of an active network for a restricted case. It showed that interesting and useful programs (such as the existing options tasks and network discovery programs) could be composed from a relatively small set of node primitives. It showed that these programs could be compactly expressed, such that it is reasonable to transfer them along with packet data.

6.3.4 Switchware and PLAN

At the University of Pennsylvania and Bellcore, the Switchware project [Alexander *et al.*, 1998] is developing a programmable switch approach that allows digitally signed type-checked modules to be loaded into a network node. Out-of-band program loading is used to support value-added services, such as network striping. This approach builds on the Advanced Intelligent Network (AIN) concept of the telecommunications industry, in which the implementation of value added services is separated from that of switching by moving the service control functions to adjunct processors. In Switchware, formal methods are applied to assure the security of the network; one goal is the identification of security properties of the underlying infrastructure for which theorems can be proved. A prototype "active bridge" demonstrates how network elements can be upgraded on-the-fly [Alexander *et al.*, 1997].

A complementary part of the effort is the development of PLAN [Hicks *et al.*, 1998], a programming language for active networks. PLAN is intended to be compact, so that small forwarding programs can be carried directly in each packet. These programs can refer to node resident code for privileged operations or common-case processing that is too large to carry directly. PLAN is also designed to facilitate the safe operation of the network.

Because the language is restricted, certain program forms, such as those that loop forever (either locally or across nodes) cannot be expressed.

6.3.5 Netscript

The Netscript project [Yemini and da Silva, 1996] at Columbia University is focusing on network management. It is designed to support new routing, packet analysis, signaling and management tasks. Netscript consists of a dataflow style programming language for scripting agents that process packet streams, along with a Virtual Network Engine, or execution environment within which agents are run at nodes. An overall program may consist of many agents that are distributed across nodes. Nodes themselves are connected via the underlying physical network by an overlay of Virtual Links. The system allows new agents to be dynamically deployed and configured using a delegation model. The goal is to enable the programming of remote nodes, including intermediate systems, as easily and quickly as end-systems.

Compared with ANTS, the Netscript approach is aimed at network managers rather than users, and hence the establishment of more heavyweight network processing configurations.

6.3.6 SmartPackets and Sprocket

At BBN, the SmartPackets project[5] has also applied active network techniques to network management tasks, but for signaling and control rather than to introduce new functionality. SmartPackets carry short programs written in a specially-designed RISC-like language, Sprocket, that is compact and targeted at network management tasks. For example, MIB names are represented compactly. These programs are then authenticated and interpreted at each router they encounter to carry out SNMP-like management tasks. The basis for organizing the system in this way is the same as that of programmable RPC: that performance increases stem from the collapse of multiple round-trip queries into a single program execution at each node.

6.3.7 Active Services

The active (distributed) services approach is based on the availability of programmable end-systems within the network infrastructure [Govindan *et al.*, 1998]. These end-systems can then be organized into service-specific overlays. Amir provides an example of an active service framework, AS1, targeted at media processing tasks such as transcoding [Amir *et al.*, 1998].

One of the main motivations of active services is to provide active network-like functionality without requiring modifications to routers. As part of the design of ANTS, I have argued that the distinction between end-systems and routers is not substantial in this context. Instead, the significant design constraints (common to both ANTS and active services) are that programmable nodes exist within the network infrastructure (and so are in a strategic

[5] *Smart Packets for Active Networks*, B. Schwartz et al., January 1998. From http://www.net-tech.bbn.com/smtpkts/smart.ps.gz.

location to support new services) but that not all nodes need be programmable (and so there is no performance penalty imposed on all routers).

A more substantial issue raised by active services is how to accommodate computationally-intensive processing, such as video transcoding and Web caching, within the network infrastructure. A reasonable view is that tasks fall into at least two domains: lightweight processing akin to ANTS forwarding routines; and heavyweight processing akin to AS1 data manipulations. Support for both domains is required and can work in synergy with each other. This view is supported by the exploration of services such as Web caching and the evaluation of the ANTS architecture. ANTS services act as the network "glue" that knits together processing that occurs at programmable end-systems into a useful overall pattern, and programmable end-systems extend the range of tasks that can be accomplished with ANTS services.

Chapter 7

Conclusions

In this dissertation I have: presented a detailed design of the ANTS active network architecture; described a Java-based implementation called the ANTS toolkit; demonstrated how the toolkit can be used to introduce new services; and evaluated the architecture by comparing it with the characteristics of the Internet.

ANTS is fundamentally different from conventional networks. It is an *active network* in which users can introduce new services using the *capsule* programming model. Capsules are special packets that combine data with a customized forwarding program. The type of forwarding is selected by end user software when a capsule is injected into the network. The corresponding program is transferred along with the data using mobile code techniques as the capsule passes through the network. It is executed at programmable routers called *active nodes* along the forwarding path. This model introduces flexibility at the router level directly, rather than in terms of an end-system overlay. It allows topology, loss and load information to be used to construct a variety of new routing, flow setup, congestion handling, and measurement services. The focus of my research has been to evaluate how much flexibility this architecture can provide in practice, while at the same time adequately addressing the performance and security concerns that it raises.

I begin this chapter with the overall conclusions that I draw from this research, follow with a description of other specific contributions, and close the dissertation with a discussion of open questions for ongoing research into active networks.

7.1 Conclusions

The main finding of this research is to validate the hypothesis that an active network can introduce new services rapidly and automatically, and at a reasonable cost compared to a conventional network. Several observations lead to this conclusion:

- *Flexibility.* The construction of new services in Chapter 4 and design patterns in Chapter 5 demonstrate that the capsule and active node model can express a variety of new and useful services. This is despite the fact that the programming model is greatly restricted when compared to a general purpose distributed programming system.

- *Rapid Deployment.* By design, the ANTS includes a code distribution system that deploys service code automatically. A new service can be developed rapidly, since only those end-systems that use it plus the trusted authority that certifies it need agree on the definition. The net result is rapid and automatic service deployment.

- *Performance.* Measurements of the ANTS toolkit provide evidence that the performance overheads of using a newly deployed service are low. Compared to IP, the additional processing steps required for common case forwarding are modest. Even the user-level Java prototype has sufficient capacity to be used as an active node at a number of locations within the network today, namely those at which routers are attached to wireless links, access links through T1 (1.5 Mbps) rates, and other bandwidth-limited links.

- *Security.* An analysis of protection and resource management mechanisms shows that an ANTS network has security that is comparable to the Internet today. Most threats can be handled by active nodes locally and without trusting external parties, with the exception that the global resource usage of a service must be certified at acceptable by a trusted authority.

More broadly, I conclude that the active network approach shows great promise as a means of lowering the barriers to innovation, stimulating experimentation, and otherwise hastening the arrival of new services. It seems inevitable that network equipment and networks as a whole will become increasingly malleable as rapid adaptation to changing requirements becomes more and more important. Active networks offer a practical and systematic means of providing the required flexibility. They have much to offer to the evolution of a large network such as the Internet that currently evolves in a manner that is greatly constrained by concerns for incremental deployment and backwards-compatibility.

7.2 Contributions

As well at this overall conclusion, the research presented in this dissertation makes a number of specific contributions that revolve around the design and evaluation of ANTS:

- The definition of an *active network* approach that is based on *capsules*, along with a specific architecture, ANTS, that embodies this approach.

- An analysis of ANTS architecture in terms of the combined levels of flexibility, performance and security that it is able to achieve.

- An understanding of how the capsule programming model can be used to express a variety of network services.

- A *demand pull* code distribution protocol that automatically deploys service code along network paths yet is compatible with performance and security goals.

- A *fingerprint-based* protection model that efficiently separates the state of different services within the network in a manner that is equivalent to conventional protocols, despite the use of untrusted code.

- A choice of active node API that allows the capsule programming model to be realized efficiently as *extensible packet forwarding*, such that forwarding requires few processing steps beyond those needed for IP.

The following sections expand on these contributions.

The Active Network Approach

The active network approach and the concept of capsules were first described in [Tennenhouse and Wetherall, 1996]. In this dissertation, I have defined these concepts in full by presenting ANTS, a concrete active network architecture that is based on capsules. In doing so, I have refined the notion of a programmable network to suit the evolution of communication tasks (rather than general-purpose distributed computation) with optimizations such as code caching and support for a heterogeneous mix of nodes. I have also enumerated the significant differences between an active network and a conventional one in terms of network services and their evolution:

- A fixed model of computation rather than a fixed service.
- The use of many special-purpose services rather than one general-purpose service.
- An emphasis on innovation rather than backwards-compatibility.

These differences amount to a new way of thinking about network evolution that has ramifications for the Internet of tomorrow. Two points illustrate this new outlook:

- It is possible to construct services that are tailored to a specific application. This allows a portion of the application to be "pushed" into the network infrastructure to gain the performance advantages of code-shipping. Internet protocols do not allow the use of computation within the network.
- Since the processing capsules can receive within the network is made explicit, network services evolve in a clean and systematic manner. This contrasts with IP, which is increasingly treated as a packet format (rather than a protocol definition) in the sense that any processing that is compatible with its existing operation can be implemented. This sometimes has surprising results, for example, in the case of NATs[1].

Analysis of the ANTS Architecture

To evaluate ANTS, I compared its flexibility, performance and security properties to those of the Internet. The same framework could be used to assess other active network architectures. Based on experience constructing new services and measurements of the ANTS toolkit, I was able to show that ANTS greatly increases the flexibility with which new services can be introduced, while adding a modest performance overhead to software-based routers and raising few additional security concerns.

More important than the point in the design space demonstrated by ANTS, however, is the set of tradeoffs that were made as part of the design. A key difficulty that an active network

[1]Internet Draft, *Architectural Implications of NATs*, T. Hain, November 1998.

must tackle is to provide a workable set of tradeoffs that address the tension among competing properties. Individually, each property can be improved by altering the design of the network architecture: flexibility would be improved by lifting the restrictions that separate capsule activity within the network; performance would be improved by eliminating active node capabilities such as the soft-store; and security would be improved by authenticating each packet against a user at each node. Such design changes would, of course, impact the other characteristics. By using an evaluation framework that considered all of these qualities, I was able to expose worthwhile tradeoffs between them.

Programming with Capsules

At the outset of this research, it was unclear whether the capsule model would be able to express useful services in a manner compatible with the operation of the network layer as well as other constraints, such as a mix of active and non-active nodes. For example, Partridge claimed that the end-to-end-argument would preclude the use of the active network approach at the IP layer [Partridge *et al.*, 1998]. This dissertation has shown that claim to be untrue.

The multicast, Web caching and path MTU discovery services provide examples of capsule programs that are compatible with: packet loss, loss of data kept in the soft-store, code size limits, segmented application data, limited processing time, node and link failures, changes in network paths, and other constraints imposed by the network layer. The programming guidelines and design patterns describe network service constructs and program forms that avoid design pitfalls. They demonstrate how to express a variety of custom routing, intercept, and merge services. This contribution is relevant to any capsule-based active network.

Experimentation with services has also helped to clarify the role of network-resident processing. Services in an active network can be application-specific, that is, they can be considered part of the application that is running within the shared network infrastructure. To design applications in this manner, a useful view is that of network processing as an optimization of end-to-end processing. This is compatible with reliability concerns and partial deployment. For example, in an online auction service, Legedza delegated bid rejection processing — but not bid acceptance processing — to network nodes [Wetherall *et al.*, 1998]. Bid rejection within the network is an optimization because if it does not occur within the network then it can be handled at the edge of the network. On the other hand, bid acceptance cannot be modeled as an optimization because if it occurs within the network it must be reliably signaled to the edge of the network.

Demand Pull Code Distribution

ANTS includes a code distribution system that automatically transfers forwarding routines to nodes of the network as they are needed and along the path that the capsule follows. This kind of deployment makes it easy to use a new service, and is fundamentally different than that of the Internet.

In the Internet, router upgrades on all but the smallest scale are effectively constrained to occur pointwise and in a backwardly-compatible fashion. This is because the functionality is deployed manually, and the period of overlap between old and new can be considerable. In contrast, by using mobile code and automatic deployment mechanisms, ANTS is able to rapidly deploy a new service across the wide area without concern for backwards-compatibility.

The design of the demand pull protocol is also compatible with performance and security goals. Given services that are amenable to caching and service code that is not large in terms of the packet size, code distribution requires little bandwidth and adds a delay that is on the order of transit time variations measured in the Internet today. The fingerprint-based capsule types and the incremental nature of the load path prevent code spoofing and provide a basis for localizing denial-of-service attacks.

Fingerprint-Based Protection

ANTS provides a novel protection model that separates the behavior of services within the network. This model balances the concerns of efficiency and security. Consider the alternatives. On one hand, it is possible to design an active network without a protection mechanism by making the network programmer responsible for service separation. This adds no runtime overhead to implement, but guarantees no isolation since it relies on the co-operation of network programmers. On the other hand, it is possible to use authentication techniques to separate the state of different users. This prevents one user from interfering with another, but is expensive to implement compared to IP forwarding since it requires a cryptographic operation for every packet.

Instead, ANTS leverages the fingerprint-based capsule types for protection. By using these types as service names, and assuming the fingerprint function is one-way, the state of different services is guaranteed to be separated. This mechanism is inexpensive to implement, since most of the work (the verification of code against fingerprint) is performed at code load-time rather than capsule run-time. Compared with the extremes above, this mechanism is able to provide both a reasonable protection model (per service rather than per user separation) and an efficient implementation (load-time rather than run-time work).

Node API and Extensible Packet Forwarding

In ANTS, the categories of node API calls are chosen to limit the impact of new services on active node processing, while allowing as many services to be expressed as possible. The result is that the capsule programming model can be realized efficiently as a simple extensible packet forwarding engine: capsules are demultiplexed to their handler, the handler is run, and the process repeats with the next capsule.

With the capsule model, the main source of processing over and above that required of an IP router is likely to stem from calls to the node API: code distribution costs can be amortized over a large number of capsules; service routines themselves are expected to consist mainly of programming language "glue" that connects calls to the node API; safe execution techniques add modest overhead; and other per packet and periodic tasks are inexpensive. The node API must therefore be designed carefully.

The selection of calls available in ANTS is based on experience constructing and using new services with a prior system [Wetherall and Tennenhouse, 1996] and early versions of the toolkit. The current set of calls is weaker than I originally anticipated but still able to form the basis of many useful services.

All node API operations are lightweight in that they run to completion locally, without requiring external information or resources and without blocking due to resource contention. For example, a major design simplification is the use of a soft (rather than persistent) store at nodes. I argue that this does not complicate service design because the need to accommodate partial node and link failures and changing routes naturally extends to soft-state. Further, the node API contains no timer functions, as might be useful for periodic activity such as refreshing routing data. Instead, it proved possible to use capsule events delivered from outside the network layer as a "clock," resulting in a simpler node model that supports approximately the same kind of services. Similarly, no capsule rendezvous primitives are provided, since packet events coupled with the ability to place capsules in the soft-store allow many of the same services to be expressed.

7.3 Open Questions

As one of the first theses in the area of active networks, this work opens a line of inquiry. The discussion below outlines some of the problems that must be tackled in order to make programmable networks a reality. I focus on open research issues that must be resolved, rather than the practical steps that are necessary to deploy a large-scale active network in the Internet.

Managing Resources Across Nodes

The design of the ANTS architecture provides a basis for securing the resources at a single active node against each execution of untrusted service code. The management of resources across many nodes, however, is not secured in this manner. I have provided a qualitative argument that per capsule resource limits accomplish little more than to ensure that infinite loops are broken. Global resource management in ANTS is thus ultimately controlled through the certification of services by a trusted authority, abstracted as a node security policy. This is roughly equivalent to IETF-based standardization today (except that an implementation is approved, rather than a specification). Like standardization, it suffers from the drawbacks of vulnerability to accidental errors and centralized authorities.

A more powerful model would be to automatically ensure that an untrusted service makes use of global resources in a reasonable way. This is a difficult task. I have begun it by postulating a definition of "reasonable" in which capsule activity is not localized in any region of the network. That is, one capsule can visit each node no more than k times, assuming there are no loops in the standard routing tables, and where k is a small number such as one or two. I refer to protocols that satisfy this definition for some value of k as $k - \mathit{bound}$. This definition readily allows multicast services and accommodates Internet protocols, but disallows loops and flooding behavior such as ICMP smurf attacks.

Further research is needed on how to check whether new services satisfy this definition. I suggest that it may be practical to statically check the form of service programs to see that they are safe. For example, routines that forward to a fixed address and do not send more than one capsule are clearly $1 - bound$, and this program form can easily be recognized. Though it may sound restrictive, this form can express selective discard, congestion notification and network merging services.

Application Autonomy with Network Transparency

The authors of the end-to-end argument suggest in their commentary on the end-to-end argument in light of active networks [Reed *et al.*, 1998] that the argument implies two complementary goals:

- *application autonomy*: higher layers, more specific to an application, are expected to organize lower-level network resources to achieve application-specific design goals efficiently.
- *network transparency*: lower layers, which support many independent applications, should provide only resources of broad utility across applications, while providing to applications a usable means for effective sharing of resources and resolution of resource conflicts.

Clearly, an active network increases application autonomy, since applications can control the processing performed within the network on their behalf. Reed, Saltzer and Clark raise the concern that, if not well designed, an active network may decrease network transparency, since services may interfere with each other in ways that are not readily predictable.

Central to the ability to program network services is the ability to model in a simple manner the expected network response to the new service. This allows a new service to be designed without being dependent on the other services that are running simultaneously with it in the network infrastructure.

The main respect in which the ANTS architecture does not facilitate a simple model compared to the Internet is the soft-store. Loss of soft-state can have a significant effect on the performance of some services, for example, it can prevent capsules from reaching their destination. The current soft-store does not facilitate a simple model, since the information used by one service is affected by the unknown loads of other services, and so it is difficult to determine when to refresh soft-state. An interesting approach might be to tie storage guarantees to the packet rate, for example, one byte per second per packet. This assumes that services that are making greater use of the network (as measured in terms of packets) deserve a larger share of the soft-store. Unfortunately, since a storage guarantee is only useful if it persists for a long time compared to the packet rate this model may require large stores, for example, for routers capable of 10^6 packets per second, each byte-second of guarantee requires 1 MB of storage.

An interesting counterpoint to this discussion is bandwidth consumption in the Internet. Arguably the complexity of congestion control stems from the fact that there is no closed model for sustainable bandwidth along network paths, thus requiring applications using the network to adapt. While the same techniques may be useful for adapting to the available soft-storage at network nodes, a network in which services must adapt to the availability of multiple resources may be too complicated to be useful.

Composition with Protection

As a final issue, there is a tension between the protection of services and their composition. In the ANTS architecture, a service protection model is provided largely at the expense of support for service composition.

The ANTS protection mechanism separates the code and state of different protocols. At the edges of the network, different code groups can be combined into a protocol and then capsules of that protocol can be sent into the network. This supports code modularity and reuse. Within the network, however, the state of different protocols is separated, even if the same code group belongs to both protocols. While the soft-store allows information to be marked world-readable, this separation prevents different sets of code groups from sharing state securely inside the network, and effectively prevents the composition of services.

An alternative design might be for the soft-store to support semantics that are closer to that of a filesystem. By using access control lists, different services could then share state within the network in whatever pattern is required. This model requires that authentication capabilities be added to the active node.

Bibliography

[Adl-Tabatabai *et al.*, 1996] A. Adl-Tabatabai, G. Langdale, S. Lucco, and R. Wahbe. Efficient and Language-Independent Mobile Code. In *PLDI'96*. ACM, May 1996.

[Alexander *et al.*, 1997] D. S. Alexander, M. Shaw, S. Nettles, and J. Smith. Active Bridging. In *SIGCOMM'97*, 1997.

[Alexander *et al.*, 1998] 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 Magazine*, 12(3), May/June 1998.

[Amir *et al.*, 1998] E. Amir, S. McCanne, and R. Katz. An Active Service Framework and its Application to Real-time Multimedia Transcoding. In *SIGCOMM'98*, Vancouver, Canada, September 1998. ACM.

[Atkinson, 1995] R. Atkinson. Security Architecture for the Internet Protocol. Request for Comments 1825, August 1995.

[Bailey *et al.*, 1994] M. L. Bailey, B. Gopal, M. A. Pagels, L. L. Peterson, and P. Sarkar. Pathfinder: A pattern-based packet classifier. In *Proc. of the First Symp. on Operating System Design and Implementation*, pages 243–253, November 1994.

[Bajaj *et al.*, 1998] Sandeep Bajaj, Lee Breslau, and Scott Shenker. Uniform versus Priority Dropping for Layered Video. In *SIGCOMM'98*, Vancouver, BC, September 1998.

[Baker, 1995] F. Baker. Requirements for IP Version 4 Routers. Request for Comments 1812, June 1995.

[Balakrishnan *et al.*, 1996] H. Balakrishnan, V. N. Padmanabhan, S. Seshan, and R. H. Katz. A Comparison of Mechanisms for Improving TCP Performance over Wireless Links. In *SIGCOMM '96*, pages 256–269, Stanford, CA, August 1996. ACM.

[Ballardie *et al.*, 1993] A. Ballardie, P. Francis, and J. Crowcroft. Core based Trees. In *SIGCOMM'93*, San Francisco, CA, 1993.

[Bhattacharjee *et al.*, 1996] Samrat Bhattacharjee, Kenneth L. Calvert, and Ellen W. Zegura. On Active Networking and Congestion. Technical Report CIT-CC-96/02, College of Computing, Georgia Institute of Technology, Spring 1996.

[Bhattacharjee *et al.*, 1998] Samrat Bhattacharjee, Ken Calvert, and Ellen W. Zegura. Self-Organizing Wide-Area Network Caches. In *INFOCOM'98*, San Francisco, CA, April 1998.

[Braden *et al.*, 1997] R. Braden, L. Zhang, S. Berson, S. Herzog, and S. Jamin. Resource ReSerVation Protocol (RSVP) – Version 1 Functional Specification. Request for Comments 2205, September 1997.

[Cejtin *et al.*, 1995] Henry Cejtin, Suresh Jagannathan, and Richard Kelsey. Higher Order Distributed Objects. *Transactions on Programming Languages and Systems*, September 1995.

[Chen *et al.*, 1998] J. Chen, P. Druschel, and D. Subramanian. An Efficient Multipath Forwarding Method. In *INFOCOM'98*, April 1998.

[Clark and Tennenhouse, 1990] D. D. Clark and D. L. Tennenhouse. Architectural Considerations for a New Generation of Protocols. In *SIGCOMM '90*, Philadelphia, PA, September 1990. ACM.

[Clark, 1988] David D. Clark. The Design Philosophy of the DARPA Internet Protocols. In *SIGCOMM '88*, pages 106–114, Stanford, CA, August 1988. ACM.

[Conta and Deering, 1995] A. Conta and S. Deering. Internet control message protocol (icmpv6) for the internet protocol version 6 (ipv6). Request for Comments 1885, December 1995.

[CRA, 1997] CRA. Research challenges for the next generation internet. Report from the Workshop on Research Directions for the Next Generation Internet, May 1997.

[Decasper and Plattner, 1998] Dan Decasper and Bernhard Plattner. DAN: Distributed Code Caching for Active Networks. In *INFOCOM'98*, San Francisco, CA, April 1998.

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

[Deering *et al.*, 1994] S. Deering, D. Estrin, D. Farinacci, V. Jacobson, C.-G. Liu, and L. Wei. An architecture for wide-area multicast routing. In *Proceedings of SIGCOMM'94*, 1994.

[Deering *et al.*, 1998] S. Deering, D. Estrin, D. Farinacci, A. Helmy, D. Thaler, M. Handley, V. Jacobson, C. Liu, P. Sharma, and L. Wei. Protocol independent multicast-sparse mode (pim-sm): Protocol specification. Request For Comments 2362, June 1998.

[Deering, 1989] S. E. Deering. Host Extensions for IP multicasting. Request For Comments 1112, August 1989.

[Deutsch and Grant, 1971] P. Deutsch and C. A. Grant. A Flexible Measurement Tool for Software Systems. In *Information Processing*, pages 320–326, 1971.

[Doar and Leslie, 1993] M. Doar and I. Leslie. How Bad is Naive Multicast Routing? In *INFOCOM'93*, 1993.

[Dobbertin, 1996] H. Dobbertin. The Status of MD5 After a Recent Attack. *RSA Laboratories CryptoBytes*, 2(2), 1996.

[Egevang and Francis, 1994] K. Egevang and P. Francis. The IP Network Address Translator (NAT). Request For Comments 1631, May 1994.

[Engler and Kaashoek, 1996] Dawson R. Engler and M. Frans Kaashoek. DPF: Fast, Flexible Message Demultiplexing using Dynamic Code Generation. In *SIGCOMM '96*, Stanford, CA, August 1996. ACM.

[Feldmeier *et al.*, 1998] D. C. Feldmeier, A. J. McAuley, J. M. Smith, D. S. Bakin, W. S. Marcus, and T. M. Raleigh. Protocol Boosters. *IEEE Journal on Selected Areas of Communication*, 16(3), April 1998.

[Fenner, 1997] W. Fenner. Internet Group Management Protocol, Version 2. Request For Comments 2236, November 1997.

[Floyd and Jacobson, 1993] S. Floyd and V. Jacobson. Random Early Detection gateways for Congestion Avoidance. *IEEE/ACM Transactions on Networking*, 1(4):397–413, August 1993.

[Floyd *et al.*, 1997] S. Floyd, V. Jacobson, C. Liu, S. McCanne, and L. Zhang. A reliable multicast framework for light-weight sessions and application level framing. *IEEE/ACM Transactions on Networking*, 5(6):784–803, December 1997.

[Floyd, 1994] S. Floyd. TCP and Explicit Congestion Notification. *ACM Computer Communications Review*, 24(5):10–23, October 1994.

[Francis and Gondivan, 1994] P. Francis and R. Gondivan. Flexible Routing and Addressing for a Next Generation IP. In *SIGCOMM'94*, London, UK, September 1994.

[Francis, 1993] Paul Francis. A Near-Term Architecture for Deploying PIP. *IEEE Network*, 7(3):30–37, May 1993.

[Francis, 1994] Paul Francis. *Addressing in Internetwork Protocols.* PhD thesis, University College London, September 1994.

[Gong and Schemers, 1998] L. Gong and R. Schemers. Implementing Protection Domains in the Java Development Kit 1.2. In *Internet Society Symposium on Network and Distributed System Security*, pages 125–134, San Diego, CA, March 1998.

[Gong *et al.*, 1997] L. Gong, M. Mueller, H. Prafullchandra, and R. Schemers. Going Beyond the Sandbox: An Overview of the New Security Architecture in the Java Development Kit 1.2. In *Symp. on Internet Technologies and Systems*, pages 103–112, Monterey, CA, December 1997. USENOX.

[Gosling, 1995] J. Gosling. Java Intermediate Bytecodes. In *SIGPLAN Workshop on Intermediate Representations (IR95)*, pages 111–118, San Francisco, CA, January 1995. ACM. Appears in SIGPLAN Notices, 30, 3 (March 1995).

[Govindan *et al.*, 1998] R. Govindan, C. Alaettinoglu, and D. Estrin. A framework for active distributed services. Technical Report 98-669, USC, January 1998.

[Hawblitzel *et al.*, 1998] Chris Hawblitzel, Chi-Chao Chang, Grzegorz Czajkowski, Deyu Hu, and Thorsten von Eicken. Implementing Multiple Protection Domains in Java . In *USENIX'98*, New Orleans, LA, June 1998.

[Hedrick, 1988] C. Hedrick. Routing Information Protocol. Request for Comments 1058, June 1988.

[Hicks *et al.*, 1998] Michael Hicks, Pankaj Kakkar, Jonathan T. Moore, Carl A. Gunter, and Scott Nettles. PLAN: A Packet Language for Active Networks. In *International Conference on Functional Programming (ICFP'98)*, 1998.

[Holbrook *et al.*, 1995] Hugh Holbrook, Sandeep Singhal, and David Cheriton. Log-Based Receiver-Reliable Multicast for Distributed Interactive Simulation. In *SIGCOMM'95*. ACM, 1995.

[Hsieh *et al.*, 1996] Wilson Hsieh, Marc Fiuczynski, Charles Garrett, Stefan Savage, David Becker, and Brian Bershad. Language Support for Extensible Operating Systems. In *Workshop on Compiler Support for System Software*, February 1996.

[Hutchinson and Peterson, 1991] Norman C. Hutchinson and Larry L. Peterson. The x-Kernel: An Architecture for Implementing Network Protocols. *IEEE Transactions on Software Engineering*, 17(1):64–76, January 1991.

[Jacobson *et al.*, 1992] V. Jacobson, R. Braden, and D. Borman. TCP Extensions for High Performance. Request for Comments 1323, May 1992.

[Johnson, 1998] E. Johnson. A Protocol for Network Level Caching. M.Eng Thesis, Massachusetts Institute of Technology, May 1998.

[Kent and Mogul, 1987] C. Kent and J. Mogul. Fragmentation Considered Harmful. In *SIGCOMM '87*, Stowe, VT, August 1987. ACM.

[Koprowski, 1998] G. Koprowski. Emerging Uncertainty Over IPv6. *IEEE Computer*, 31(11), November 1998.

[Lakshman and Stiliadis, 1998] T.V. Lakshman and D. Stiliadis. High-Speed Policy-based Packet Forwarding Using Efficient Multi-dimensional Range Matching. In *SIGCOMM'98*, Vancouver, BC, September 1998.

[Legedza and Guttag, 1998] Ulana Legedza and John Guttag. Using Network-level Support to Improve Cache Routing. In *3rd International WWW Caching Workshop*, Manchester, UK, June 1998.

[Legedza *et al.*, 1998] Ulana Legedza, David Wetherall, and John Guttag. Improving the Performance of Distributed Applications Using Active Networks. In *INFOCOM '98*, 1998.

[Lehman *et al.*, 1998] Li-Wei Lehman, Stephen Garland, and David Tennenhouse. Active Reliable Multicast. In *INFOCOM'98*, 1998.

[Leiner *et al.*, 1997] B. Leiner, V. Cerf, D. Clark, R. Kahn, L. Kleinrock, D. Lynch, J. Postel, L. Robberts, and S. Wolff. The Past and Future History of the Internet. *Communications of the ACM*, 40(2), February 1997.

[Levine and Garcia-Luna-Aceves, 1997] B. N. Levine and J.J. Garcia-Luna-Aceves. Improving internet multicast with routing labels. In *IEEE International Conference on Network Protocols (ICNP'97)*, pages 241–50, October 1997.

[Li and Cheriton, 1998] D. Li and D. Cheriton. Oters: Exploiting the routing topology for high-performance reliable multicast. In *IEEE International Conference on Network Protocols (ICNP'98)*, October 1998.

[Mathis *et al.*, 1996] M. Mathis, J. Mahdavi, S. Floyd, and A. Romanow. TCP Selective Acknowledgement Options. Request for Comments 2018, October 1996.

[McCann *et al.*, 1996] J. McCann, S. Deering, and J. Mogul. Path MTU Discovery for IP version 6. Request for Comments 1981, August 1996.

[McCanne *et al.*, 1996] Steven McCanne, Van Jacobson, and Martin Vetterli. Receiver-driven Layered Multicast. In *SIGCOMM'96*, pages 117–130, Stanford, CA, August 1996.

[Mogul and Deering, 1990] J. Mogul and S. Deering. Path MTU Discovery. Request for Comments 1191, November 1990.

[Necula and Lee, 1996] G. Necula and P. Lee. Safe Kernel Extensions Without Run-Time Checking. In *2nd Symposium on Operating System Design and Implementation*, pages 229–243, Seattle, WA, October 1996. USENIX.

[Necula and Lee, 1998] G. Necula and P. Lee. The Design and Implementation of a Certifying Compiler. In *PLDI'98*, Montreal, Canada, June 1998. ACM.

[Nygren, 1999] E. Nygren. PAN: A High-Performance Active Network Node Supporting Multiple Code Systems. In *OPENARCH'99*, New York, NY, March 1999. IEEE.

[O'Malley and Peterson, 1991] S. O'Malley and L. Peterson. TCP Extensions Considered Harmful. Request For Comments 1263, October 1991.

[O'Malley and Peterson, 1992] S. W. O'Malley and L. L. Peterson. A dynamic network architecture. *ACM Transactions on Computer Systems*, 10(2):110–143, May 1992.

[Papadopoulos *et al.*, 1998] C. Papadopoulos, G. Parulkar, and G. Varghese. An Error Control Scheme for Large-Scale Multicast Applications. In *INFOCOM'98*, San Francisco, CA, April 1998.

[Pardyak and Bershad, 1996] Przemyslaw Pardyak and Brian Bershad. Dynamic Binding for an Extensible System. In *Proc. of the Second USENIX Symposium on Operating Systems Design and Implementation (OSDI'96)*, pages 201–212, Seattle, WA, October 1996.

[Partridge *et al.*, 1993] C. Partridge, T. Mendez, and W. Miliken. Host Anycasting Service. Request For Comments 1546, November 1993.

[Partridge *et al.*, 1998] C. Partridge, T. Strayer, B. Schwartz, and A. Jackson. Commentaries on the Active Networking and End-to-End Arguments. *IEEE Network Magazine*, 12(3):67–69, May/June 1998.

[Partridge, 1992] Craig Partridge. *Late-Binding RPC: A Paradigm for Distributed Computation in a Gigabit Environment.* PhD thesis, Harvard University, 1992.

[Paxon, 1997] V. Paxon. End-to-End Internet Packet Dynamics. In *SIGCOMM'97*, Cannes, France, September 1997.

[Perkins, 1996] C. Perkins. IP Mobility Support. Request For Comments 2002, October 1996.

[Postel, 1981a] J. Postel. Internet Control Message Protocol. Request For Comments 792, September 1981.

[Postel, 1981b] J. Postel. Internet Protocol. Request For Comments 791, September 1981.

[Postel, 1981c] J. Postel. Transmission Control Protocol. Request For Comments 793, September 1981.

[Ramakrishnan and Floyd, 1999] K. Ramakrishnan and S. Floyd. A Proposal to add Explicit Congestion Notification (ECN) to IP. Request For Comments 2481, January 1999 1999.

[Reed *et al.*, 1998] D. P. Reed, J. H. Saltzer, and D. D. Clark. Commentaries on the Active Networking and End-to-End Arguments. *IEEE Network Magazine*, 12(3):69–71, May/June 1998.

[Rivest, 1992] R. Rivest. The MD5 Message-Digest Algorithm. Request For Comments: 1321, April 1992.

[Robshaw, 1996] M. Robshaw. On Recent Results for MD2, MD4, and MD5. *RSA Laboratories Bulletin*, November 1996.

[Saltzer *et al.*, 1984] J. H. Saltzer, D. P. Reed, and D. D. Clark. End-to-End Arguments in System Design. *ACM Transactions on Computer Systems*, 2(4):277–288, 1984.

[Srinivasan *et al.*, 1998] V. Srinivasan, G. Varghese, S. Suri, and M. Waldvogel. Fast Scalable Algorithms for Level Four Switching. In *SIGCOMM'98*, Vancouver, BC, September 1998.

[Stamos and Gifford, 1990] James W. Stamos and David K. Gifford. Remote Evaluation. *ACM Transactions on Programming Languages and Systems*, 12(4):537–565, October 1990.

[Stevens, 1997] W. Stevens. TCP Slow Start, Congestion Avoidance, Fast Retransmit and Fast Recovery Algorithms. Request For Comments 2001, January 1997.

[Tennenhouse and Wetherall, 1996] D. L. Tennenhouse and D. Wetherall. Towards an Active Network Architecture. In *Multimedia Computing and Networking 96*, San Jose, CA, January 1996. A revised version appears in CCR Vol. 26, No. 2 (April 1996).

[Tian and Neufeld, 1998] J. Tian and G. Neufeld. Forwarding State Reduction for Sparse Mode Multicast Communication. In *INFOCOM'98*, San Francisco, CA, April 1998.

[Tschudin, 1993] C. Tschudin. *On the Structuring of Computer Communications*. PhD thesis, Geneva University, 1993.

[Tschudin, 1994] C. Tschudin. An Introduction to the m0 Messenger Language. Technical Report Report No. 86, Geneva University, 1994.

[van der Merwe *et al.*, 1998] J. van der Merwe, S. Rooney, and I. Leslie aned S. Crosby. The Tempest – A Practical Framework for Network Programmability. *IEEE Network Magazine*, 12(3), May/June 1998.

[Van, 1997] Van C. Van. A Defense Against Address Spoofing Using Active Networks. M.Eng Thesis, Massachusetts Institute of Technology, June 1997.

[Wahbe *et al.*, 1993] Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham. Efficient Software-based Fault Isolation. In *14th Symp. on Operating Systems Principles*, Ashville, NC, December 1993. ACM.

[Waitzman *et al.*, 1988] D. Waitzman, C. Partridge, and S. Deeering. Distance Vector Multicast Routing Protocol. Request For Comments 1075, November 1988.

[Wessels and Claffy, 1998] D. Wessels and K. Claffy. Icp and the squid web cache. *IEEE Journal on Selected Areas of Communications*, pages 345–357, April 1998.

[Wetherall and Tennenhouse, 1996] David J. Wetherall and David L. Tennenhouse. The ACTIVE IP Option. In *7th SIGOPS European Workshop*, Connemara, Ireland, September 1996. ACM.

[Wetherall *et al.*, 1998] David Wetherall, John Guttag, and David Tennenhouse. ANTS: A Toolkit for Building and Dynamically Deploying Network Protocols. In *OPENARCH'98*, 1998.

[White, 1994] J. E. White. Telescript Technology: The Foundation for the Electronic Marketplace. General Magic, 1994.

[Yahara *et al.*, 1994] M. Yahara, B. Bershad, C. Maeda, and E. Moss. Efficient Packet Demultiplexing for Multiple Endpoints and Large Messages. In *Proc. of the 1994 Winter USENIX*, 1994.

[Yemini and da Silva, 1996] Y. Yemini and S. da Silva. Towards Programmable Networks. In *FIP/IEEE Intl. Workshop on Distributed Systems Operations and Management*, Italy, October 1996.

[Zander and Forchheimer, 1983] J. Zander and R. Forchheimer. Softnet - An Approach to High-Level Packet Communication. In *ARRL 2nd Computer Networking Conference*, San Francisco, CA, March 1983.

[Zappala *et al.*, 1997] Daniel Zappala, Deborah Estrin, and Scott Shenker. Alternate Path Routing and Pinning for Interdomain Multicast Routing. Technical Report TR97-655, CS Dept., University of Southern California, 1997.

[Zhang *et al.*, 1998] Lixia Zhang, Scott Michel, Khoi Nguyen, and Adam Rosenstein. Adaptive Web Caching: Towards a New Global Architecture. In *3rd International WWW Caching Workshop*, Manchester, UK, June 1998.