PRACTICAL ACTIVE PACKETS

Jonathan T. Moore

A DISSERTATION

in

Computer and Information Science

Presented to the Faculties of the University of Pennsylvania

in Partial Fulfillment of the Requirements for the Degree of Doctor of Philosophy

2002

Scott M. Nettles
Supervisor of Dissertation

Benjamin C. Pierce
Graduate Group Chair

In loving memory of

Thomas G. Moore, Ph.D.

Acknowledgments

I have many people to thank for their influence on this dissertation. I turn naturally to my family first, as my entire extended family has always promoted curiosity, creativity, and a love of learning—all traits vital to completing a Ph.D. They have always supported my academic endeavors, even when they took seven years.... My parents, Thomas and Nancy, were my earliest role models (and still are): one with a Ph.D. and one with a Master's degree. For the longest time I thought *everyone* went to grad school after college, so of course I had to go, too! Mom and Dad both encouraged my desire to learn more; I distinctly remember having Mom teach me long multiplication before I learned it in school, and having Dad write down a formula for displaying character-based graphics on our Ohio Scientific home computer (perhaps my first run-in with algebra). I would clearly not be the person I am today without all of their love and support over the years.

My sister, Jen, has been through most of these learning experiences with me and is now a biochemical engineer (incidentally, her present to me after my defense was a trip to the Franklin Institute and a session of making alka seltzer rockets out of film canisters). I have always been impressed by her creativity and inquisitiveness and am grateful we are so close (in both the spiritual and geographical senses).

Turning aside from family for a moment, I would like to thank the members of my thesis committee: Michael Greenwald, Peter Lee, Benjamin Pierce, and Jonathan Smith. All of them have given a great deal of input towards the final version of this document, especially Benjamin Pierce; this dissertation is substantially better now than it was when I defended it, thanks to these people. I owe a special debt of gratitude to Jonathan Smith, whose generosity has allowed me to stay here at Penn for so many years.

Certainly, my advisor, Scott Nettles, is the person most responsible for my research abilities. I have truly valued the time spent under his supervision; our in-person interactions have always been invigorating and productive, and I feel this has given me a taste for enjoyable research collaborations. I hope that Scott's dedication to running good experiments, his ability to state and organize the "story" behind a research paper, and his speaking abilities have rubbed off on me. I believe he has made sure I have developed the necessary skills to be an independent researcher, and for that I will always be thankful.

I also must not forget Matthew Keesan, Hooman Radfar, and Julian Zbogar-Smith, who worked on the AVid system as their senior project. I thoroughly enjoyed working with them; the enthusiasm of undergraduates is infectious. They hammered out the specifics of the algorithms used for our active video-on-demand system, developing them on a less-than-stable SNAP platform. I thank them for their patience and dedication.

No graduate school experience (or dissertation acknowledgment section, for that matter) would be complete without a cast of fellow grad students and other lab denizens. My time in the DSL has been full of personalities and memories: watching Jonathan Shapiro ask questions at department colloquia; learning the lore of the Penn CIS department from Sanjay Udani and Scott Alexander; absorbing Unix sysadmin arcana from Alex Garthwaite; surveying Ilija Hadzic's hardware stockpile in the center of the lab; learning how to expense business trips from Angelos Keromytis; getting Linux tips from Steve Muir (and learning that yes, IBM keyboards *are* pretty rugged); being motivated by Bill Arbaugh (who at times seemed to be making more progress part-time than I was as a full-time student); learning about Napster from Luke Hornof; learning how to *really* fill out expense reports from Sotiris Ioannidis; learning how to build a PC from scratch from Stefan Miltchev; learning that people really can telecommute effectively by watching (or rather, not watching) Kostas Anagnostakis; discovering that Aaron Marks thinks I am an android; learning that Chuck Davin finds thesis proposals exciting because they are a sign that people actually *graduate*; learning from Frey Kuo that there are fewer interruptions when you work late at night; learning from Peifang Zheng that being a parent is hard work; learning from Björn Knutsson that the butchers in Sweden are much better than the ones we have here; and learning from Dekai Li how to evade leading questions posed by Greeks.

Of course, there are two fellow graduate students who deserve special recognition. The first is Michael Hicks, my partner in the renowned "Mike and Jon show"—the earliest parts of our lists of publications are identical (and I have every intention that substantial parts of our future *curricula vitae* should overlap as well). Mike and I both started at Penn in the fall of 1995, took the same classes, had the same advisor, and worked on the same research projects. Naturally, we became quite close friends. I learned a great deal about dedication and a desire to "get the details right" from Mike; I think I taught Mike more patience than anything else, so I clearly came out on top of that exchange.... Mike also provided many useful comments and insights throughout the development of SNAP (even taking a break from his own thesis to write the PLAN-to-SNAP compiler). More importantly, however, I have always appreciated Mike's insight and advice on life in general, and could not have asked for a better best man at my wedding (nor for a better best man's toast).

This brings me to the last of my fellow graduate students, my wife Jessica. Jess has sometimes been a cheerleader, and often a motivator (particularly when I was honing my Starcraft skills a bit too much!), but always the right role at the right time. I have learned a great deal from her, particularly with respect to planning, managing several things at once, and keeping one's priorities straight. In addition, I feel lucky that she is also a computer scientist, as she has truly been able to understand and sympathize with my various travails. I think it is hard to adequately thank Jess for her contribution to the quality of my dissertation, since her method has generally been to improve the quality of my life as a whole. I thank her for all of her love and support, and I thank God for bringing her into my life.

Abstract

PRACTICAL ACTIVE PACKETS

Jonathan T. Moore

Supervisor: Scott M. Nettles

Active networking adds programmability to the network infrastructure to promote service introduction. One approach involves *active packets* that carry programs rather than standard passive headers. To date, no one has proposed an active packet system that is truly practical: providing added flexibility over passive packet schemes without sacrificing either safety or efficiency. In this work, we propose a new system, SNAP (Safe and Nimble Active Packets), that strikes a useful balance.

First, SNAP is safe. We use a combination of language design (limited expressiveness) and safe interpretation techniques that allow us to show that SNAP exhibits *robustness* (resistance to malicious or buggy code), *isolation* (non-interference with other packets), and *resource predictability*. In particular, we prove that each execution of a packet program consumes at most an amount of CPU and memory resources that are linearly proportional to the program's length; we also show the total number of packet executions that can be caused by a packet or its descendents is bounded.

Second, SNAP is efficient. We designed the SNAP packet format to minimize memory overhead: most packets can be executed *in-place* in kernel network buffers. We show experimentally that SNAP latency and bandwidth microbenchmarks perform within a few percent of their IP-family counterparts in software routers connected by 100 Mb/s Ethernet links. SNAP incurs extremely low overhead: IP-like functionality is available at IP-like performance.

Finally, SNAP is flexible, despite the fact that we have limited its expressiveness. We briefly describe a compiler that translates PLAN (an earlier active packet language) into SNAP using a combination of function inlining and sending copies of the current packet over a node's loopback interface to emulate backward branches. Since this loopback technique consumes one unit of resource bound, we find that the resource bound field of the packet is a convenient knob for trading off tight bounds on global resource usage for increased flexibility (in terms of the amount of looping available to the packet). We also describe two new native SNAP applications: distributed denial-of-service attack detection and active video-on-demand.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

The rapid growth of the Internet has led to increased demand for a wide array of new applications and services. Some can be accommodated by establishing new end-to-end protocols; indeed, email (SMTP) [Pos82], file transfer (FTP) [PR85], and the World Wide Web (HTTP) [FGM+99] are all successful examples of this approach.

However, we are also seeing a demand for new *network*-layer services, and the one-size-fits-all service model of IP [Pos81b] is proving too inflexible. New protocols are forced to take *ad hoc* approaches, either using layer-crossing network elements to examine non-IP portions of a packet (*e.g.*, firewalls, proxies, and web caches) or re-using "unused" portions of the IPv4 header[1] (*e.g.*, core-stateless fair queueing [SSZ98] and packet-marking traceback schemes [SP01, SWKA00]).

Clearly, IPv4 is not flexible enough. Unfortunately, the standardization processes of bodies like the Internet Engineering Task Force (IETF) move too slowly to track technology advances. Consider that the first IPv6 request-for-comments (RFC) appeared in 1995 [BM95], its latest RFC [DH98] is dated 1998, and yet it is *still* not widely deployed [Law01]. Such slow standardization and adoption have prompted research into *active networks* (AN): namely, adding programmability to the network infrastructure to promote the introduction of new services in a timely manner.

One of the most aggressive approaches to active networking is the use of *active packets* (or *capsules* [TW96]). Active packets contain a program instead of a traditional packet header. This packet program executes at interior network nodes and determines the service the packet will receive.

[1] Typically, the type-of-service and fragment identification fields.

From the perspective of active packets, we can also view the IP specification as a packet programming language definition. Ordinary IP headers, then, are "programs" in this language (essentially, "deliver me to the given host"), and routers "interpret" these programs in the process of packet forwarding. As we observed above, developers are finding they want to write packet programs that IPv4 is unable to express. Active packet systems seek to address this problem by replacing IPv4 with a more expressive programming language.

While the active packet approach offers extreme flexibility—customizability on a per-packet basis—it has to this point been received with skepticism by the traditional networking community. Current active packet system prototypes [WGT98, HKM+98b, NGK99, Tsc97, SJS+99, WJOP00] have demonstrated poor performance and/or inadequate safety guarantees. These facts lead naturally to the question, "Can active packets ever be practical?"

In this work, we answer "yes" to this question by presenting a new, *practical* active packet scheme called SNAP (Safe and Nimble Active Packets). We will demonstrate that SNAP provides strong *safety* guarantees, that a SNAP implementation can be *efficient*, and that SNAP is *flexible* enough to deploy new and interesting network protocols.

1.1 Practicality Framework

Before we can say that a particular active packet system is practical, we must establish a framework, or set of criteria, for making such a judgment. Informally, we would consider an active packet system to be practical if it could be reasonably deployed in some significant subset of the Internet. We consider three main criteria in our framework:

- **Safety (and Security):** preventing outages caused by malicious or incorrect packet programs;
- **Efficiency:** providing acceptable performance; and
- **Flexibility:** providing a useful range of programmable functionality.

Before considering each of these points in detail, we first argue that these are the appropriate issues to consider. First, if active packets make it possible to crash the network

or otherwise compromise network security, it certainly will not be practical (or prudent) to deploy them in the public network. Second, inefficient active packet implementations will make it much harder for the added flexibility to provide end-to-end application performance benefits. Finally, adding flexibility is the whole motivation for using active packets at all. Thus, they must add significantly to our ability to evolve and customize the network as contrasted to passive packet systems. We may wish to limit active packets' flexibility, especially to help provide safety and efficiency, but not so much that they cannot be programmed to do useful tasks.

Clearly, the three criteria of safety, efficiency, and flexibility are *necessary* for an active packet system to be practical. Are they sufficient? There are certainly other desirable characteristics an active packet system might have, such as usability (How easy is it to program? Does it interface well with existing languages such as Java or C?) and deployability (Can it co-exist with legacy IP nodes?). However, we feel that the most critical technical challenges arise in balancing the basic tradeoffs between safety, efficiency, and flexibility (we argue below that no existing system balances them well), so we will consider them sufficient for the purposes of the current work. We will explore these tradeoffs in more detail after we more thoroughly define each of our three criteria and explore the current state-of-the-art.

1.1.1 Safety (and Security)

Because the Internet is a shared resource it must be protected from damage caused by improper use. Protection must be provided against malicious attacks as well as unintentionally damaging behavior (*e.g.*, a buggy piece of client software that sends malformed packets). Internet security is already imperfect: for example, the infamous "ping of death" [CER96] resulted in crashes due to improper handling of oversized IP packets.

The advent of active networks and, in particular, active packet systems, complicates the picture. Execution environments for active packets are likely more complex than "passive" protocol stacks, making it more difficult to locate bugs. Furthermore, the packet programs *themselves* may perform malicious actions or have bugs, even if the execution environment is correctly implemented.

We define the *safety* of an active packet system as the degree to which it protects the overall network from attack or misuse, whether deliberate or accidental[2]. In particular, in a safe system, an active packet should not be able to crash or otherwise subvert a network node; active packets should not be able to interfere with packets belonging to other users without permission; and active packets should not be able to carry out denial-of-service attacks that consume so many computational resources on a single node or so many resources throughout the network that some or all of the network becomes unusable to others.

From this point of view traditional passive packets are fairly safe. Malformed packets are checked for and (usually) discarded, and well-formed packets typically require little processing. Indeed, the "programs" that can be expressed by the usual static header fields are very simple; thus, a single packet cannot, in general, tie up an untoward amount of resources while being processed on a node. We contend that an active packet system should provide *at least* the level of safety of traditional passive packet systems. Failure to do so would be a very sizeable barrier to deployment in the Internet.

1.1.2 Efficiency

Part of the goal of active networking is to address the limitations of IP's one-size-fits-all model. Nonetheless, the very simplicity of IP's service model is one of the reasons for its success. Because the functionality provided is minimal, IP can be efficiently supported over a diverse set of link layer protocols.

It seems likely that an active packet system providing greater functionality will be more complex than a traditional passive network-layer protocol; this added complexity may come at the cost of performance both in terms of providing actual service as well as in ensuring safety. It is important that performance is not degraded so much that the active packet system is rendered essentially useless.

We define the *efficiency* of an active packet system in terms of its *intrinsic overhead*: how much does it hurt performance (especially for common tasks like simple forwarding) to support programmable packets? A system's efficiency will be an important factor in

[2]This broad definition of safety also encompasses "security," so in the remainder of this work we will use the shorter term "safety" instead of the longer "safety and security."

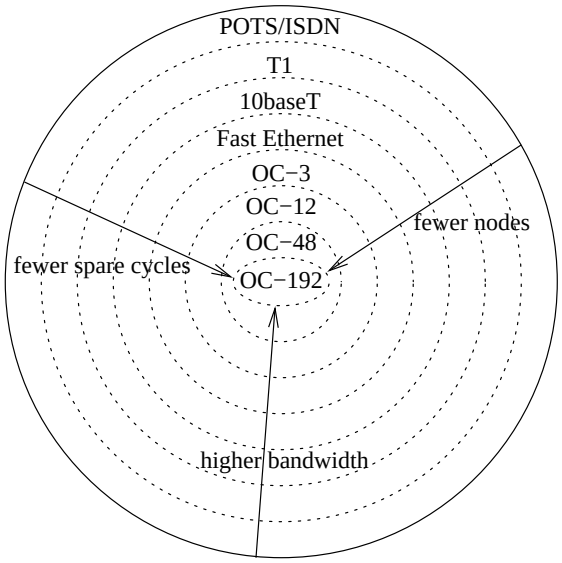

Figure 1.1: Computation over bandwidth (COB) ring model [Ale98].

determining where it is deployed. Current firewalls already have lower efficiency than routers, and control-plane operations are infrequent and thus will more easily tolerate low efficiency than data transport. As the efficiency of a system increases, the regions of the network where it may be used for in-band data transport increase. As shown in Figure 1.1, at the lower-speed edges of the network, more cycles are available per unit of bandwidth provided and lower efficiency systems will be practical for data transport; as we move towards the core, higher efficiency will be needed. In general, the efficiency of an active packet system directly impacts how many nodes in the Internet we could practically make "active." Of course, the majority of nodes in the Internet are around the outside edges, so there is likely to be substantial impact even if an active packet system's efficiency is not sufficient for the highest-bandwidth core.

1.1.3 Flexibility

The appeal of active networking is increased flexibility resulting from a programmable network infrastructure. This added flexibility should permit faster or better upgrades of network services as well as enabling new, previously unsupported services.

We define the *flexibility* of an active packet system as the degree to which new functionality is made possible by the system as contrasted to passive approaches. Here, the IP-based Internet is a natural reference point, as an active packet system offering less flexibility than IP would hardly qualify as an "active networking" system.

Unfortunately, flexibility is a property that is difficult to quantify. In cases where we can emulate one active packet system with another, we can conclude that the second is at least as flexible as the first, but this merely provides a relative ordering rather than an absolute measure. As this emulation is often non-trivial, attempting it on a pair-wise basis for all existing systems to get a complete ordering in terms of flexibility is well beyond the scope of this dissertation.

On the other hand, existing systems have demonstrated a variety of active packet applications. Thus we can use the following questions as heuristics for evaluating the flexibility of active packet system:

1. Can we implement existing active applications using the given system?

2. Can we implement new active applications using the given system?

A system for which we could answer "yes" to both questions would likely have enough flexibility to be considered as a candidate for a practical active packet system.

1.2 State of the Art

In this section we examine the state of the art in existing active packet systems, using our three criteria of *safety*, *efficiency*, and *flexibility* as guides. We focus mainly on these attributes, mentioning the various approaches taken by existing projects. The main projects we will discuss are: ANTS [WGT98], PLAN [HKM+98b, HMA+99], PAN [NGK99], Smart

System	Safe	Efficient	Flexible
ALIEN	√		√
ANTS	√		√
MØ			√
PAN		√	√
PLAN	√		√
SafetyNet	√		√
Smart Packets	√		
StreamCode		√	√

Table 1.1: Summary of practicality criteria for existing active packet systems.

Packets [SJS+99, SJS+00], ALIEN [Ale98], MØ [Tsc97], SafetyNet [WJOP00], and StreamCode [HEK00, EHH01]. A brief summary of these projects in terms of our three criteria is given in Table 1.1. For a more in-depth, project-by-project treatment of these systems, see Chapter 8.

1.2.1 Safety

Here we consider the safety (and security) both of individual nodes and of the network as a whole. Most existing systems are able to protect individual nodes and other users' flows from ill-behaved active packets. However, there is generally less success in providing global network protection.

Three issues arise when protecting individual nodes: nodes must not be crashed or subverted (*robustness*), nodes must not have all of their resources consumed (*resource predictability*), and packets must not be allowed to interfere directly with or observe other packets without permission (*isolation*). Generally speaking, three strategies are combined to provide robustness. First, type-safe languages, array-bounds checking, and garbage collection are used to enforce memory safety [WGT98, HMA+99, Ale98, WJOP00]. Second, packet execution that takes place in a controlled environment (*e.g.*, a virtual machine) allows various kinds of access control to be enforced [SJS+99, Tsc97, Ale98, WGT98, HKM+98b]. Finally, the interface made available to active packets is limited, thus restricting these programs from invoking functionality that is unsafe [HKM+98b, SJS+99]. Most systems provide good individual node safety using these approaches.

Existing systems fail in controlling the resource utilization of active packets well. Several systems use time-to-live (TTL) counters to limit packet proliferation [HKM+98b, WGT98, NGK99, SJS+99] and watchdog timers and allocation limits to terminate packets using too many local resources [WGT98, SJS+99]. In systems implemented in Java, these approaches sacrifice safety because premature termination can be unsafe [HCC+98]. Furthermore, they incur execution overhead to monitor dynamic instruction counts and memory allocation requests.

By contrast, PLAN [HKM+98b] limits packet execution by restricting its expressiveness so that all programs must terminate. However, bounding this termination time is problematic as packets may run in time exponential in their length[3] without further restrictions to the language. In general, we would prefer that packet resource usage be more predictable, so that the network can decide how to process packets most effectively.

If individual nodes are protected from corruption, the network as a whole is protected. Unfortunately, this is not true for resource utilization, since a packet can simply move to another node and consume more resources. Worse, many applications of active packets require that they be able to split into more than one packet (*e.g.*, multicast), again raising the possibility of exponential global resource consumption. For example, StreamCode [EHH01], PAN [NGK99], and ANTS [WGT98] contain multicast primitives that could potentially allow a single packet to fan out (*e.g.*, splitting recursively into two packets at each hop) and then converge on a single victim in a denial-of-service attack. PLAN [HKM+98b, HMA+99] attempts to deal with this problem by using a "conservation of hop count" model where parent packets must divide their time-to-live (TTL) among their children. Unfortunately, none of these systems provides a completely satisfactory solution: we seem forced to choose between allowing exponential fanout (bad for safety) or making it difficult for a programmer to apportion remaining resources among spawned packets (bad for usability).

[3] For example, consider the behavior of calling `d()` in the following program:

```
fun a() = ()
fun b() = (a(); a())
fun c() = (b(); b())
fun d() = (c(); c())
```

1.2.2 Efficiency

All of the systems whose performance has been measured provide enough efficiency to be used for most control-plane tasks. One thing is clear, however: the systems that require per-packet cryptographic authentication for all packets [Ale98, SJS+99] will not be suitable for transport-plane applications in any environment but those with the most spare cycles compared to bandwidth (namely, those around the very edge of the ring model shown in Figure 1.1). Alexander [Ale98] demonstrates that it is critical to efficiency that authentication be optional, although making it available will facilitate applications where it is needed and its costs are tolerable.

ANTS [WGT98] and PLAN [HMA+99] are both suitable for transport in some parts of the cycle/bandwidth space, but not for line rates common in workplace desktops. Only PAN [NGK99] and StreamCode [HEK00] can saturate a 100 Mb/s Ethernet from a desktop machine, but PAN does so by sacrificing significant safety (nodes are easily compromised) and both StreamCode and PAN fail to provide global network resource predictability.

Although more an aspect of implementation than of design, it should also be mentioned that there are gains to be made by building better interpreters and virtual machines than currently exist for active packet systems. Although other opportunities no doubt exist, two obvious improvements are in the implementation of thread systems and garbage collectors, both of which impose significant overheads in current systems [WGT98, NGK99, HMA+99, Ale98].

1.2.3 Flexibility

The good news is that all of the systems achieve significant flexibility, especially the more mature systems, such as ANTS [WGT98] and PLAN [HMA+99]. This is demonstrated by the wide variety of applications that have been built, such as application-driven routing [HMA+99], transparent redirection of web requests to nearby caches [LG98], distributed on-line auctions [LWG98], reliable multicast [LGT98], mobile code firewalls [HKS02], content-sensitive multicast [EHH01], and reduced network management traffic [SJS+00].

It is interesting to ask whether these applications really require all the flexibility provided by existing systems. Specifically, we might still be able to implement all of these

applications in a system that was less flexible. If so, and the reduced flexibility of the system allowed it to provide better safety and/or efficiency, it would potentially occupy a more desirable point in the active packet design space.

1.3 Tradeoffs

As we have just seen, existing systems have made significant advances in meeting our criteria of safety, efficiency, and flexibility. Indeed, in the areas of node robustness [HMA+99, WGT98] and performance [NGK99, HEK00], some have already approached the levels afforded by traditional passive packet systems, all while providing more flexibility. Unfortunately, no single system has been able to provide acceptable levels of both safety *and* efficiency.

Striking a good balance between safety, efficiency, and flexibility is a difficult task partly because they are all in tension with one another. For example, providing safety may require additional run-time checks that hurt efficiency (in fact, PAN [NGK99] explicitly omits safety considerations to achieve high efficiency). Similarly, the more flexible an active packet system, the more difficult it can be to ensure that it is safe (for example, permitting a packet to apply arbitrary binary patches to a router's code is extremely flexible but difficult to make safe). Finally, it is likely to be harder to implement a more flexible (and often more complex) active packet system efficiently.

One of the major contributions of this work is the presentation of a new system, SNAP (Safe and Nimble Active Packets), that finds a "sweet spot" in this tradeoff space. Seen at a high level, SNAP's implementation avoids many of the overheads of previous active packet systems, restricts the language to provide resource predictability, and then provides a tunable "knob" to trade off between tight global guarantees and overall flexibility.

1.4 Goals and Approach

In the present work, we show that practical active packet systems can be built by demonstrating that SNAP achieves acceptable levels of safety, efficiency, *and* flexibility. Our specific goals for SNAP are:

- **Safety**. SNAP packets should not be able to subvert or crash a node (*robustness*); SNAP packets should not be able to directly interfere with other packets without permission (*isolation*); and SNAP packets' resource usage should be *predictable*, both for individual packet executions as well as globally across multiple nodes (*resource predictability*).

- **Efficiency.** Although complex packet programs will likely be more expensive to execute than passive packet headers, it is important that the infrastructure needed to permit more flexibility should not add significant overhead. Specifically, IP-like functionality should be available at IP-like performance. We focus on the very common setting of software routers connected by 100 Mb/s Ethernet links.

- **Flexibility.** It should be possible to express a wide range of useful and interesting active applications with SNAP. In particular, SNAP should offer significantly more flexibility than traditional passive packet schemes.

1.4.1 SNAP overview

A key contribution of this work is our model of active packet execution, which is described more fully in Chapter 2. Programs are carried in packets and are used to connect and control node-resident functionality, much as a Unix shell script can be used to connect and control more basic utility programs like `grep` and `sort`. For safety and security reasons, though, packets do not have unlimited abilities; in particular, we want to ensure that packets cannot subvert a node. Most importantly, we require *resource predictability*: each execution of a packet program is guaranteed to consume amounts of CPU and memory resources that are linearly proportional to the program's length. Furthermore, a *resource bound* field in the packet limits the total number of packet executions that can be caused by a packet or its descendents.

Our implementation of this model is the SNAP programming language, which we introduce in Chapter 2. SNAP is a low-level, domain-specific language that has been designed with the above practicality goals specifically in mind. SNAP is a "second generation"

active packet system: we have built upon the successes of previous systems (primarily PLAN/PLANet [HKM+98b, HMA+99]) while addressing their shortcomings.

SNAP programs execute on a stack-based bytecode virtual machine (VM). The primitives of the language allow a packet program to obtain information about itself and its environment, perform simple computations, make decisions, and send new packets. One important restriction in SNAP is that *all branch instructions must go forward*; this restriction is essential for meeting our model's linear resource usage requirements for local packet executions. Another important characteristic of a SNAP packet is its *resource bound* (RB) field. This field is decremented for each network hop, like IPv4's time-to-live (TTL) field. Furthermore, we have a *conservation of resource bound* property, where a packet must donate some of its own RB to any child packets it sends out.

1.4.2 Demonstration

The main part of this dissertation is a demonstration that SNAP meets our design goals of safety, efficiency, and flexibility. We treat each of the criteria separately, and use a different approach for each one.

Safety. A practical active packet system must have three main safety characteristics:

1. *Robustness*: node integrity must be protected from malicious or buggy active packet programs.

2. *Isolation*: active packets should not be able to directly affect other packets without permission.

3. *Resource Safety*: active packets must use a *predictable* amount of resources, both locally on a per-execution basis and globally on a per-packet-lifetime basis.

These are all implied by our model of active packet execution, so our demonstration in Chapter 3 will be an argument that SNAP fits the model. We take the approach of *language restriction* (as in PLAN [HKM+98b]): we have intentionally limited the expressiveness of the SNAP language, mainly with our choices of language primitives and with our "forward

branch only" restriction. In fact, the branch restriction was added expressly to support resource predictability.

We argue that SNAP achieves robustness and isolation simply because there are no primitives available for directly affecting a node or other packets, and a safe interpreter can ensure that active packets "play by the rules." These techniques are already well known and in use by previous active packet systems.

The most novel part of our model, however, is the notion of resource predictability. We prove that SNAP satisfies this property; namely, that a given local execution of a SNAP program will have execution time and memory usage in linear proportion to its length and that the number of local executions that can be caused by a packet or any of its descendents can be strictly bounded. This global predictability means that a router can use a packet's resource bound field as a "knob," setting it lower or higher to trade off between having a tight bound on the packet's resource usage or allowing the packet more flexibility.

The predictability proofs are not difficult; in fact, the forward branch restriction was added to SNAP specifically to simplify this proof. A sketch of the local predictability proof is as follows: generally speaking, each SNAP instruction can be executed in constant space and time. The forward branch restriction guarantees that each instruction in a given SNAP program is executed at most once. Therefore, a local execution's resource usage is linear in the number of SNAP instructions. The global predictability proof follows straightforwardly from our conservation of resource bound property: the original packet's RB must be subdivided among all of its children, and since 1 RB is consumed per network hop, the number of local executions that can be caused is strictly bounded by the original packet's RB field.

We then develop a more general model of SNAP resource usage that provides a way to account for the resource usage of active packet constructs, including ones that are not currently a part of the SNAP language. This model will allow us to understand the impact of non-constant time or space instructions and node-resident services such as those in use by existing active packet systems. The general idea is to consider the resources that would be used by the new service, and contrast it with the resource usage of a

"normal" SNAP packet. This information will allow us to appropriately charge for the new service by adjusting a packet's resource bound field to maintain our global network resource guarantees.

Efficiency. Our goal for efficiency is that a practical active packet system will impose a minimal overhead for traditional, passive-packet-style functionality. We demonstrate that SNAP can be efficiently implemented, thereby achieving this goal.

We have implemented a SNAP virtual machine (VM) that uses bytecode-style wire formats and execution, permitting small space and time overheads. In particular, we have designed our wire format to allow a packet program to be executed *in-place* in a packet buffer, thereby avoiding potentially expensive marshalling, unmarshalling, or copying. We have also optimized for certain important common cases (like basic packet forwarding). Finally, where possible, we have opted for small, dynamic costs rather than large, static costs (*e.g.*, software fault isolation rather than static type checking), since an ephemeral active packet may not even execute all of its code on a given node.

We demonstrate the efficiency of this implementation experimentally in Chapter 5. Specifically, we compare an in-kernel SNAP VM to the standard Linux IP protocol family implementation. We find that SNAP imposes low overhead for basic passive packet service: SNAP programs for *ping* and datagram delivery can achieve latency and throughput performance that is quite close to their IP-family equivalents (ICMP ECHO-REPLY [Pos81a] and UDP [Pos80]) in settings where software routers are practical.

Flexibility. A practical active packet system should be able to express a useful range of active applications. In Chapter 6, we show that SNAP meets this criterion in two main ways: first, we demonstrate that SNAP achieves similar flexibility to existing active packet systems. Second, we demonstrate new active applications implemented in SNAP.

To demonstrate the first point, we describe a PLAN-to-SNAP compiler. The existence of this compiler means that SNAP can express the same algorithms that can be programmed in PLAN. As PLAN has been used to implement some nontrivial active applications (namely, PLANet, an entirely active internetwork) and is generally considered one of the more flexible existing systems, this argues strongly for SNAP's flexibility.

We then describe new applications that are made possible by SNAP's flexibility. Specifically, we use SNAP packets' ability to direct themselves to implement a lightweight mobile-agent-style network management application. Secondly, we use the ability to perform application-specific computation inside the network to achieve improved end-to-end application performance.

These two applications are:

1. *Distributed Denial-of-Service (DDoS) Detection*: Active packets are used to query the load on an autonomous domain's (AD) ingress routers to detect a possible denial-of-service attack. Because SNAP packets can act as lightweight mobile agents, we can reduce overall network management traffic by distributing computation and avoiding having to query all nodes in some cases.

2. *Active Video-on-demand*: In the Active Video (AVid) application, real-time streaming video data is carried by active packets to multiple receivers. SNAP permits two main advantages here: first, a dynamic multicast framework that interoperates with legacy IP nodes (even ones not supporting IGMP [Fen97]), and second, the use of application-specific knowledge at congestion points to achieve better end-to-end application performance.

These applications explore SNAP's flexibility in both data-plane (AVid) and control-plane (DDoS Detection) roles. In particular, they stress-test SNAP in the areas where its flexibility has been limited (namely, the forward-branch restriction makes looping difficult). Furthermore, since these application areas are of general interest, we expect the algorithms presented here to have an impact beyond just the active networking community.

The dissertation concludes with a discussion of future directions in active packet research, an in-depth, project-by-project assessment of related work in terms of our practicality framework, and a summary of our research contributions. Finally, we include an appendix containing a SNAP instruction set reference.

Chapter 2

Safe and Nimble Active Packets

We develop a model of execution that allows active packets to serve as "network shell scripts": packet programs perform simple computations to control more powerful node-resident services. This *two-level architecture* of packets and services appears in most existing active packet systems, notably PLANet [HMA+99].

Since safety is one of our primary concerns, our model also requires packets' abilities to be somewhat limited so that they cannot subvert a node. We have two main requirements: first, we must be able to control what node-resident functionality is accessible, and second, packet program executions must use a predictable amount of resources. Our intuition is that these restrictions are acceptable: most active packet programs will be simple and small (as they must fit in a single packet, after all). Later, in Chapter 6, we will demonstrate that useful packet programs can still be written in this restricted model.

Still, this is a very abstract model of active packet execution. Indeed, although we have a particular implementation of this model (SNAP), one could imagine other active packet systems that could satisfy this model as well (*e.g.*, ensuring safety by different means). Further work could be based on the model itself (in particular, based on its resource predictability) without necessarily being tied to SNAP. For example, traffic engineering efforts based on the predictability of packets' resource usage would be orthogonal to how the resource properties were guaranteed.

It is important to realize that unicast IPv4 packets satisfy this model as well. As we mentioned in Chapter 1, IPv4 headers can be viewed as programs in a very restricted programming language. Given that we would like to replace passive packet headers with

programs in a different (and hopefully more flexible) programming language, it is useful to have a model that encompasses both active and passive packet approaches.

2.1 Implementing the Model

SNAP (Safe and Nimble Active Packets) is a low-level, domain-specific language that has been designed to satisfy the above model. SNAP provides basic control-flow operations, simple computations (*e.g.*, arithmetic), and facilities for invoking node-resident services.

Recall that our model requires us to restrict the functionality available to packets for robustness (and general security). A common technique in other active packet systems [SJS+00, HKS02] is to have a set of "default abilities" that are available to every packet but permit a packet to present cryptographic credentials to gain access to expanded functionality. With SNAP, we follow the approach developed in PLAN [HKM+98b]: we intentionally *restrict* the language so that we can guarantee that all programs are safe, and leave authentication optional for those services that require it.

With SNAP, our language primitives have been chosen so that a packet can control what happens to itself, but not much else. A carefully constructed execution environment, or "safe interpreter," can guarantee that a packet program is restricted to our defined primitives and cannot subvert the system by going "out-of-bounds" (*e.g.*, with a buffer overrun).

The second part of our model requires that the amount resources an active packet can use must be predictable from both local and global viewpoints. It is possible (and simple) to guarantee per-execution linear CPU and memory usage via the use of watchdog timers and memory allocation limits (as many previous active packet systems have done). By contrast, we have used the approach of language restriction, as we will describe below. This has two main advantages: first, it relieves the implementation from the overhead of constantly checking instruction counts and memory allocations against their limits, and second, programmers know that if they can express their programs in SNAP that they will run to completion (assuming no errors occur).

Instruction class	Examples
network control	**forw**, **forwto**, **send**, **demux**
flow control	**bne**, **beq**, **ji**, **paj**
stack manipulation	**pint**, **pop**, **pull**
environment query	**getsrc**, **getrb**, **here**, **ishere**
simple computation	**add**, **addi**, **xor**, **eq**
tuple manipulation	**mktup**, **nth**
service access	**calls**

Table 2.1: SNAP instruction classes

The resource predictability guarantees we provide for SNAP programs can be derived from three main properties of the language. First, we require that all SNAP instructions execute in constant time and space, with a few exceptions that we shall mention below. Secondly, we require that *all branches go forward*. Taken together, these requirements guarantee that one local execution of a SNAP program will have linear CPU and memory usage.

Finally, to ensure global predictability, each packet carries with it a *resource bound* (RB), as in PLAN [HKM+98b]. This resource bound works very similarly to the IPv4 time-to-live (TTL) field; each node decrements it on receiving a packet, and a packet is dropped when it depletes its resource bound. In addition, packets are required to give some of their remaining resource bound to any child packets they spawn (conservation of resource bound), thus limiting the effects of exponential packet fanout[1].

2.2 The SNAP Language

A SNAP program consists of a sequence of bytecode instructions, a stack, and a heap. The datatypes supported are integers, floating point numbers, addresses, exceptions, byte arrays, and tuples of the preceding types. Each SNAP instruction consists of an opcode and, optionally, an immediate argument. SNAP instructions fall into seven classes, listed in Table 2.1; a full list of instructions and their informal semantics is presented in Appendix A.

[1] For example, a packet with initial resource bound R that "forks" at each hop and sends two copies of itself must divide its resource bound between them. This means that the last generation of packets can only have gotten at most a distance of $\log_2 R$ hops from the original node.

```
forw          ; move on if not at dest
bne     5     ; jump 5 instrs if nonzero on top
pint    1     ; 1 means "on return trip"
getsrc        ; get source field
forwto        ; send return packet
pop           ; pop the 1 for local ping
demux         ; deliver payload
```

Figure 2.1: SNAP code for ping.

2.3 Example: Ping

We illustrate SNAP's basic features concretely by using a version of *ping* coded in SNAP, shown in Figure 2.1. Although it only contains seven instructions, *ping* nonetheless provides a good example of the functionality available to SNAP programs.

Let us assume that we have two neighboring nodes, A and B, and that we want to ping B from A. In this case, we create a packet containing the ping code and an initial stack of $[0 :: port :: payload]$. The 0 on top of the stack indicates the packet is moving from source to destination, *port* is a port number corresponding to the ping application on the sending node, and *payload* is a byte array carried along with the packet.

The packet is injected into the network at A and executes there first. The first bytecode executed is **forw** ("forward"), which compares the packet's destination header field to the current host's address. If they do not match, a copy of the packet is forwarded towards the destination, and the currently running packet terminates; this is what happens on A initially. When the packet reaches its destination, **forw** simply drops through to the next instruction; this is what happens when the packet reaches B.

Forw is a special case of a more general "network control" instruction, **send**, used to spawn new packets. **Send**(s, n, r, d) creates a new packet with a copy of the current code, a stack consisting of the top s current stack values, an entry point n (the index of the first instruction to execute), r resource bound, and a destination field set to d. Executing **send** results in r resource bound being used by the child packet and thus deducted from the packet that executes the **send**. Unlike **forw**, however, **send** does not terminate the

current packet's execution. Thus **forw** is just shorthand for a test to see if the packet has reached its destination, and if not, terminating after performing a **send** that keeps the same destination and entry point as the current packet while taking the whole current stack and all of the current packet's resource bound.

Recall earlier that we claimed that each SNAP instruction executes in constant time. However, the network control operations operate in time linearly proportional to the packet's length (as the entire packet must be copied out over the network interface). We address this problem by limiting the number of network control operations to a constant number per packet (currently, the number of network interfaces of the current node), thus amortizing their costs across the rest of the instructions.

On B, the next bytecode executed is **bne** ("branch if not equal to zero"). **Bne** consumes the top stack value as an argument, and if it is nonzero takes the branch by adding the immediate argument to the program counter. In our example, the 0 on top of the stack is popped, and, since the test fails, the branch falls through to the next instruction.

In addition to **bne**, we provide a variety of branch types, including conditional branches (**beq**, **bne**), unconditional branches (**ji**, "jump immediate"), and branches whose targets are carried on the stack (**paj**, "pop and jump"). All branches must "go forward" (offsets must be positive), implying that standard loops and function calls cannot be straightforwardly encoded. Despite this seemingly harsh restriction, we will see in Section 6 not only that certain programming techniques still allow limited looping but also that a large class of useful programs do not require backward branches.

The next bytecode executed is **pint** ("push integer"), which pushes a 1 onto the stack, signifying that the packet is now on the return trip. (Now the stack is $[1 :: port :: payload]$.) SNAP supports the usual stack manipulation operations: **push** variants for the other value types, **pop**, **pull** (copy a value from within the stack), *etc.*

Next, **getsrc** pushes a copy of the source address onto the stack. In general, SNAP provides several "environment query" instructions, which allow a packet to read the contents of its header fields (*e.g.*, **getrb** for the remaining resource bound, **getep** for the current entry point) or to query the node itself for information (*e.g.*, **here**, which pushes the current node's address on the stack).

Next, **forwto** causes the packet to be sent back towards A. **Forwto** is like **forw**, but it reads a destination argument from the stack (in our case, the address A pushed by **getsrc**). Thus the return packet is sent with a destination field set to the original source address, carrying a stack of $[1 :: port :: payload]$.

When the packet arrives at A, execution again begins at the **forw**, which falls through, since the return packet's destination address is A. The **bne** consumes the 1 on top of the stack, takes the branch and jumps 5 instructions to the **demux** instruction. **Demux** takes two arguments from the stack: a port number and a value to deliver to the port. By design, this is exactly what remains on the stack: $[port :: payload]$. Like the packet sending operations, data delivery with **demux** is not constant-time. Therefore, we limit a packet to at most one data delivery operation.

Finally, the **pop** is only executed when the destination and source are the same host. In that case, when the packet is logically "at the destination," the **forwto** will fall through, so before **demux**ing we must pop the 1 we just pushed.

2.4 Other Instructions

The instruction classes from Table 2.1 not appearing in the ping program are "simple computation," "heap manipulation," and "service access." The simple computation instructions, such as **add** or **sub**, pop one or more arguments from the stack, perform a computation (optionally with an immediate argument) and push the result. We provide standard integer and floating point arithmetic operators and relational operators. Another important simple computation instruction is **isx** ("is exception?"). In SNAP, when an exception occurs, a special exception value is placed on top of the stack; a program can then check for exceptions using **isx**, which indicates whether the top stack value is an exception value or not.

The heap manipulation instructions allow the program to allocate length-n tuples on the heap (by **mktup** n) as well as to select the ith field from existing tuples (**nth** i). We require that each **mktup** n instruction be followed by $n-1$ non-**mktup** instructions

(using *no-ops* as needed). This allows us to amortize the n allocated small values over n instructions.

Finally, the instruction **calls** s allows a packet to invoke a *service* named by the string s. Services are node-resident, general-purpose routines that augment the limited functionality of the packets. In some sense, services provide an "escape hatch" out of the expressiveness restrictions imposed by the SNAP language (and hence we will treat them specially with respect to safety). For example, we might have a service that allows packets to store soft-state on the routers they traverse. Services differ from normal instruction implementations, in that the service namespace is extensible, meaning that we can (in theory) upload new service routines at runtime to add new functionality. This is the same model of services supported in PLAN [HKM+98b].

Chapter 3

Safety

The first step of our demonstration of SNAP's practicality will be showing that it is *safe*. In the introduction, we set the following design goal:

- **Safety**. SNAP packets should not be able to subvert or crash a node (*robustness*); SNAP packets should not be able to directly interfere with other packets without permission (*isolation*); and SNAP packets' resource usage should be *predictable*, both for individual packet executions as well as globally across multiple nodes (*resource predictability*).

As we mentioned in the previous chapter, our general approach is one of language restriction: the SNAP language has been specifically designed to achieve all of the above goals.

3.1 Robustness and Isolation

To ensure that SNAP packets cannot directly interfere either with a node or with other packets, the SNAP instruction set only gives a SNAP program the ability to affect itself; there are simply no operations that act directly on the node or other packets. Proving robustness to crash or subversion simply involves ensuring that the SNAP virtual machine (VM) enforces the semantics accurately (*e.g.*, by making sure that heap accesses actually fall within the packet's heap or that a branch target is actually within the code segment of a packet).

This sort of sandboxing safety is already found in systems like ANTS [WGT98] and PLAN [HKM+98b]. ANTS and PLAN, however, achieve this by ensuring type safety,

which is a stronger form of safety than we provide. In SNAP, we take an approach similar to software fault isolation [WLAG93]: a SNAP program may contain type errors, but our VM restricts the effects of those errors to the packet itself. In this way, we greatly simplify safety enforcement. The main properties with which we are concerned are *control safety*, *stack safety*, and *heap safety*.

Control safety ensures that we only execute actual SNAP instructions (as opposed to interpreting stack data as instructions, for example). In our implementation we maintain the invariant that the program counter always points to a valid SNAP instruction. In practice this is easily maintained; each SNAP instruction is a fixed size (as we will see in the next chapter) and we only change the program counter by multiples of this size. For most instructions, the program counter simply advances to the next instruction; for branches, the program counter moves forward by one or more instructions (we check that the branch offset is positive). The last remaining check is then that the new program counter falls within the code segment of the packet[1].

Stack safety ensures that the program's stack is manipulated in a safe and consistent way. Instructions that access items in the stack first execute a test that the stack contains the necessary number of items (so that, for example, it is not possible to **pop** an empty stack). Finally, whenever an instruction causes the stack to grow, we ensure that there is enough space for the new stack items. Both of the above tests are simplified by the fact that all stack items are of fixed size.

Lastly, heap safety governs both heap allocation and heap access. In the first case, we check that there is enough room in the heap for the new allocation, and in the second case we must ensure that a heap access actually corresponds to a location within the heap. Our implementation can perform these tests simply, because heap "pointers" are represented as offsets into the heap, so a validity test merely involves ensuring that the offset is smaller than the current heap size.

We terminate a packet that attempts an instruction that violates one of these notions of safety. Note, however, that we are not concerned with type errors; for example, an instruction that expects a heap pointer as a stack argument (such as **nth**) will merely

[1] A program counter just beyond the code segment is interpreted as an implicit **exit**.

interpret the top stack value as a heap pointer and do the necessary bounds checks, even if it is really an integer, exception, or floating point value. Similarly, we do not care whether a heap offset points to the head of a heap object; we just ensure that it points somewhere in the heap. Although either of these sorts of type errors probably indicates a programming error, the virtual machine nonetheless ensures that the program still stays in its "sandbox."

Finally, we should stress that our definition of isolation means that packets cannot *directly* affect one another without permission. Indirect effects such as interactions that occur via services or competition for local resources are outside our scope, although we do consider them in Section 3.3 below.

3.2 Resource Predictability

In this section, we prove three theorems that show that all SNAP programs satisfy the resource predictability properties required by our model of execution from the previous chapter. Resource predictability has heavily influenced SNAP's design—we selected its features (forward branches only, constant time and space instructions, and conservation of resource bound) to make these proofs trivial.

The general proof sketches are as follows: due to the forward branch restriction, each SNAP instruction is executed at most once per local packet execution. Then, since each SNAP instruction runs in amortized constant time and space, we get our per-execution linear bounds on CPU and memory usage. We achieve our global predictability via SNAP's conservation of resource bound principle.

As we mentioned in the previous chapter, the services available via **calls** are node-resident, and we expect that they will be written in more general-purpose programming languages than SNAP. Therefore, we will consider service resource predictability separately in Chapter 4 and will omit **calls** from our discussions here. This decision is in keeping with our two-level model of active packet execution, which separates node-level service primitives from ephemeral active packet code.

Although we provide more detailed calculations for completeness, there are no surprises with respect to the above sketches. Those readers not particularly interested in the details (*e.g.*, the bookkeeping for amortization) are advised simply to read the statements of the theorems and then skip ahead to Section 3.3, where we discuss the implications of resource predictability.

Lemma 3.1 (One-shot Instructions). *For a given SNAP program, each instruction in the program will be executed at most once per packet execution.*

Proof. All non-branch SNAP instructions fall through to the next instruction, and all branch instructions are constrained to go forward. Therefore, the program counter is always making forward progress, so no instruction can execute more than once. □

Theorem 3.1 (Per-execution CPU Safety). *One execution of a SNAP program on a given node will require time that is at most linearly proportional to the packet's length.*

Proof. With the exception of the network control instructions **forw**, **forwto**, **send**, **hop**, **demux**, and **demuxi** and the heap allocation instruction **mktup**, all SNAP instructions execute in constant time. Let t_i be the maximum execution time of any of the constant-time instructions.

Let $demux(x)$ denote the time required to deliver x bytes of data to an application. Now, practically speaking, $demux(x) \in O(x)$, as the x bytes must typically be copied from kernel to user space. So then $demux(x) \leq dx + d_0$ for some constants d and d_0. By the SNAP language definition, a program may execute at most one **demux** or **demuxi**.

Let $send(x)$ denote the time required to send a packet of size x. Again, practically speaking, $send(x) \in O(x)$, as the x bytes must be copied out over the network interface. So then $send(x) \leq sx + s_0$ for some constants s and s_0. Also, recall from Chapter 2 that a SNAP program may execute only a constant number of packet sends equal to the number of network interfaces on the node; call this constant c.

Let $alloc(i)$ denote the time required to allocate an i-tuple in the heap (**mktup** i). Note that $alloc(i) \in O(i)$, as i stack values must be copied into the newly-allocated heap tuple. So $alloc(i) \leq ai + a_0$ for some constants a and a_0. However, since **mktup** i must be followed by $i - 1$ non-**mktup** instructions, this cost can be amortized across the following

instructions, such that we can assign *every* instruction a (conservative) upper bound on heap-allocation time of $a + a_0$.

Now, Lemma 3.1 says that each instruction can execute at most once. Suppose we have a packet with n instructions and p bytes of payload (both stack and heap together). All SNAP instructions are a fixed size, call it m_i. Of the n instructions, suppose that i of them are **demux** or **demuxi** ($0 \leq i \leq 1$) and j of them are **forw**, **forwto**, **send**, or **hop** ($0 \leq j \leq c$). Then, the execution time of this packet is bounded above by:

$$
\begin{aligned}
& (\textit{normal instructions}) + \textbf{demux} + \textbf{send} + \textbf{mktup} \\
= \;& (n - i - j)t_i + i(dp + d_0) + j[s(n + p) + s_0] + n(a + a_0) \\
= \;& n(t_i + js + a + a_0) + p(id + js) - it_i - jt_i + id_0 + js_0 \\
\leq \;& n(t_i + id + js + a + a_0) + p(t_i + id + js + a + a_0) - it_i - jt_i + id_0 + js_0 \\
= \;& (t_i + id + js + a + a_0)(n + p) - t_i(i + j) + id_0 + js_0 \\
\leq \;& (t_i + id + js + a)(nm_i + p) - t_i(i + j) + d_0 + cs_0 \\
\in \;& O(nm_i + p).
\end{aligned}
$$

So therefore, the execution time of the packet is linearly bounded in its length (the size of the packet being $nm_i + p$ bytes). □

Theorem 3.2 (Memory Safety). *One execution of a SNAP program on a given node will require memory that is at most linearly proportional to the packet's length.*

Proof. First, we begin by noting that except for **mktup**, every SNAP instruction increases the stack size by at most one value (although in some cases, the stack size remains constant or decreases). Let the size of a stack value be m_s bytes.

Now, the instruction **mktup** i allocates some fixed amount of memory h corresponding to an object header in the heap, plus im_s bytes for the new tuple. However, it also consumes i stack values, so the size of the stack is decreased by im_s bytes. Thus, **mktup** effectively allocates h bytes in the packet.

Again, Lemma 3.1 says that each instruction executes at most once. Suppose we have a packet with n instructions and p bytes of payload. Of these instructions, suppose that

k of them are **mktup**. Let the size of an instruction be m_i bytes. Then, the amount of memory the program can allocate on the stack or heap is bounded above by:

$$(n-k)m_s + kh = nm_s + k(h - m_s).$$

Since a node must also keep the packet itself in memory, the overall memory usage for the lifetime of the packet is bounded above by:

$$\begin{aligned}
& nm_s + k(h - m_s) + nm_i + p \\
\leq\ & nm_s + nh + nm_i + p \\
=\ & (nm_i + p) + n(m_s + h) \\
\leq\ & (nm_i + p) + nm_i(m_s + h) + p(m_s + h) \\
=\ & (m_s + h + 1)(nm_i + p) \\
\in\ & O(nm_i + p).
\end{aligned}$$

So, the memory usage of the packet is linearly bounded in its length (the size of the packet being $nm_i + p$ bytes). □

Theorem 3.3 (Global Predictability). *The number of local packet executions that can be caused by a SNAP packet or any of its descendents is strictly bounded by the resource bound of the original packet.*

Proof. This follows straightforwardly from SNAP's conservation of resource bound property. Because a packet execution can only be initiated by sending a packet, and each network hop consumes 1 RB, the resource bound of the original packet is a strict upper bound on the number of network hops that can be effected by the packet or any of the descendents it might spawn. □

3.3 Broader Implications

Local resource predictability (in particular, being able to have strict bounds on CPU and memory usage for a single packet execution) has clear benefits. First, it provides

traction for process schedulers, particularly those for real-time operating systems [RLLS98, BBDS97], as deadlines are easily calculated. Second, it provides for very easy admission control: a simple calculation suffices for a router to determine if it has enough available resources to process this packet. If it does not, the router may always treat it as a regular passive packet, thereby providing a "best effort" approach to active processing and allowing for graceful degradation of service, as suggested by Hjálmtýsson and Bhattacharjee [HB99].

Now, the global predictability we have established in Theorem 3.2 is also useful. For example, consider the case of a SNAP packet entering a given autonomous domain (AD). The AD's ingress routers may make a simple computation to bound the amount of work the packet could do if it were admitted to the network. In fact, research of this sort [SN02] is currently underway using our model of active packet execution for traffic billing.

Although this global bound is easy to calculate, it may not be particularly helpful: for example, if a packet arrives with a very large resource bound, the maximum global resource usage may be above what the AD is willing to tolerate. In this case, the AD can always truncate the incoming packet's RB to an acceptable level. There is a clear tradeoff here, however: setting this maximum to a low level provides strong and useful assurances, but hampers the flexibility of what the active packets can accomplish (for example, broad multicast trees in the style of the active video-on-demand system we present later may not be possible). On the other hand, setting this maximum high permits a wide range of applications from the active packets, but does not necessarily provide strong resource guarantees. Fortunately, this "incoming RB limit" provides a convenient tuning knob for AD administrators.

One hybrid approach would involve putting a severe limit on the incoming RB of a SNAP packet, but making "application gateways" at ingress points available. Incoming SNAP packets would invoke an application-specific service on the ingress router and then terminate. The service, in turn, would launch a new packet (or set of packets) to traverse the rest of the AD. The advantage to this approach is that because these new packets originate within the AD, their code and its behavior is known, and they can be confidently provisioned with higher resource bounds.

It is important to note that the results in this chapter apply to the resource usage of *one packet*. While an important first step, we do not consider the resource usage of flows of packets. In particular, we do not address multiplexing a single node's resources, nor do we address the rate at which users can inject multiple packets. For the first case, we have shown that a local packet execution is linearly proportional to its length (as is the case for IPv4 packets). Therefore, it seems likely that approaches like RCANE [Men99] and Nemesis [BBDS97] that separate multiple flows of packets can still be applied to SNAP packets.

The rate at which multiple packets can be introduced by a user is an open problem in the Internet today. However, it seems that approaches like DiffServ [BBC+98] that perform marking and metering of flows coming into an AD would be applicable here: a service-level agreement (SLA) is negotiated between the AD and its neighbor; any packets exceeding the rate specified by the SLA are marked as "out of profile" and are treated in a best effort fashion—if there are spare resources, they are processed, but if there are not, they are dropped. Billing schemes [SN02] should also be quite applicable here, especially since the resources packets can consume are straightforwardly calculable.

3.4 Recap

At this point, we have established that SNAP meets the resource predictability criteria of our practical active packet model. However, we have not yet dealt with the resource usage of any node-resident services that might be called via **calls**. Furthermore, we would like to understand what would happen to SNAP's resource predictability were we to add new instructions to the language. In the next chapter, we develop a more general resource usage model that will give us insight into these questions.

Chapter 4

Incorporating Services

In this chapter, we develop a more general model of resource usage that includes services. Although the preceding chapter was sufficient for our purposes of showing that SNAP has practical safety properties, it is not really the whole picture. Recall that the active packet approach to active networking assumes a two-level architecture: active packets provide control logic and "glue" for node-resident services. In fact, active packets would not really be practical without services. Our proofs in Chapter 3 factored out service calls, and yet a packet language is not really safe if it can call unsafe node services.

We are mainly concerned with the resource usage properties of SNAP services; robustness and isolation issues can be handled in ways used by current network protocol stacks. This is a reasonable assumption, as the code for services is node-resident and really can be properly viewed as a system call or a regular network protocol with respect to these security concerns. What really interests us is how service calls can affect the resource predictability properties established for SNAP in the previous chapter.

The approach we take here is to look at the resource usage of a "normal" SNAP packet and then consider the resource usage of SNAP programs calling a given service. We will see that there are a variety of methods we can apply to deal with problematic services. First, we may want to place a constant bound on the number of times a service can be called to help maintain our local resource properties. Second, we may want to "charge" a packet for using a service by deducting some amount of its resource bound to maintain our global predictability. Finally, there are some cases where we cannot easily guarantee resource predictability, but we would still like to allow access in some cases; in these instances, we can resort to authentication-based access control.

Finally, we examine *resident state* services in more detail. These services allow packets to leave state behind for other packets to read. Because this is an important service, we work through calculations that show how to appropriately charge a packet's resource bound for the use of this service.

4.1 Assumptions and Definitions

In the previous chapter, we saw that a SNAP packet's local execution runs in time linearly proportional to its length. This guarantee is a consequence of two facts: first, that each instruction executes at most once, and second, that each instruction executes in (possibly amortized) constant time and space. In this chapter, we will make the simplifying assumption that each instruction in fact executes in constant time and space, allowing us to omit any bookkeeping for amortization. We will make use of the following definitions:

- n is the number of instructions in a given packet
- p is the initial data payload (stack plus heap) in the packet, expressed in number of instructions (since all instructions are the same size, instruction count is a measure of space)
- t_i is the maximum time necessary to execute one instruction, in cycles
- m_i is the amount of memory one instruction can allocate on the stack or in the heap, expressed in number of instructions

As a final note before we begin our calculations, one might ask why we use asymptotic complexity notation like $O(n)$ when referring to the relationship between execution time or memory usage and program length n. The maximum transmission unit (MTU) of an underlying network technology places a hard upper bound on the size of a program, so there is really a constant upper bound on the amount of resources a local packet execution can use. However, this constant upper bound would not help us understand the resource usage of non-maximally sized packets as well as our asymptotic complexity bounds do. Furthermore, if the current trend towards increasing MTU sizes continues [Cur98], it will be useful to understand the implications of running SNAP in those settings as well. The

use of asymptotic complexity notation allows us to draw conclusions that are independent of a given network MTU size.

4.2 Normal Resource Usage

We begin by examining the processor time and memory usage of a single SNAP packet execution. Due to the forward-branch restriction and Lemma 3.1, we know that a given packet will run for at most nt_i cycles.

With respect to memory, recall that a packet's state can be completely recycled after the packet has finished executing, as packets are, by nature, ephemeral. So from a long-term point of view, the memory consumption of a packet on a node is zero (especially since we have just observed that a packet is guaranteed to terminate within a predictable amount of time). Nonetheless, a packet *does* consume memory while it is processing, so we need to measure this in some way; the unit of concern is a space-time product (*e.g.*, byte-cycles, or, in our case, instruction-cycles since we are expressing space in terms of instruction count). In other words, we want to know *how much* memory a packet will use and for *how long*.

Suppose a packet arrives at time $t = 0$. Since each instruction can allocate at most m_i memory, the upper bound on the memory footprint of the packet can be expressed as in the following series:

time t	memory footprint (in instructions)
0	$n + p$
t_i	$n + p + m_i$
$2t_i$	$n + p + 2m_i$
$3t_i$	$n + p + 3m_i$
$\ldots$	$\ldots$
kt_i	$n + p + km_i$

Now, the node must hold each of the above footprints for a duration of t_i, so over the lifetime of the packet, the overall space-time resource usage for basic SNAP, RU_{base}, is

then bounded by:

$$\begin{aligned} RU_{base} &\leq \sum_{k=0}^{n-1}[(n+p+km_i)t_i] \\ &= t_i\sum_{k=0}^{n-1}p + t_i\sum_{k=0}^{n-1}nn + t_i\sum_{k=0}^{n-1}km_i \\ &= npt_i + t_in^2 + \frac{1}{2}t_im_in(n-1) \\ &= t_i(\frac{1}{2}m_i+1)n^2 + t_inp - \frac{1}{2}m_it_in \\ &\in O(n^2+np). \end{aligned}$$

Now, with this baseline, we can consider what happens if we relax some assumptions about the SNAP execution model.

4.3 Adding New Services

Before we discuss adding new services, we point out that there is really an isomorphism between new services and new opcodes in the SNAP language itself. For any given operation, it could equally easily be implemented as a service or as a new SNAP opcode, yet the resource considerations would be the same either way. Since we imagine that new services will be added more frequently than new versions of SNAP will be deployed, we will make a natural simplification for the rest of this chapter and just refer to new services.

Our first observation is that a service that runs in constant time and space and does not involve any packet sends or an increase in the packet's RB does not change the overall behavior of the language in terms of predictability (although it may change the value of either m_i or t_i). Examples of such services include additional arithmetic operators and many node query operations (*e.g.*, "how many interfaces do you have?").

Therefore, we should consider new services that violate safety constraints for local execution time, local memory usage, and/or the global number of packet executions. We can explore different combinations of "unsafeness," depicted pictorially in Figure 4.1.

We should point out at this time that we may not be able to guarantee that a certain service is safe, yet it may be so useful that we wish to have it available anyway. For example,

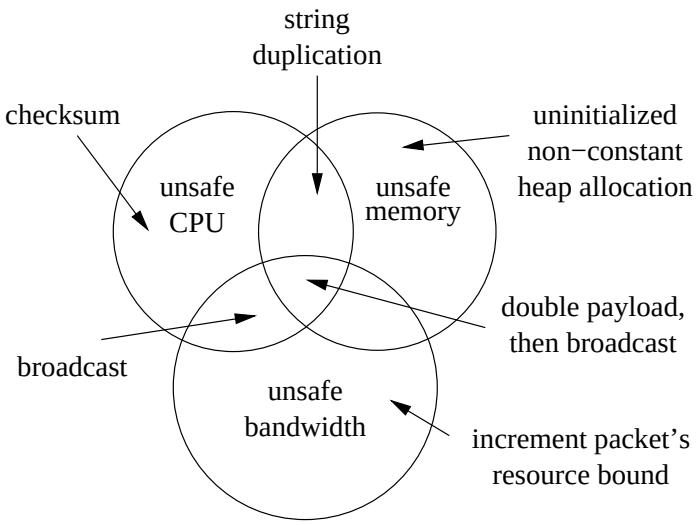

Figure 4.1: Different classes of unsafe services, with examples

consider a "resource bound refueling" service that would allow a packet to recharge itself mid-network. As we will see below, such a service cannot be made safe with respect to our model. However, this simply means we cannot allow *unauthenticated* access to such a service.

To provide authenticated access[1], we can take the approach of PLAN [HKS02] and ALIEN/SANE [Ale98, AAKS98]. Here, the service namespace available to a packet is determined by the privileges with which it runs. There is some minimal set of "safe" services available to all packets without authentication. A packet can carry cryptographic credentials that can be presented to the node in exchange for an expanded service namespace—access to additional services. This style of *namespace security* allows packets to only pay the costs of cryptographic authentication on an as-needed basis. Although we have not implemented this service for SNAP, it would be straightforward: the credentials could be carried in the packet's heap as a byte array and then handed to an "`auth`" service that would side-effect the service namespace for the packet.

[1]We would expect that we would need an authentication infrastructure for some sensitive node services anyway (*e.g.*, altering the routing table), regardless of their resource usage properties.

4.3.1 Unsafe Execution Time

First we consider a service that requires unsafe (meaning, non-constant) CPU time; a good example of this is checksumming an entire packet (code and data). A checksum implementation requires only constant memory, but requires time $O(n+p)$ to execute. In essence, this takes the same time as forwarding another copy of the packet and executing it (also $O(n+p)$), so one possibility is that we simply charge 1 RB for each use of this operation. Of course, the checksum consumes only a constant amount of memory and no network bandwidth, so modeling it as another packet execution is a conservative approximation.

While this solution preserves the global resource usage of the packet, we have lost the linear bound on local execution time. Suppose we had n calls to the checksum service; then the bound on execution is $O(n^2+np)$ rather than $O(n+p)$ as we would like. Therefore, it seems that we need to put a constant cap on the number of times the checksum service can be called, much as we have already done for network sends in SNAP. Interestingly enough, this constant cap technique would work for *any* operation that runs in time linear in the length of the packet. For practicality's sake, however, the constant cap must be tuned to keep the upper bound on execution time low (an upper bound of 2 hours per packet, even though a constant, would not ensure a very high-throughput router).

This does present an interesting design decision, however. We might be willing to permit long local execution times as long as global resource predictability was ensured (for example, simply allowing backward branches and charging 1 RB per use). This may admit a greater possibility of denial-of-service attacks, however, as a single node could get flooded with long-looping packets. Fortunately, as we saw in the previous chapter, autonomous domains can set a "maximum ingress RB" to balance the amount of looping permitted against the tightness of the global resource bounds.

4.3.2 Unsafe Global Resource Usage

We will begin by noting that any service that causes a network send may or may not allow violations of global predictability, but it certainly also falls under the category of unsafe execution time (as a network send must copy the packet out over the network interface).

Thus, some aspects of unsafe network primitives will be covered by constant-cap techniques such as those of the previous subsection.

The obvious exception is a service that sends fixed-size, non-active packets (*e.g.*, acknowledgments or simple UDP datagrams). In these cases, the most straightforward solution is to force the active packet to directly donate its resource bound to become the TTL of the new passive packet, thereby preserving global bandwidth usage. However, as this service requires constant time and space, we can permit an active packet to call it as many times as it likes (or until it runs out of RB, whichever comes first).

Recall that our global predictability property states that the number of network transmissions (and therefore, number of packet executions) is bounded by the RB of the initial parent. This property is primarily guaranteed in SNAP by requiring parent packets to donate some of their RB to their children—conservation of resource bound. Therefore, any service that allows a packet to increase the amount of resource bound attributed to it is potentially globally unsafe.

At this point, let us propose adding multicast and/or broadcast abilities to SNAP. One possibility is that we just permit **forw** and the other network sends to use multicast or broadcast IP addresses as their targets. This is clearly unsafe, as this creates multiple copies of the sent packet, all with a copy of the child's resource bound. Therefore, the total resource bound in the system at this step would increase by a factor equal to the number of listening nodes on the receiving end of the multicast or broadcast. This is clearly undesirable in our current model[2].

Note that even were we to take a fairly draconian stance, and allow broadcast only if the parent's RB is set to zero (to prevent further transmissions after the broadcast) and the outgoing packet's RB is set to 1 (to allow it to make one hop), we are still in trouble: suppose there are 1000 nodes on a segment ready to receive the broadcast; then the resource bound present in the system *still* increases via the broadcast. Therefore we must conclude that general broadcast and multicast primitives in SNAP are inherently unsafe. If, however, we know the number of receivers for a given multicast transmission, we can safely divide the parent's resource bound by the number of receivers, and then multicast it. Indeed, this

[2]In Chapter 7 we discuss a scheme [WW02] that might make this acceptable.

is the approach we take in the multicast framework for our video-on-demand application described in Chapter 6, although we program the multicast algorithm in SNAP itself rather than encapsulating the whole protocol in a service implementation.

One other way that we might increase the amount of RB in a system is by increasing a given packet's resource bound. Without any other adjustments, however, such a primitive is unsafe, as it would allow a packet to execute forever, by increasing its resource bound (by at least 1), then forwarding itself back to the same node, thus consuming 1 RB and leaving the packet with its original RB. Therefore, as we mentioned above, we must disallow unauthenticated access to such "RB refueling" services.

4.3.3 Unsafe Memory Usage

As we saw in our derivation of RU_{base}, resource predictability depends not only on how much memory is allocated, but also how long that memory is occupied on a router. Normal SNAP instructions allocate only a constant amount of stack or heap memory each, and all of a packet's state is reclaimed once the packet program terminates. Services that allocate a non-constant amount of packet memory (in either the code, stack, or heap segments) and memory that outlives the packet that allocated it (node-resident state) both violate these guarantees. Because node-resident state is an important service in its own right, we will treat it specially in the following section.

Again, most services that are memory unsafe are also CPU unsafe if the newly-allocated memory is eagerly initialized; if we allocate a non-constant amount of memory, we must do a non-constant amount of work to initialize it. Therefore, as above, we will likely require constant caps on the number of times such a service could be called, or will require that calls to such a service be followed by a proportional number of safe instructions (as we did with **mktup**) in order to retain safe local execution bounds. In either case, we can again be sure that the memory usage of the packet is still linear in the packet program's length.

Just for completeness, consider services that allocate a non-constant amount of uninitialized stack or heap data. Suppose the service allocates some amount of stack or heap memory represented by the function $f(n, p) > 1$. Now, the resulting packet would have the same execution time and memory allocation behavior as the original packet (since the

code segment has not changed), but forwarding such a packet will transmit $n+p+f(n,p)$ instructions, rather than $n+p$. Note that:

$$n+p+f(n,p) < nf(n,p)+pf(n,p) = f(n,p)(n+p).$$

So sending $f(n,p)$ copies of the original packet is a (conservative) upper bound on the bandwidth consumed by the new packet; therefore, dividing the packet's resource bound by $f(n,p)$ will guarantee that the new packet's global resource usage is less than or equal to the original packet's[3].

Finally, we can consider services that modify the code portion of a packet. Clearly, any service that does not increase the code size should be safe (and needs merely to be charged for modifying the instructions themselves: for example, similar to checksumming as above). Indeed, this is highlighted by the fact that transmission errors could make arbitrary changes to instructions, but a safe SNAP implementation will ensure that nothing untoward happens (robustness).

Let us consider services that lengthen the code segment. Again, as above, suppose the new primitive increases the code segment by some function $g(n,p) > 1$. Now, the new packet will have more instructions than the original, but if we do not allow newly-allocated instructions to be executed on a node (as that would allow a packet to exceed its local execution time bound), then we really only have to account for the added cost of network transmission; the resulting packet will still satisfy linear bounds on execution time on any other nodes it reaches. So, by the argument above for uninitialized allocation, we can simply divide the current packet's RB by $g(n,p)$ to account for the increased transmission costs.

4.4 Resident State

Resident state (namely, some state that outlives the execution of the packet that allocates it) is an extremely useful abstraction for implementing active applications, as shared variables are a convenient method for inter-packet communication. The first thing to notice

[3]Note that if $f(n,p) = 1$ then the packet's RB is unchanged, which is consistent with our handling of regular SNAP instructions.

is that if this resident state is never reclaimed, we have infinite space-time resource usage. Therefore, this state *must* be reclaimed at some point in order to maintain the strict predictability the SNAP model gives us. One natural way to do this is to implement *soft state* that "times out"—a notion commonly used in current network protocols.

In the rest of this section, we will carry out calculations that show how to accurately charge against a packet's RB for the use of the resident-state service. Generally, we compute the "extra load" over and above executing the packet itself that is placed on the node by having to hold the resident state, and then charge a corresponding amount of RB. The reader not interested in the details of the calculations is advised to skip ahead to Section 4.5.

Suppose we were to add a service to SNAP that would allocate m_i units (relative to the size of an instruction) of resident state that would time out after time t_o. Now consider a packet program consisting of n of these allocations. Then, following the same formula as before, the space-time usage would be:

$$\begin{aligned}
\sum_{k=0}^{n-1}[(n+p)t_i] + \sum_{k=0}^{n-1} k m_i t_o &= t_i \sum_{k=0}^{n-1} p + t_i \sum_{k=0}^{n-1} n + t_o m_i \sum_{k=0}^{n-1} k \\
&= npt_i + t_i n^2 + \frac{1}{2} t_o m_i n(n-1) \\
&= (t_i + \frac{1}{2} t_o m_i) n^2 + npt_i - \frac{1}{2} t_o m_i n \\
&\in O(n^2 + np).
\end{aligned}$$

So, resident state with finite timeouts requires no more node resources than normal packet execution, from an asymptotic complexity point of view. However, we *are* concerned with the constants here. So what we really want to know is the additional load placed on a node, expressed as a ratio r of the space-time usage with resident state to the normal space-time usage:

$$\begin{aligned}
r &= \lim_{n \to \infty} \frac{(t_i + \frac{1}{2} t_o m_i) n^2 + (pt_i - \frac{1}{2} t_o m_i) n}{t_i (\frac{1}{2} m_i + 1) n^2 + t_i (p - \frac{1}{2} m_i) n} \\
&= \frac{t_i + \frac{1}{2} t_o m_i}{\frac{1}{2} t_i m_i + t_i}
\end{aligned}$$

$$
\begin{aligned}
r &= \frac{2t_i + t_o m_i}{t_i m_i + 2t_i} \\
&= \frac{\frac{t_o}{t_i} m_i + 2}{m_i + 2}.
\end{aligned}
$$

So, a packet that does n resident state allocations is r times worse than a single packet. Another way of stating this is that the resident-state packet with n allocations consumes as many resources as r "normal" packets. Therefore, a simple way to equalize things is to charge r/n resource bound units per resident-state allocation.

4.5 Practical Considerations

The above discussion generally depends upon knowing the running times or memory allocation behavior of services beforehand. Unfortunately, it is undecidable to determine this for an arbitrary piece of software[4], so we would not be able to deploy unauthenticated service code in our network, as we would have no way to guarantee resource safety.

Therefore, we must rely on some resource usage claim associated with each service, whether a proof of the claim is provided to a node or the claim is merely asserted by a trusted authority. Fortunately, the programming languages community offers multiple techniques for asserting and estimating properties about the resource usage of programs [CW00, NL97, Hof99, HP99, HPS96, RG94]. Indeed, it seems likely that a proof-carrying code [Nec97] approach would work well here; a service implementation could arrive with a verifiable proof of its resource usage properties, and a node could then ensure that the appropriate safety measures are taken (either adjustments to a packet's resource bound, or establishing per-packet constant caps on the number of times a service could be called). Alternatively, some of the above programming language techniques could be applied in a domain-specific service language (rather than a general purpose language, which many of the techniques attempt to handle) such that well-formed (or well-typed) programs in the new language would be guaranteed to be resource safe services.

[4] This would require solving the halting problem.

4.6 Actual Services

Just to complete our discussion, we list below the services used in the development of the example applications presented in Chapter 6.

- **getld** returns the running CPU load average of the node. This service runs in constant time and space.
- **qlen** takes as an argument an integer identifying one of the node's network interfaces, and returns the length of the outgoing packet queue for that interface. This service operates in constant time and space.
- **put** takes two arguments, a variable name and a value, and stores a piece of resident state on the node. Currently no resource bound is charged for this service, although we could apply the calculations in Section 4.4 above to do so. Resident state also currently does not time out, as this facilitates application debugging (although, as we saw above, timeouts would be necessary for safety in a production implementation of SNAP).
- **get** takes a variable name and retrieves a (previously-stored) piece of resident state. For non-constant-sized pieces of resident state (*e.g.*, strings), we could again charge appropriate amounts of resource bound.
- **ifput** is similar to **put**, but takes an additional argument: an integer identifying one of the node's network interfaces. This allows applications to store interface-specific resident state.
- **ifget** is the analogous interface resident-state retrieval service.

This is a fairly short list of services, especially considering the complexity of the applications involved (most notably the video-on-demand application). This speaks to the usefulness of SNAP itself, but also shows that realistic applications can be built with a modest set of safe services.

Chapter 5

Efficiency

The second step in our demonstration of SNAP's practicality will be to show that it is *efficient*. Our high-level design goal from Chapter 1 was:

- **Efficiency.** Although complex packet programs will likely be more expensive to execute than passive packet headers, it is important that the infrastructure needed to permit more flexibility should not add significant overhead. Specifically, IP-like functionality should be available at IP-like performance. We focus on the very common setting of software routers connected by 100 Mb/s Ethernet links.

In this chapter, we show experimentally that the overhead of our active packet system *itself* is minimal; with SNAP a user would only pay for the extra computation he or she requests. Indeed, the applications we present in the next chapter show how this extra computation can result in improved application performance.

Our experiments in this chapter consider scenarios involving software routers running on commodity PCs and connected by 100 Mb/s Ethernet links. This is an extremely common situation: many enterprise networks, home networks, and research PC clusters run at these speeds. Indeed, when we consider the "onion" model of the Internet we showed in Figure 1.1 in Chapter 1, these speeds would cover all but the optical backbone links at the center. Furthermore, this is already a scenario where existing active packet systems have failed to achieve adequate safety and efficiency at the same time. While a hardware implementation of SNAP in a high-end router would certainly be interesting, it is well beyond the scope of this dissertation.

5.1 Implementation Overview

We have implemented a SNAP VM in C and integrated it with Linux kernel version 2.2.19 as part of RedHat Linux 6.2; it is accessible to user applications by a special socket type. SNAP is currently positioned as a shim layer between the network and transport layers, using the IP Router Alert option [Kat97] to indicate the packets are active. Thus, SNAP-enabled routers can detect SNAP packets and execute them, while legacy "passive" routers can simply forward SNAP packets towards their destination.

We followed four implementation principles:

1. avoid overheads suffered by previous systems
2. minimize the size of SNAP program representations
3. prefer small incremental costs to large initial ones
4. optimize for important common cases

In total, our intention was to keep SNAP's overhead low, especially for common case programs like data delivery and diagnostics.

5.1.1 Packet Format

Our main concerns for SNAP's packet format are compactness and avoiding marshalling overhead; previous systems have suffered from overly large program representations [Ale98] or from high marshalling costs [WGT98, HKM+98b]. Our general approach is to design a wire format for SNAP that will allow a SNAP program to be executed *in-place* in a packet buffer, often avoiding expensive marshalling, unmarshalling, or copying.

We achieve in-place execution with the packet format shown in Figure 5.1. This figure shows both the SNAP packet format and the encapsulating IPv4 header (including the Router Alert option). We actually re-use those parts of the IPv4 header depicted, most notably the source and destination addresses. The time-to-live (TTL) field is re-used to provide SNAP's resource bound, and the protocol field is set to indicate that the IP payload contains a SNAP packet.

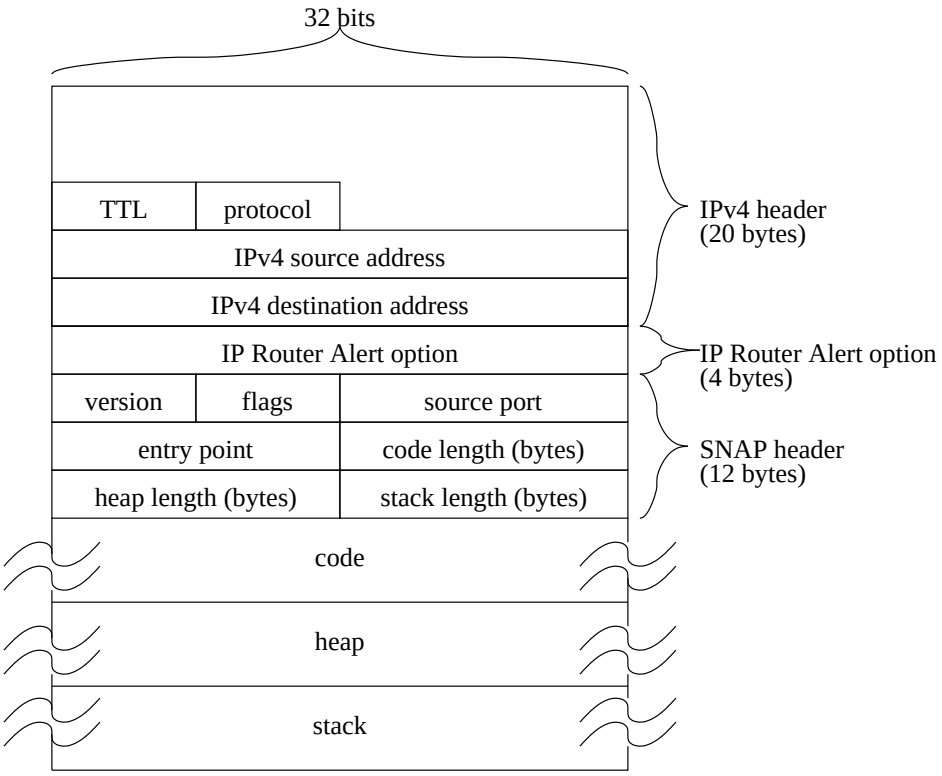

Figure 5.1: SNAP packet format

The SNAP header itself contains version and flag fields (not currently used) and a source port address similar in function to a UDP source port field [Pos80]. The remaining four fields in the SNAP header are the ones required for execution. The code length, heap length, and stack length fields define the sizes of the variable length code, heap, and stack segments that follow the SNAP header. The entry point indicates which instruction in the code segment should be the first executed (with the first instruction of the code segment being numbered zero).

Following the SNAP header are the code, heap, and stack segments. The packet is laid out in this order to permit execution in place, without additional copying, as we describe below. As a result, packet execution can begin almost immediately upon arrival, following a very few structural checks (*e.g.*, that the entry point is within the code, that the various lengths do not exceed the buffer size, etc.). This is in contrast to systems like PLAN [HMA+99] and ANTS [Wet99] that do not directly interpret packets' wire format; instead, values must be unmarshalled into corresponding OCaml or Java objects before execution as well as serializing them again before transmission.

5.1.2 Program Representation

The SNAP program is generally represented as follows. The code section consists of an array of uniformly-sized instructions. Stack values are also uniformly-sized, consisting of a tag and a data field. The tag indicates the type, and the data contains the actual value. For values that are too large to fit on the stack (like tuples or byte arrays), the data resides in the heap and is pointed to by the stack value. This pointer is implemented as an offset relative to the base of the heap. This allows the packet to be arbitrarily relocated in memory at a cost of an extra calculation during interpretation. This feature eliminates the large fixed cost of adjusting pointers in the code and stack before execution. Heap objects each contain a header with length and type information.

We implemented this scheme to require as little space as possible. All instructions and stack values are one 32-bit word, and heap objects have a one word header. Stack values are divided into an n-bit tag, and a $(32 - n)$-bit data part. Integer precision is reduced as a result, and network addresses and floating point values must be allocated in the heap.

This works well in some cases, as floats are used infrequently, and integers almost never require high precision in the context of simple packet programs. Unfortunately, addresses are used often, thereby resulting in heap allocation even for fairly simple programs, as we shall see below. Instructions are similarly an n-bit opcode and a $(32 - n)$-bit immediate corresponding to the data part of a stack value. We have 102 distinct instructions in our final encoding; therefore we set n to be 7 bits for tags and opcodes, meaning that our integer precision is 25 bits.

5.1.3 SNAP interpreter

The interpretation of most of the instructions is extremely straightforward, with the exception of **send** and the other network operations, as explained below. The interpreter is constructed as a loop around a large `switch` statement, with one case for each opcode. Most instructions extract arguments from the stack or heap, perform some computation, and then push the result back on the stack.

If the initial packet buffer is sufficiently large, packet execution may occur "in place." That is, with the packet at the front of the buffer, the stack is allowed to grow towards the end of the buffer during execution. Heap allocation takes place within a second heap region, situated at the end of the buffer and growing towards the stack[1]. In our kernel implementation, the buffer we receive from the kernel is not much bigger than the packet itself. If the stack attempts to grow beyond this buffer or if heap allocation is attempted, we copy the packet into a maximally sized buffer (recall from Chapter 3 that we can calculate the maximum amount of memory a program will require, in the absence of unsafe services[2]). This on-demand approach allows us to avoid copying the packet for programs that do minimal stack manipulation or no heap allocation[3].

[1] Although this separate heap area means we must do some copying to get a contiguous buffer for packet sends, an appropriate low-level scatter/gather interface would allow us to avoid this extra copy.

[2] Even in the presence of unsafe services, it is quite likely this "maximum" size will be a good estimate, so we will usually avoid any further copying to resize the buffer.

[3] For example, the ping program described in Chapter 2 simply executes a **forw** on intermediate nodes, thereby avoiding a packet copy. However, on the destination host, the **getsrc** instruction causes a heap allocation, requiring a packet buffer copy.

5.1.4 Implementing send

Send creates a new packet containing its parent's code, subsets of its stack and heap, and some of the parent's resource bound. Creating this new packet presents two difficulties. First, if any heap allocation has taken place, then the parent packet has two heaps that must be consolidated into a single heap in the child packet. Second, we would prefer to include only the portions of the parent packet that will be needed by the child packet. We address both issues by employing a scheme similar to copying garbage collection [Wil92]. This process ensures that only the portions of the two parent heaps that are *reachable* from the child's stack will be copied into the child heap, and it adjusts any heap offsets (whether in the code, stack, or heap) to point to the correct locations in the child heap.

While this approach is general, it is both computationally and memory intensive. Fortunately, we can do better in certain common cases. First, if we have not performed any heap allocation, we can copy the parent heap and stack subset directly to the new packet, without requiring heap offset fix-ups, since the position of heap objects will not have changed. The result is faster packet creation times but potentially larger packets, since any objects that are unreachable will not have been removed.

Even better, if the stack required by the new packet consists of the entire stack of the current packet, we may *reuse* the current packet buffer, requiring only modifications to its header, but only if transmission will occur before further modifications take place. Since a program terminates after executing **forw** or **forwto**, simple routing of active packets may occur without any significant marshalling costs.

5.2 Experimental Setup

Having seen that the SNAP implementation is tailored for efficient execution, we now present experimental evidence that SNAP is efficient enough to be practical. To do so, we examine SNAP's performance compared to IP, for both latency and bandwidth.

Our experiments show that SNAP is competitive with IP: SNAP latencies are at most 7% longer than ICMP for maximum payloads. Furthermore, SNAP's useful throughput over a 100 Mb/s Ethernet link is at least 95% of UDP's for all but the smallest payload

sizes. The highest switching rate for our SNAP router is over 53,400 packets per second, within 96% of the UDP switching rate of 55,800 packets per second.

All experiments were run on 1 GHz Pentium III (Coppermine) systems with 256 MB of RAM. These machines have 32 KB split (16 KB instruction/16 KB data) on-chip first level caches and 256 KB unified on-chip second level caches, rating 402 on SPECint2000. The machines run RedHat Linux 6.2 (kernel version 2.2.19-6.2.7, modified for SNAP support) and are connected by 100 Mb/s Fast Ethernet links. Successive `gettimeofday` calls on these machines can be completed in 445 ns on average; however, our measurements here do not consider significant digits beyond 1 μs.

5.3 Latency Experiments

To compare SNAP's latency to that of IP, we measured the round-trip times of both the SNAP ping program described in Figure 2.1 in Chapter 2 and the standard ICMP ECHO-REQUEST/ECHO-REPLY [Pos81a]. For both tests, we generate a binary packet image: either a SNAP packet (including SNAP header, code, heap including payload, and stack) or an ICMP packet (including ICMP header and payload). We then start a timer, send the packet image on a raw IP socket, wait for a reply, and stop the timer. We collect 101 ping times in this fashion.

For these experiments, we connect 5 machines serially, as shown in Figure 5.2. All pings begin at the machine "hermes," with the ping destinations varying by hop count as shown. We used both the minimum (8 bytes) and maximum (1416 bytes) payloads possible for SNAP without fragmentation. For each SNAP trial, we compare the performance to an ICMP packet with the same payload size. We expect our SNAP overhead to consist of two parts: first, a space overhead due to larger header sizes and having to carry code, and second, an execution overhead due to interpreting the SNAP code. In order to tease these two overheads apart for a given SNAP packet, we also take a measurement of an ICMP packet with the payload padded to result in the same Ethernet frame size as the SNAP packet. In our case, the space overhead for SNAP is 48 bytes, so we compare a SNAP packet carrying n bytes of payload to an ICMP packet carrying $n + 48$ bytes of payload.

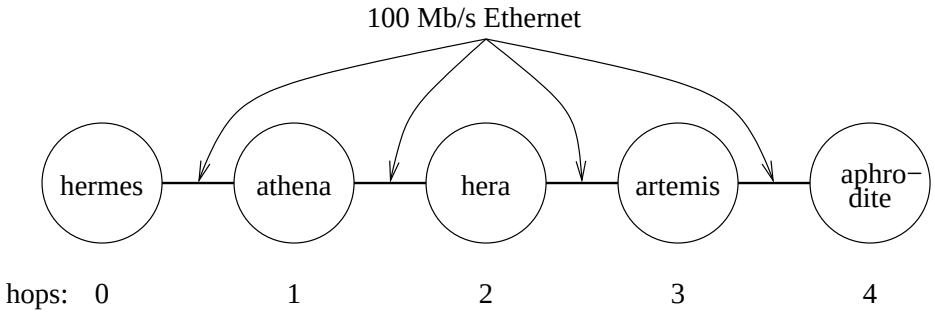

Figure 5.2: Experimental setup for latency benchmarks

For uniformity in the discussion, we will refer to this packet as using the "ICMP+48" protocol to carry n bytes. So, the difference between SNAP and ICMP+48 represents the overhead of SNAP interpretation, while the difference between ICMP+48 and ICMP represents SNAP's space overhead.

Our measurements are shown in Figure 5.3; the y-axis shows latency in microseconds (μs), and the x-axis presents the number of hops, *i.e.*, network links traversed (0 hops is a machine pinging itself). Each point represents the median of 101 trials; error bars indicating the upper and lower quartiles for each measurement are drawn, but are not visible, as all quartiles are within 6 μs of the displayed medians. The one-hop transmission time for minimally-sized ICMP packets is approximately 3 μs (36 bytes at 100 Mb/s plus propagation delay over 10 feet of Ethernet cable). Thus, the overall transmission time for the 1-hop case is 6 μs, or less than 10% of the overall elapsed time, so the differences between ICMP and SNAP for minimal packets are reflective of their processing time differences. The same data is shown in table form in Table 5.1. The general trend is clear: SNAP performs quite respectably compared to ICMP.

For maximally-sized packets, SNAP's latencies are at most 7% slower than those for ICMP, depending on the hop count, while they are as much as 23% slower for minimally-sized packets. When comparing to ICMP+48, we see that a large portion of this overhead is simply due to the larger space overhead of SNAP: SNAP is at most 4% slower than ICMP+48 for maximally-sized packets, and at most 8% slower for minimally-sized packets.

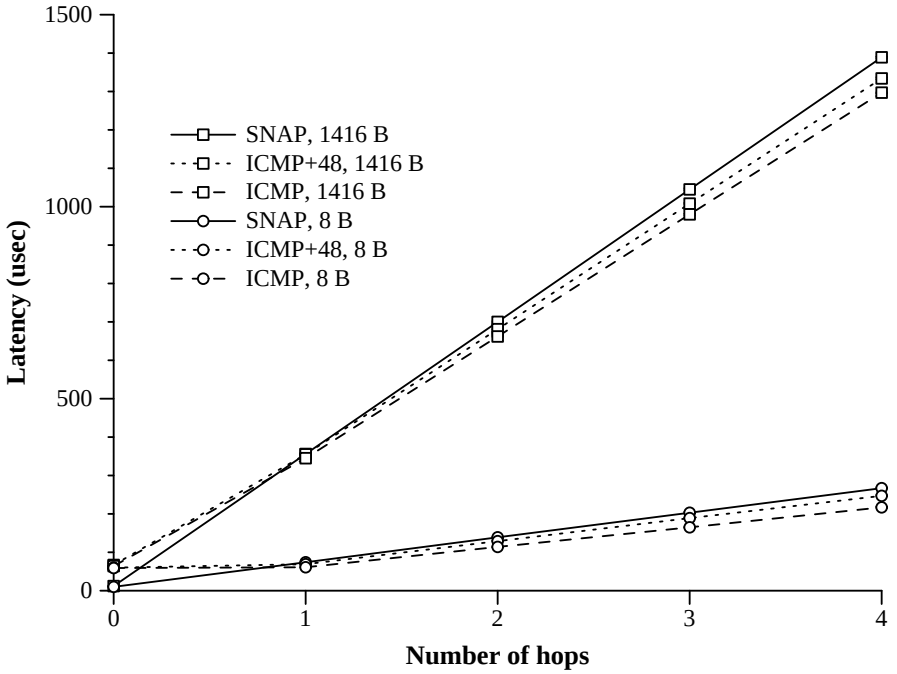

Figure 5.3: Ping latencies: latencies are shown for pings across several hop distances and for minimum and maximum SNAP payloads.

This suggests that optimizations to reduce the space overhead of SNAP programs would be quite useful (for example, as we will discuss in Chapter 7, it would be possible to use 1-byte instead of 4-byte instructions).

For the zero-hop case, SNAP is actually faster than both ICMP and ICMP+48, which is surprising. Further investigation revealed the cause: whereas the SNAP ping program delivers its payload to a transport-layer SNAP socket, the ICMP ping program must receive its ECHO-REPLY over the same raw socket that it uses to send the ECHO-REQUEST. For the zero-hop case, the ping receiver actually receives *both* the outgoing request *and* the returning reply (since both packets are destined to the local machine). The receiver must check the first packet, see that it contains an ECHO-REQUEST, and discard it. Due to the fast turnaround time of the ping, the ECHO-REPLY actually arrives before the ping application finishes discarding the request and is ready to `recvfrom` on the socket again. For the cases where the hop count is greater than 1, the ping application sees only the ECHO-REPLY (as the ECHO-REQUEST has a destination address different from the

	SNAP		ICMP+48		ICMP	
hops	8 B	1416 B	8 B	1416 B	8 B	1416 B
0	10	12	60	67	59	64
1	74	356	69	355	61	345
2	139	700	129	681	114	662
3	203	1045	189	1008	165	980
4	267	1389	247	1334	217	1297

Table 5.1: Ping latencies (same data as in Figure 5.3). All figures shown are in microseconds (μs).

Protocol	Per-packet (μs)	Per-byte (μs)
SNAP	62	0.199
ICMP+48	59	0.190
ICMP	50	0.189

Table 5.2: Per-node switching costs

local machine), so this is not an issue. Therefore, for our further calculations (most notably the linear regressions below), we omit the zero-hop case.

To make a more detailed comparison, we calculated per-byte and per-packet switching costs for traversing a router for both IP and SNAP. We calculated these costs by first using linear regression to find the per-hop cost for each given packet size and then doing a further regression with packet size as the independent variable to find the per-byte and per-packet costs. The results of the per-payload regressions are shown in Figures 5.4–5.9. Note that in all cases there is zero mean error, indicating that we have very clean data.

The slopes of these graphs represent the additional cost per hop for a given protocol and payload size. To see the relationship between added latency and payload size, for each protocol we did a linear regression using payload as the independent variable, and the slopes above as the dependent variables. The y-intercept of this new line is the per-packet cost for the protocol, while the new slope is the per-byte cost for the given protocol. The results are shown in Table 5.2. We see that the fixed cost per packet is about 12 μs (24%) higher for SNAP than for ICMP, while the per-byte cost is only 0.01 μs (5%) higher. Note that these are consistent with our latency measurements: for small payloads, we would

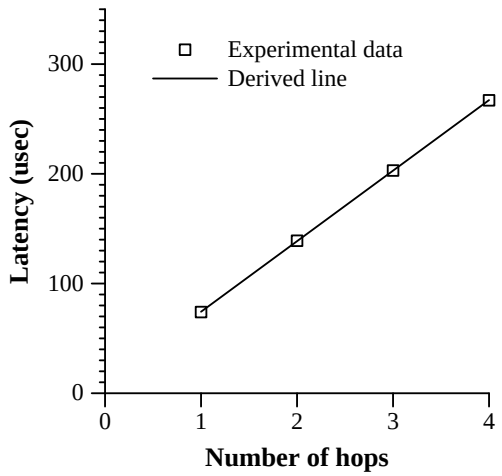

Figure 5.4: Linear regression for SNAP, 8 byte payload. The derived line is $y = 64.30x + 10.00$, with mean error 0.00.

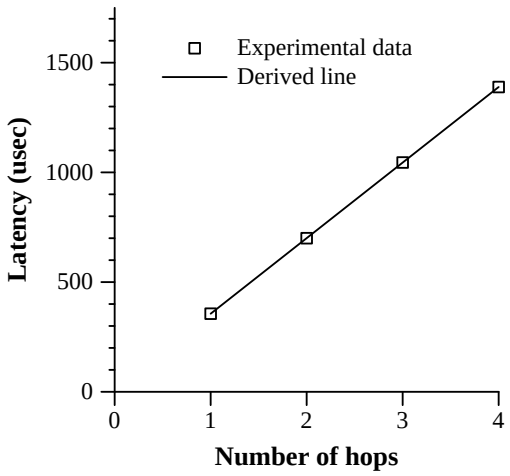

Figure 5.5: Linear regression for SNAP, 1416 byte payload. The derived line is $y = 344.40x + 11.50$, with mean error 0.00.

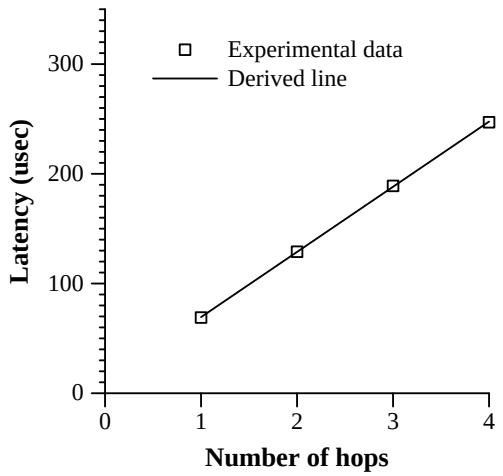

Figure 5.6: Linear regression for ICMP+48, 8 byte payload. The derived line is $y = 59.40x + 10.00$, with mean error 0.00.

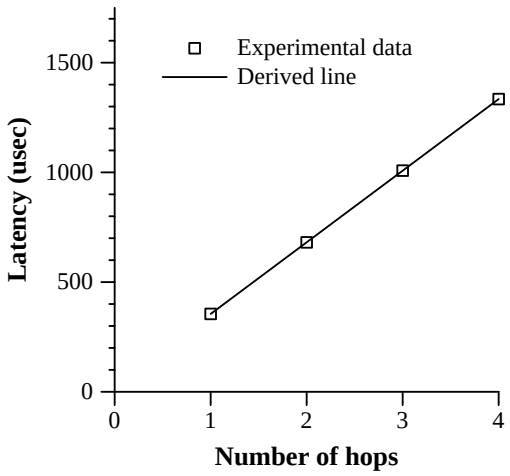

Figure 5.7: Linear regression for ICMP+48, 1416 byte payload. The derived line is $y = 326.40x + 28.50$, with mean error 0.00.

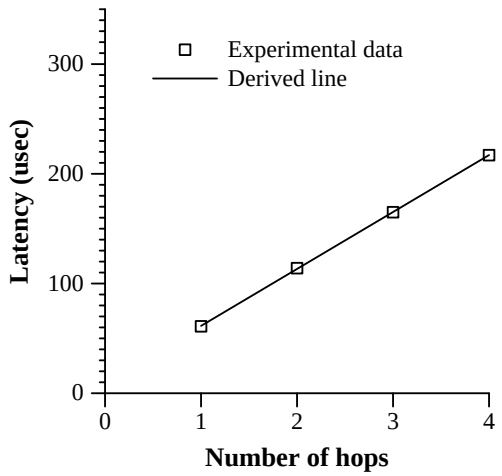

Figure 5.8: Linear regression for ICMP, 8 byte payload. The derived line is $y = 51.90x + 9.50$, with mean error 0.00.

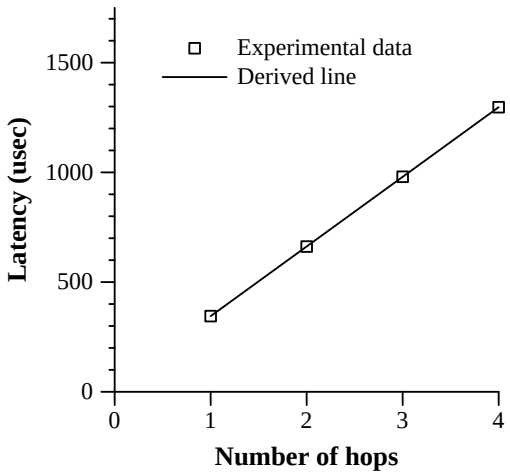

Figure 5.9: Linear regression for ICMP, 1416 byte payload. The derived line is $y = 317.40x + 27.50$, with mean error 0.00.

expect the per-packet fixed costs to dominate, and indeed, SNAP was 23% slower than ICMP. Similarly, for the larger payloads, we would expect the per-byte costs to dominate, and again, SNAP was 7% slower than ICMP.

Now, by comparing to the ICMP+48 derived costs, we can see that SNAP's per-packet cost is only 3 μs (5%) higher, and the per-byte costs are 0.009 μs (less than 5%) higher. The higher per-packet cost are basically representative of SNAP's execution overheads: the cost of executing the **forw** instruction versus the basic forwarding of IP packets for ICMP, as well as the cost of executing the "turnaround" portion of the SNAP ping program versus the ICMP protocol implementation of the Linux kernel (receiving the ECHO-REQUEST, changing the type to an ECHO-REPLY, and sending the packet back). The higher per-byte costs can basically be attributed to the fact that our SNAP interpreter must copy the entire packet buffer on the turnaround host, due to the heap allocation of the source address. As we discuss in future work (Chapter 7), these overheads could be avoided with a tagless stack value representation, allowing floating point numbers and IPv4 addresses to reside unboxed on the stack.

5.4 Bandwidth Measurements

To measure internal switching bandwidth, we compared SNAP's equivalent of UDP (**forw** followed by **demux**) to regular UDP [Pos80] packets with checksumming disabled. We performed a series of measurements on a three-machine configuration, varying the payload from the minimal (8 bytes) to the maximal (1440 bytes) sizes permitted by SNAP. We used the same sending program for both SNAP and UDP, modified as appropriate to either build binary SNAP packets or binary UDP packets. Both cases send the resulting packet over a raw IP socket. Our program also contains a configurable delay loop to evenly space out packets. For our benchmarks, we tuned this delay so that we lost fewer than 0.1% of the packets sent. In this way, we got a realistic perspective on the steady-state throughput supportable by our SNAP router (if the delay was too small, the router could not keep up, but if it was too high, the router would not be taxed). For each payload

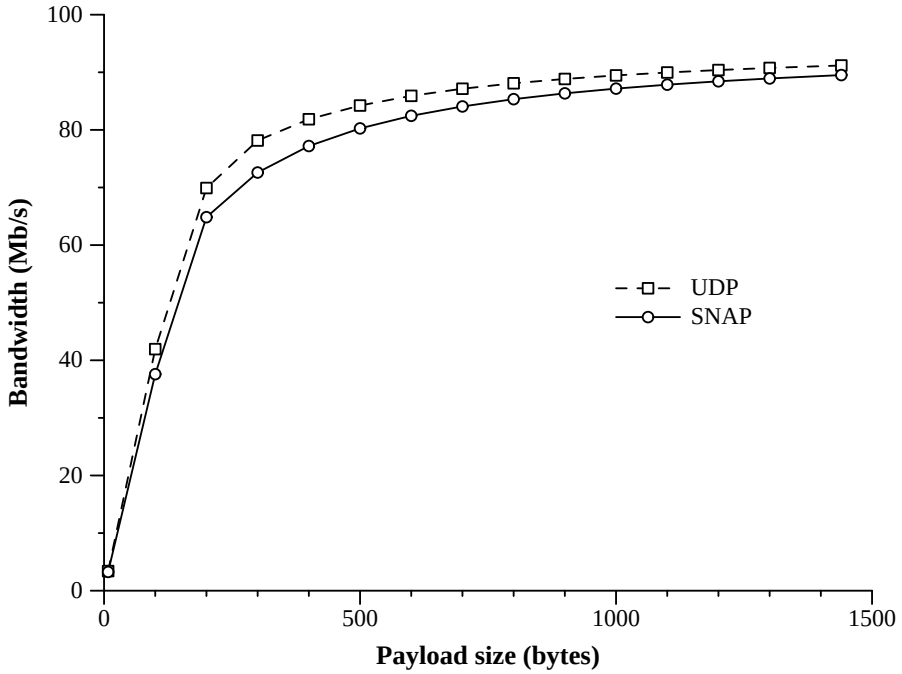

Figure 5.10: Throughput measurements: basic SNAP delivery compared to UDP (without checksums) for a variety of payloads.

size, we sent enough packets to guarantee that the overall transmission took longer than 20 seconds, again to ensure that we were looking at a faithful steady-state situation.

The space overheads are as follows: UDP contains simply an 8 byte header, whereas SNAP contains 12 bytes of header, 8 bytes of code, 8 bytes of stack, and 4 bytes of heap object header, for a total of 32 bytes, or 24 bytes more than UDP.

The results are shown in Figure 5.10 and presented in tabular form in Table 5.3. The y-axis plots throughput in Mb/s (based on payload), while the x-axis plots the payload size in bytes. The numbers here are taken by summing the total transmitted payload bytes over 10 trials, and dividing by the total elapsed time, as per Jain [Jai91]. The same experiments are represented in Figure 5.11 in terms of packets per second.

Here, the result is again clear: SNAP performs quite favorably compared to UDP. SNAP achieves at least 95% of the bandwidth of UDP for all payload values except for 100 bytes and 200 bytes. For these, SNAP achieves 90% and 93% of the corresponding UDP bandwidth for the given payload. Considering that the space overhead of the SNAP

payload (bytes)	throughput (Mb/s)	
	SNAP	UDP
8	3.26	3.41
100	37.59	41.94
200	64.85	69.91
300	72.60	78.16
400	77.20	81.85
500	80.26	84.23
600	82.45	85.91
700	84.07	87.14
800	85.33	88.09
900	86.34	88.84
1000	87.17	89.46
1100	87.85	89.96
1200	88.43	90.39
1300	88.93	90.75
1440	89.52	91.18

Table 5.3: The same throughput experiment depicted in Figure 5.10, but in tabular form.

delivery program is 24 bytes (or, 15% of the total Ethernet frame size for the 100 byte payload, 10% of the total Ethernet frame size for the 200 byte payload), these are quite respectable numbers. We would expect to see bandwidth improvements resulting from a reduction of SNAP's space usage in the packet.

5.5 Recap

In this chapter, we described an implementation of the SNAP model and showed experimentally that it was efficient. More specifically, we established that IP-like functionality could be provided at IP-like performance: the overhead of the SNAP system itself is small. Thus, SNAP is efficient enough to be deployed wherever software routers are currently practical.

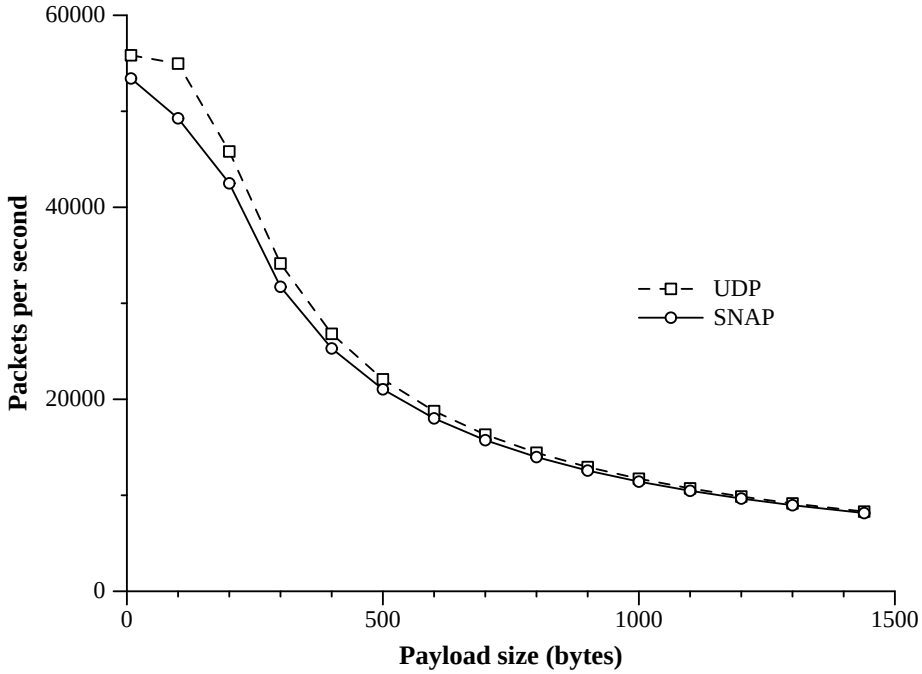

Figure 5.11: Throughput in packets per second: basic SNAP delivery compared to UDP (without checksums) for a variety of payloads.

Chapter 6

Flexibility

The last step in our demonstration of SNAP's practicality will be to show that it is *flexible*. Recall that our goal here is:

- **Flexibility.** It should be possible to express a wide range of useful and interesting active applications with SNAP. In particular, SNAP should offer significantly more flexibility than traditional passive packet schemes.

We take two approaches to demonstrate this property. First, we demonstrate that existing active packet applications can be encoded in SNAP. Second, we present new active applications that are made possible by SNAP's excellent efficiency.

6.1 Relating SNAP's Flexibility to Previous Work

First, we suggest that we can use SNAP to implement a variety of applications already published in the literature (given an appropriate set of services). The first piece of supporting evidence is a PLAN-to-SNAP compiler developed by Michael Hicks [HMN01].

6.1.1 A PLAN-to-SNAP Compiler

Most parts of PLAN are straightforwardly translatable, in no small part because SNAP derives much of its execution model and primitives from PLAN. However, the compiler must use some unusual compilation strategies to account for the fact that SNAP does not have backward branches; in particular, function calls and loops (both of which are present in PLAN) would normally involve the use of backward branches in translated

code. For function calls, the compiler inlines function bodies at call sites to avoid using backward branches. As an optional optimization for this process, the compiler can perform a topological sort on the basic block call graph to duplicate as little code as possible during inlining. This topological sort ensures that if one basic block calls another, the callee will be located after the caller in the code segment (and therefore the control transfer will require a forward branch).

For loops, the compiler has a tunable parameter that permits loop bodies to be unrolled a certain number of times, again avoiding the use of backward branches. However, in cases where avoiding backward branches results in too much code bloat from inlining or loop unrolling, backward branches can be emulated by sending a copy of the current packet back to the same host. We will explore this particular strategy in more detail in our discussion of our video-on-demand application below.

The PLAN-to-SNAP compiler can compile all 70 of the sample PLAN programs that come with the PLANet distribution. Of these, 55 do not involve looping (*i.e.*, the PLAN `fold` construct) at all. For illustration, we chose ten programs from the PLANet distribution [HMA+99]; they represent a variety of different networking tasks, from simple payload delivery (`deliver`) to information gathering (`devinfo, getNeighbors, getRoutes`) to multicast (`multiprint`) to simple network diagnostics (`ping`, `traceroute`). Of these, only `devinfo` and `multiprint` involve looping via `fold`.

Figure 6.1 contrasts the resulting packet sizes for the native PLAN wire format for each program with the packet size for the corresponding SNAP program produced by our compiler. Generally, the resulting SNAP programs for straight-line execution are 30% smaller than their PLAN equivalents. For two of the programs, `devinfo` and `multiprint`, the resulting SNAP programs are actually larger than the PLAN originals. A closer analysis of these pathological cases reveals a great deal about the importance of various optimizations in our compiler.

One first order effect is code bloat from loop unrolling; both programs iterate over all devices present on a given node. By default, the compiler unrolls `fold` five times (with further iterations handled by looping back with **send**). To understand the effect of unrolling, we parameterized the amount of unrolling the compiler does for loops and set

	size (B)		Ratio
Program	PLAN	SNAP	(SNAP/PLAN)
`deliver`	406	284	0.70
`devinfo`	586	1624	2.77
`getNeighbors`	123	80	0.65
`getRoutes`	117	80	0.68
`multiprint`	810	1856	2.29
`ping`	300	204	0.68
`ping_pong`	268	184	0.69
`pingtime`	556	384	0.69
`query_gc`	100	72	0.72
`traceroute`	519	336	0.65
Median	353	244	0.69

Figure 6.1: Code size experiments. We present the wire format of the given PLAN program, that of the SNAP program output by our compiler, and the ratio of SNAP size to PLAN size.

the compiler so it does not unroll at all, relying instead on simulating backward branches with `send`, as discussed in the previous section. In this case, the resulting SNAP program sizes were 728 bytes and 576 bytes respectively, giving SNAP/PLAN ratios of 1.24 and 0.71. This restores the typical 30% improvement for the multicast example, but not for `devinfo`.

If we furthermore apply the topological sorting of basic blocks as discussed above, we can trim the resulting size of the `devinfo` SNAP program to 544 bytes, resulting in a ratio of 0.92. Now at least, the SNAP program is smaller than the original PLAN version, but not by much. A quick perusal of the SNAP program reveals that most operations are (often redundant) stack management operations. Hand-tuning to eliminate cases like a `push` immediately followed by a `pop` results in a new program size of 476 bytes (a ratio of .81), much closer to the usual. We would expect a more aggressively-optimizing compiler could achieve similar results (the current compiler does only simple optimizations like graph coloring to minimize stack frame sizes and consecutive-jump coalescing).

In summary, most PLAN programs do not require backwards branches and so can be straightforwardly compiled. The other PLAN programs do not have linear per-execution

resource usage (either due to loops or multiple nested function calls). In these cases, the resulting SNAP programs are correspondingly larger (from inlining or loop unrolling) or additional resource bound must be consumed to emulate backward branches. We feel that this code bloat is actually *desirable* in that it makes the resource usage of the program more apparent.

6.1.2 A Word about Services

It is important to qualify that the flexibility of an active packet system ultimately depends upon the set of node-resident services available. In an ideal world where services can be dynamically deployed on demand, as in SwitchWare [AAH+98], one might ask: why not simply do all active processing via new services?

Primarily, there is a differing granularity of change. Once an active packet language is deployed, new programs can be easily written and deployed. As a salient example, the active video application we describe below uses *different* SNAP programs depending upon what type of video frame is being carried and whether or not the packet is the last fragment of a frame, resulting in a total of six different programs. Having to download and deploy six different services for one application stream would seem to be quite heavyweight. Furthermore, we would be subject to potential thrashing resulting from a large number of applications competing to have their service code resident in shared nodes.

By contrast, the two native SNAP applications we develop below only require a very modest set of services, as listed earlier in Chapter 4, with most of these being for resident state—a very general service. This suggests that we could likely deploy a variety of new applications without having to deploy new services[1].

[1]Since services are currently statically bundled with the SNAP virtual machine, deploying new services is no harder (or easier) than upgrading the SNAP interpreter itself. Safe dynamic loading of mobile service extensions would make the difference between updating the language and deploying new services more evident, however.

6.1.3 Existing Applications

We list here several active applications from the literature that we could straightforwardly implement in SNAP. Most of these applications were originally implemented in PLAN and ANTS[2]; this suggests that SNAP can, in fact, express a variety of interesting applications.

Active Internetworking. Hicks *et al.* [HMA+99] have built an active internet using PLAN: in this system, all packets are active PLAN packets. PLANet is one of the most extensive active network applications to date. PLANet provides functionality for address resolution, dynamic routing, fragmentation, encapsulation, error reporting, *etc.* Building an entire internetwork infrastructure using PLAN points to the language's high flexibility. Indeed, many of the services and language constructs found in PLAN were directly inspired by implementing PLANet. One of the interesting insights was that a two-level architecture of active packets and node-resident services provides a flexible framework for network protocol development, where protocols could just as conveniently be expressed in an active packet mobile-agent style or as a more conventional message-passing service style, or a combination of both. For example, ARP-like address resolution [Plu82] is primarily active packet-driven, whereas RIP-style dynamic routing [Hed88] is primarily service-driven.

As mentioned above, our PLAN-to-SNAP compiler shows we could likewise build PLANet using SNAP instead of PLAN. Currently, Nettles *et al.* are reimplementing PLANet in a system called FASTNet [Hor00] implemented in TAL/Popcorn [MWCG99, MCG+99]. The hope is to use an improved version of the compiler (*e.g.*, one that does more aggressive optimization) to transparently translate PLAN programs to SNAP for transmission and execution.

Application-driven routing. Hicks *et al.* [HMA+99] describe the use of control-plane active packets in a scheme called Flow-Based Adaptive Routing (FBAR). Here, an application periodically sends active "scout packets" that fan out over the network to find the best available path to the destination according to some metric, such as least delay or least congestion. The scout packet with the best path then works its way back to the source,

[2]Since PLAN and ANTS are informally equally expressive [HMWN02], it seems likely we could implement in SNAP any applications developed in ANTS.

associating its route on each node with a "flow key," which it reports back to the sending application. The sender, in turn, sends data packets that use the most recent flow key to make their routing decisions. Hicks *et al.* demonstrate an application detecting congestion and successfully routing around it using this method. In fact, the PLAN programs used here are ones that can be translated by the PLAN-to-SNAP compiler.

Transparent Web Cache Redirects. Legedza and Guttag [LG98] describe a system based on ANTS to make web caches more effective. Here, routers maintain a set of "redirection pointers" for the URLs held in the nearest cache. Then, active web request packets traverse the usual shortest path towards the server; if a router is encountered with a redirection pointer for the desired URL, then the packet turns aside to the cache, rather than continuing on towards the server. Thus, a web request encounters at most one cache miss and is never deflected very far from the normal shortest path to the server. Given our resident-state service to store the redirection pointers, this could easily be implemented in SNAP as a program that checks the local node for a redirection pointer and follows it if present, otherwise proceeding towards the usual server.

Distributed On-line Auctions. Legedza *et al.* [LWG98] also use ANTS to improve on-line auctions. The auction server pushes the current highest bid information out into the network. Then, active packets carrying bids check this information on their way towards the server. If the bid each is carrying is not high enough to beat the currently cached bid, then the packet immediately turns around and reports a bid failure. This has the twofold benefit of improving the response time for the bidder as well as relieving the server of the obligation to process the failed bids. Again, this is easily implemented in SNAP with our resident-state service.

Active Reliable Multicast. Lehman *et al.* [LGT98] describe a scheme based on active packets for improving support for reliable multicast sessions. First, active multicast data packets install copies of their data in any multicast tree nodes equipped with special multicast caches before forwarding themselves on the downstream links. Then, when a receiver determines it has not received a particular data packet, it sends an active NACK

packet back up the multicast tree. As it progresses up the tree, it leaves behind a NACK record so that other NACK packets from the same subtree can simply record their interest in a retransmission without continuing further (thus avoiding NACK implosion at the root of the multicast tree). Retransmissions can either be found from multicast caches at strategic points in the tree or from the source. Then active "repair" packets work their way back down the subtree, doing selective retransmission based on those subtrees that had expressed their interest in the NACK records. All in all, this scheme distributes the load of performing retransmissions by keeping retransmissions as local as possible. Most of the complexity of this service derives from resident-state services (especially caching whole payloads for a period of time); the other aspect is a multicast distribution. Since we implement a basic multicast service in our Active Video-on-Demand application discussed below, this suggests we could use SNAP to implement Active Reliable Multicast.

Mobile Code Firewalls. Hicks and Keromytis [HKS02] describe an active firewall for PLAN. Here, active packets carry cryptographic credentials with them; the service interface they may access is then dependent upon policy. Thus, with this *namespace security*, unauthenticated packets may only invoke a set of "safe" services, whereas properly authenticated packets may choose from a richer set of node-resident services. This makes an active packet firewall possible where the firewall encapsulates incoming active packets with a restrictive "guest" identity, effectively limiting the service interface for outsider packets. Again, it would be simple for SNAP programs to carry their credentials in their heap and present them to an authentication service upon arrival at a node; this in turn would affect the service namespace available.

6.2 New Active Applications

With the exception of PLANet, the above applications are fairly narrow in scope, and are not really in the mainstream of ongoing networking research. We therefore have developed new active applications focused on two current hot topics: real-time multimedia and network monitoring for denial-of-service attack detection. SNAP's high efficiency plays a key

role in both cases, for it is only when network resources are scarce in these scenarios that we will want to apply active technology to adapt.

Furthermore, these two applications are a "stress-test" of various aspects of our SNAP implementation. Our network monitoring application explores SNAP's utility for control-plane applications. The multicast delivery part of the Active Video-on-Demand system probes SNAP's limited expressiveness, as the natural way to implement multicast using native unicast is to loop over all the outgoing interfaces, sending one copy of the packet on each. Furthermore, the multicast delivery is coupled with application-specific handling of congestion where SNAP's efficiency will be necessary to meet real-time demands.

6.3 Active Denial-of-Service Detection

In this section we describe SNAP's use for the scalable detection of distributed denial-of-service (DDoS) attacks. The key to scalability here is using SNAP as a lightweight mobile agent platform. We begin with a description of the application domain and then present the SNAP program used for the DDoS detection. We show that SNAP is sufficiently lightweight with some performance measurements and conclude with a discussion.

6.3.1 Distributed Denial-of-Service Attacks

Distributed denial-of-service (DDoS) attacks are an increasing problem, targeting well-known e-commerce and government sites [Yas01, Lem01, CS01, SPI00, BBC00]; easy to use tools for carrying out these attacks [CER01, CER00, CER99] are becoming widely available. For e-commerce sites, response time for such attacks is critical, as serious revenue losses can accrue during downtime, not to mention the impact on customer satisfaction. The first step of a response is detecting the attack in the first place.

To detect a DDoS attack, we can measure the amount of incoming traffic T into our administrative domain. Generally, as a network manager, we will have some traffic threshold T_{alarm}; if incoming traffic exceeds T_{alarm}, we want to sound an alarm and take action[3].

[3]Whether the surge in traffic is due to a DDoS attack or a massive influx of interest [For00], the network is being overwhelmed and we must take action, if only to determine the cause.

The current traffic input T is a good example of a *distributed variable*: namely, a variable whose value depends on information at multiple locations. In our case, we need to query incoming traffic load on multiple nodes. A centralized polling approach will quickly overwhelm the network operations center (NOC) with management data as the number of managed nodes grows, so this solution does not scale. The key to scalability lies in being able to distribute the computation of T, as suggested by Raz and Dilman [RD01].

Mobile agents have often been proposed as a solution for this scalability problem in network management (*e.g.*, [PT00, SBP98b, SBP98a, BGP97, BM98, SMB97] just to name a few). Having a programmable management platform also makes it easier to customize management applications to particular networks, or to quickly take new management actions (for example, a response to a new virus or denial-of-service attack).

Unfortunately, most mobile agent implementations (often based on Java [GJS96]) are so heavyweight as to limit their practical uses. Complex mobile agents consume valuable bandwidth as they migrate if their code representations are too large [RD99]. Feature-rich mobile agent runtime systems may put a substantial burden on nodes, even when there are no agents presently running on them. Finally, many systems may simply be too slow for effective real-time management; although mobile agents may be more scalable in terms of asymptotic complexity, the constants of proportionality are currently too large [RD99, BP98].

In the next subsection, we give an example of a network management mobile agent programmed in SNAP. As we have seen in Chapter 5, SNAP has extremely low overhead and so can provide a very lightweight mobile agent execution platform.

6.3.2 SNAP Surveyor Packets

Figure 6.2 shows the SNAP "surveyor" programs that we use to address the scaling problem mentioned in the previous subsection. This packet program carries a list of nodes to query and visits each in sequence. At each node i it queries the local traffic load T_i and keeps a running sum, T_{sum}. Once all nodes have been visited, the surveyor returns to the NOC and reports the current value of T. The algorithmic intuition is the following: with n nodes to manage, a centralized polling approach requires $O(2n)$ network hops (out to each node

```
main:
        forw            ; get to next hop
        bne     athome  ; if homeward flag set, just deliver data

        ;; else, we need to update load sum
        calls   "getld" ; get current load
        add             ; load sum

        ;; any more nodes to visit?
        pull    1       ; get n (number of remaining nodes)
        bez     gohome  ; if out of addrs, go home

        ;; re-arrange stack state in preparation for transit
        pull    2       ; get next node's address
        pull    2       ; get n
        subi    1       ; n--
        store   4       ; put new n over old next hop
        pull    1       ; pull load sum
        store   3       ; put load sum over old n
        push    0       ; still more hops to go; unset homeward flag
        store   2       ; put flag over old load sum
        forwto          ; move on to next hop

gohome:
        push    1       ; set homeward-bound flag
        getsrc          ; find out where home is
        forwto          ; go there

athome:
        getspt          ; get port number for delivery
        demux           ; deliver load sum
```

Figure 6.2: SNAP "surveyor" program

and back), whereas the surveyor approach requires $O(n)$ hops. Perhaps more importantly, in the centralized approach, all $O(2n)$ hops involve the NOC, whereas in the surveyor approach, only 2 hops involve the NOC. Thus, not only is the network traffic reduced, but it is also distributed.

The above program consists of just 20 instructions, for a total of 80 bytes of code in a SNAP packet. This leaves significant room in the packet for carrying accumulated data and/or addresses of nodes to visit. Even with a maximum transmission unit (MTU) as small as 256 bytes, there are still 132 bytes of room left over in the packet after headers and code (enough to visit 32 nodes, assuming 4 bytes of accumulated data as in the above example). With more realistic autonomous domain MTUs of 1500 bytes, one packet could easily visit over 300 nodes.

6.3.3 Performance Evaluation

In this section we present experiments to demonstrate that SNAP provides a lightweight network monitoring system. First, we compare simple polling overheads between SNAP, SNMP [CFSD90], and ICMP [Pos81a]. Then we explore possible savings resulting from SNAP's flexibility.

Experimental Setup

Our experiments are performed on the same PC cluster used for the microbenchmarks in Chapter 5. Each machine has a Pentium III (Coppermine) 1 GHz CPU, 256 MB of RAM, and a SuperMicro Super 370 DE6 motherboard with on-board Intel Speedo3 100 Mb/s Ethernet card. The cluster runs RedHat Linux 6.2, with kernel version 2.2.19-6.2.7 patched to provide SNAP support. All machines are on the same LAN, switched by an Asanté Fast100 Ethernet hub.

For our polling microbenchmarks, we perform one-hop polls: for SNMP, we use `ucd-snmp` version 4.1.1-3 and do a simple *get* for the `system.sysName.0` MIB variable. For ICMP, we use the standard `ping` program. For SNAP, we use the surveyor program described above. For the polling microbenchmarks here, we use a list of only one node, to

Protocol	Latency (μs)
SNMP get	256
SNAP surveyor	93
ICMP ping	73

Table 6.1: Polling microbenchmark results.

mirror the one-hop polling of the other applications. All microbenchmarks are driven by user-space applications.

Polling microbenchmarks

Table 6.1 shows the median of 21 trials for each of the polling microbenchmarks (these results are also shown in Figure 6.3 in the one hop case). The upper and lower quantiles are within 1 μs of the median. As we can see, there are substantial savings to be gained from directly querying MIB variables in the kernel: the median latency for SNAP is 64% faster than that for querying the user-space `ucd-snmpd`. Furthermore, we see that SNAP imposes minimal overhead over a simple ICMP ECHO-REPLY request (which contains no control logic and queries no MIB variables) being only 20 μs slower.

These results suggest that SNAP is more than lightweight enough to be a drop-in replacement for SNMP. Thus, SNAP could practically serve in a variety of management schemes ranging from centralized polling all the way to mobile agents, and network operators may simply make the appropriate tradeoffs between latency and bandwidth. In other words, SNAP offers significantly increased flexibility over simple SNMP without undue overhead.

Surveyor results

Figure 6.3 shows the latencies for SNAP surveyor packets; the x-axis shows the number of managed nodes whose loads were queried, and the y-axis shows the round-trip latency seen by the monitoring application. We present the median of 21 trials, with the upper and lower quartiles represented as error bars (although the variability is so small that the error bars are hard to see; all quartiles are within 4% of the median (and often closer)).

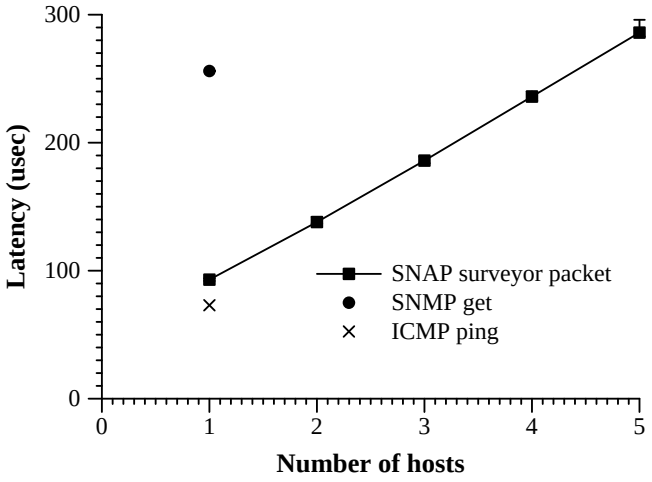

Figure 6.3: SNAP "surveyor" packet latencies

As expected, the round-trip latency scales linearly with the number of queried nodes. There is approximately a 50 μs overhead per hop in terms of latency, which is extremely lightweight: the SNAP surveyor packet can sum the loads on 4 nodes and report back in less time than one SNMP `get`. Of course, in a more realistic setting, packets would incur much longer propagation delays, so the speed advantage of SNAP execution would be lessened.

However, a SNAP-based querying application may successfully "piggyback" several node queries with one surveyor packet. Indeed, this surveyor approach scales well: monitoring applications have plenty of flexibility with regard to how many surveyor packets need to be sent simultaneously and thus how many nodes each surveyor will be responsible for querying. Thus, management applications have complete flexibility to trade off higher latency for reduced bandwidth.

6.3.4 Discussion

The specific example presented here is just one of a class of *distributed threshold detection* problems; Raz and Dilman [RD01] point out several such problems, including monitoring general network traffic, Web mirror loads, software licenses, bandwidth brokerage, and

denial-of-service attacks. In each case, we want to know whether some global network threshold has been exceeded.

Raz and Dilman's approach, efficient reactive monitoring, apportions some "ration" of the global threshold to each monitored node; the node monitors its own state and, if the ration is used up, triggers an alarm to the NOC, which then issues a global poll. As long as none of the nodes exceeds its ration, the global usage cannot have exceeded the global threshold.

This approach can be adapted to use SNAP surveyors in the following way: at some point, the surveyor may be able to determine, based on the remaining number of nodes to visit and the maximum load value on each node, that $T < T_{alarm}$ (for example, during low traffic periods, it is likely that the load sum of the first few nodes will be very low). Conversely, during an attack, the threshold may be exceeded after visiting only a handful of nodes, at which point the surveyor may simply return to the NOC to sound the alarm. In both cases, we leverage domain-specific knowledge of the system to avoid querying some nodes at all. Being able to correctly monitor a network by using such conservative estimates can allow us to reduce the overhead of network monitoring. This is exceptionally important during crisis periods when the network's resources are stretched to the limit.

Since SNAP has basic arithmetic and logic capabilities, it seems to be well-suited for use as a network monitoring mobile agent language (especially given services that provide an SNMP-like MIB interface). Because of its low overhead (especially compared to SNMP), it can be fairly straightforwardly used to replace existing SNMP-based central polling routines. Once a SNAP system is in place, the old polls can be altered to use a more scalable mobile agent approach, and new polls can be easily accommodated due to SNAP's programmability. Stated simply, SNAP can achieve a large part of the flexibility of mobile agents at costs similar to SNMP.

6.4 Active Video-on-Demand (AVid)

In this section we describe a video-on-demand service implemented using SNAP. This system provides experience with some key aspects of SNAP; specifically, this setting involves

high-speed, real-time network traffic, so it is critical that SNAP's overhead be small. In addition, we examine a multicast setting, so we must cope with both the fact that SNAP does not support a native multicasting primitive as well as the fact that the most natural way to implement multicast using unicast network operations involves looping (backward branches). Finally, our video-specific algorithms rely on node-resident state, so we gain experience with our service framework.

From here, we proceed to describe video-on-demand in more detail. Then we will describe the SNAP programs used to implement the AVid system and present some initial performance evaluation showing gains from the use of active packet technology. Finally, we conclude this section with a discussion assessing the impact of our successful SNAP programming on our evaluation of SNAP's flexibility.

6.4.1 Video-on-Demand

Video-on-demand (VoD) [GVK+95] is a service similar to pay-per-view television: users may select media to be delivered immediately (or with a short delay). The main characteristic of concern is that the video must be delivered in real-time; it is unacceptable to require a user to download an entire video segment before beginning to view it. Generally speaking, VoD servers transmit media over multicast channels where possible, thereby reducing overall resource usage.

There are many existing VoD systems [AWY96b, AWY96a, BM99, DHS96, DSS94, DSS96, EFV99, EV98, GT99, GLM95, HS97, VI96, RRG01], with varying degrees of sophistication. We have implemented an active VoD system (AVid) using SNAP. Our goal here is not to implement a new, highly complicated VoD architecture, but rather to explore SNAP's flexibility using this application domain. AVid uses SNAP programs in two main ways: first, to establish a multicast framework for data delivery, and second, to perform video-specific reactions to congestion to improve overall video quality.

6.4.2 SNAP programs used

There are two main SNAP programs used in AVid[4]: `refresh`, which is sent periodically by the clients to maintain their subscription to the multicast tree, and `multdeliver`, which performs multicast delivery combined with video-specific congestion reaction.

Multicast tree maintenance

The multicast framework is similar to the one proposed by Tani *et al.* [TMT01] where the multicast group is a single sender/multiple receiver scenario (as opposed to a more general multiple sender/multiple receiver setup). The full SNAP code for `refresh` is shown in Figures 6.7–6.12 at the end of the chapter; for now we will present the code in pieces to better explain it.

The multicast tree is represented by a resident-state variable "`m_list`" on each node that contains a list of the current node's children in the tree. The main purpose of `refresh` is then to keep these variables up to date. Finally, leaf nodes (clients) are marked by setting a resident state variable `endnode`.

When a client sends a `refresh`, it first executes in the SNAP VM on the client node to mark the current node as a leaf, record the client's address, and start working up the tree towards the server:

```
on_client:
      push   1          ; indicate this is a client node by setting
      push   "endnode"  ; resident state: endnode := 1.
      calls  "put"
      push   <servaddr> ; server address
      here              ; this node's address
      push   first      ; new entry point = first
      push   2          ; carry two stack vals (server, here)
      getrb             ; use rest of resource bound
      pull   4          ; (server) -> destination
      send              ; move on towards server
      exit
```

Upon arriving at an interior node of the tree, the packet sets the `linkActive` resident-state variable on its incoming interface to indicate there are downstream subscribers on that link

[4]These SNAP programs were written by Hooman Radfar and Matthew Keesan, senior undergraduates at the University of Pennsylvania, under supervision of the author.

(as we will see later, the presence of this variable prevents this link from being pruned out of m_list):

```
first:
      pull   0       ; extra copy of previous node for storing
      push   "linkActive"
      pull   2       ; previous node again
      route          ; find next hop to previous node
      rtdev          ; find outgoing device for previous node
      store  1       ; rearrange stack
      calls  "ifput" ; store previous node address on device
```

Next, the packet checks whether the m_list variable exists on the current node, by attempting to retrieve it (an exception is raised if the variable does not exist):

```
prep:
      push   "m_list"     ; find out what the current list is
      calls  "get"
      isx                 ; was there an exception from get?
      bne    no_list - pc ; if so, no list installed, we must create it
      pull   0            ; copy first tuple of list
```

If the list does not exist, we must create it:

```
no_list:
      pop             ; pop exception
      push   0        ; push null pointer
      mktup  2        ; cons (previous node, null)
      push   "m_list"
      calls  "put"    ; store the new list
```

Once the list has been created, the packet must move upward in the tree (towards the server) to make sure the current node is added to its parent's m_list:

```
      here                  ; moving on. this node must be added on
                            ; next node
      push   first          ; entry point = first
      push   2              ; carry 2 stack vals (whole stack)
      getrb                 ; use rest of resource bound
      pull   4              ; server's address
      pull   0              ; copy server address
      here                  ; get current address
      eq                    ; compare
      bne    avid_exit - pc ; if equal, we are at server, done
      send                  ; else, forward towards server
      exit
```

Now, in the case that `m_list` already exists on a node, we must determine if the previous node's address is already in the list. The loop uses **send** to emulate backward branches where necessary. First, we look at the first address in the list and compare it to the previous node's address. If we find it, we can exit the loop:

```
loop:
      pull   0    ; extra copy of current tuple
      nth    1    ; extract host part
      pull   3    ; copy of previous node address
      eq          ; are the two addresses equal? if so, we are
                  ; done and just need to check if we should
                  ; propagate towards the server
      bne    doWeNeedToUpdateIntro - pc
```

If the current list element does not match the previous node's address, we iterate down `m_list` to check the next element. If there are no more list elements, the loop terminates:

```
      nth    2          ; else, hosts don't match, get next tuple in
                        ; list
      pull   0          ; make copy of next pointer
      bez    end - pc   ; if next == 0 (null) we are not in the list
```

If there are additional list elements, however, we must branch backward to the beginning of our checking loop:

```
      push   loop    ; entry point = loop
      push   4       ; 4 stack vals (all of current stack)
      getrb          ; use rest of resource bound
      here           ; dest addr = here
      send           ; send packet here
      exit
```

If we get to the end of the loop without finding our previous node's address, we must add it to `m_list`:

```
end:
      pop                           ; pop null pointer
      pull   1                      ; (previous node)
      pull   1                      ; (first tuple in list)
      mktup  2                      ; cons
      store  1                      ; rearrange stack
      pull   0                      ; copy of new cons cell
      push   "m_list"
      calls  "put"                  ; store new multicast list
      ji     doWeNeedtoUpdate - pc  ; check for propagation
```

At this point, we know that `m_list` exists and contains the previous node's address. Now we want to know if we need to add the *current* node's address to its parent's multicast list. We use a resident-state variable called "`wentUp`" that is set by a packet before it propagates upward in the tree. If this variable is set, then another packet recently refreshed the current node's subscription to the tree, and the current packet may terminate. If the variable is not set (recall that resident-state is meant to time out), then the current packet must propagate upwards.

```
doWeNeedToUpdateIntro:
      pop                   ; pop extra tuple copy
doWeNeedtoUpdate:
      store  1              ; stack rearrangement
      push   "wentUp"       ; see if wentUp variable exists
      calls  "get"
      isx                   ; is exception? variable not defined
      bne    next - pc      ; if exception, need to update
      bne    avid_exit - pc ; else, someone else went upwards
                            ; recently; we are done
```

Now the packet has determined it must propagate upwards. Before doing so, we take the opportunity to do some maintenance by pruning the current node's `m_list` to remove children that are no longer actively subscribed. Again, this loop uses **send** to branch backward. The packet constructs a new version of `m_list`; for each child in the old list, we must check whether the `linkActive` variable mentioned above is still set on the appropriate outgoing interface. If so, we add the child to the new list, otherwise we do not:

```
loopA:
      pull   0            ; copy of m_list
      nth    1            ; first host in list
      pull   0            ; copy of host
      push   "linkActive" ; boolean variable stored on iface
      pull   1            ; copy of host in list tuple
      route               ; route to host
      rtdev               ; outgoing device towards host
      store  1            ; stack rearrangement
      calls  "ifget"      ; see if linkActive exists on this device
      isx                 ; if exception, linkActive no longer exists
                          ; (timed out)
      bne    loopC - pc   ; in that case, go to loopC; skip this host
```

If the link is not active, we skip this host, advance down the old m_list, and branch backward to the beginning of this loop:

```
loopC:
      pop               ; pop exception
      pop               ; host of tuple
      nth    2          ; get next pointer

      ;; loop back to loopA
      push   loopA      ; entry point
      push   3          ; carry whole stack (3 values)
      getrb             ; use rest of resource bound
      here              ; stay on same host
      send              ; send packet
      exit
```

If the link was active, we must add the current child to our new version of m_list:

```
      popi   2          ; linkActive exists, get rid of extra args
      pull   1          ; copy current list element
      nth    2          ; get next list element (cdr)
      pull   1          ; copy host
      pull   4          ; copy previous list element
      mktup  2          ; cons
      store  4          ; overwrite previous tuple in stack
      store  2          ; overwrite next tuple in stack
      pop               ; stack rearrangement
```

Finally, if there are any more elements in the list, we must branch back to the beginning of the loop, otherwise we are done looping:

```
      pull   0          ; if next is null, done
      bez    cleanup - pc

      ;; loop back to loopA--next tuple is non-null
      push   loopA      ; entry point = loopA
      push   3          ; carry three stack values
      getrb             ; use rest of resource bound
      here              ; loop back to here
      send              ; send packet
      exit
```

Once the list has been pruned, we install the new version of m_list, and then set the wentUp variable to indicate we have propagated upwards recently:

```
cleanup:
      pop                ; pop null pointer
      push   "m_list"    ; install new list on node
      calls  "put"
      push   1           ; indicate we went upwards recently
      push   "wentUp"
      calls  "put"       ; wentUp := 1
```

Finally, our packet must propagate upwards in the tree. Before sending the new packet, we check whether we are actually at the server; if we are, then we do not need to propagate and the packet can terminate. Otherwise, we record the current node's address, and set the entry point for the outgoing packet to `first` (the main entry point for internal nodes):

```
      here                   ; current node will be the previous node of
                             ; the next hop towards the server
      push   first           ; entry point
      push   2               ; 2 stack values: server, current host
      getrb                  ; use rest of resource bound
      pull   4               ; server address
      pull   0               ; copy of server address
      here                   ; current address
      eq                     ; compare addresses
      bne    avid_exit - pc  ; already at server, done
      send                   ; else move on
avid_exit:
      exit
```

Multicast video delivery

The other SNAP program used, `multdeliver`, combines multicast transport with video-specific congestion reaction [BCZ96, RRV93]. The `multdeliver` program follows the multicast tree described by the `m_list` resident-state variables set up by the `refresh` program above. If at any point it determines that part of a video frame has been lost, it pre-emptively drops the rest of the frame to reduce network resource usage. Furthermore, if it determines that there is congestion on the outgoing link, it may pre-emptively drop less

important frames to give preference to more important ones[5]. The full code for `multdeliver` is shown in Figures 6.13–6.18 at the end of the chapter.

A video delivery packet begins by executing in the SNAP VM on the server and setting up the stack appropriately. In addition to a piece of MPEG data, each packet also carries the number of the frame to which it belongs, as well as a packet number within that frame (with packet number zero being the first part of a frame). These variables will be used later on to detect when pieces of frames have been lost.

```
initial:
      push   <mpeg data>
      push   <frame number>
      push   <packet sequence number within frame>
```

As it arrives on each node (including the server), it determines whether the node is an interior node in the multicast tree or is a leaf node. Recall that leaf nodes are marked with the `endnode` resident-state variable set by `refresh` packets:

```
get_nodetype1:
      push   "endnode"
      calls  "get"               ; check endnode resident variable
      isx                        ; check for exception
      bne    get_nodetype2 - pc  ; if exception, continue
      bne    deliver - pc        ; else, deliver data
```

If the node is a leaf node, we simply deliver our video payload:

```
deliver:
      pull   2                ; pointer to mpeg data
      demuxi <portnum>        ; deliver to client
      ; note: demuxi results in a packet exit
```

Otherwise, we look for the multicast list `m_list`. If it is not present, the packet simply terminates:

[5]MPEG streams are divided into I-, P-, and B-frames; I-frames may be decoded independently, while P-frames depend upon previous I-frames, and B-frames depend upon previously-received I- and P-frames. Thus, the I-frames are the most important, with P-frames being the next most important, and B-frames being the least important.

```
get_nodetype2:
      pop                     ; exception
      push   "m_list"         ; load current multicast list
      calls  "get"
      isx                     ; if exception, no list present
      bne    full_exit - pc   ; in that case, just drop ourselves
      bne    tests - pc       ; else move on to congestion tests
```

If `m_list` is present, we iterate over the list, sending a copy to each child in the list. Before sending, however, we perform our video-specific congestion adaptation. For each child, the packet checks the queue length on the appropriate outgoing interface, and drops itself if the queue is too long, where the definition of "too long" depends on the type of frame being carried (recall that I-frames are more important than P-frames, which are more important than B-frames):

```
congestion_test:
      pull   0                ; copy of list
      nth    1                ; extract host element
      pull   0                ; copy of host
      route                   ; determine next hop to host
      rtdev                   ; determine outgoing interface
      store  1                ; stack rearrangement
      pull   0                ; copy of interface
      calls  "qlen"           ; get outgoing queue length
      pull   0                ; copy of queue length
      subi   5                ; check for congestion
      gti    0                ; is (len-5)>0? then len>5
      bez    frag_check1 - pc ; if not, no congestion
      push   <frame weight>   ; weighting for different frame types
      div                     ; queue length / frame weight
      gti    2                ; compare to 2
      bez    frag_exit - pc   ; too congested, drop
      push   0                ; else continue (push dummy stack val)
```

Next, we check to see whether we have lost any pieces of the current frame. Previous packets will have set the `frame_num` and `packet_num` resident-state variables to indicate the last packet that successfully got through. These variables are stored on a per-interface basis, as different outgoing links may have differing congestion. Our current packet then determines whether it is indeed the next packet in sequence or whether some packets have already been dropped. First, the packet gets the expected frame number:

```
frag_check1:
      pop                       ; queue length
      push   "frame_num"        ; see what frame number was stored
      pull   1                  ; copy of device
      calls  "ifget"            ; stored per-interface
      isx                       ; if exception, nothing stored there
      bez    frag_check2 - pc   ; else move on to next phase
```

If the variable does not exist and we are the first frame of a packet (packet number zero), then we skip the rest of the fragmentation checks. Otherwise, we have already lost a piece of our frame (at least the first one) and we drop ourselves:

```
fix1:
      pop                       ; exception
      pull   4                  ; copy of current frame number
      push   "frame_num"
      pull   2                  ; copy of outgoing device
      calls  "ifput"            ; store current frame number
      pull   3                  ; copy of current packet number
      pull   0                  ; duplicate
      bez    finish_loop - pc   ; first packet of the frame, ok
      pop                       ; packet number
      ji     frag_exit - pc     ; otherwise, fragmentation: drop packet
```

Next we perform the same check for the packet sequence number. We begin by retrieving the previous packet number:

```
frag_check2:
      push   "packet_num"
      pull   2                ; copy of outgoing device
      calls  "ifget"          ; get current packet number for device
      isx                     ; if exception, nothing stored
      bez    check_seq - pc   ; not exception, continue checking
```

If there is no packet number stored, we assume there are pieces lost and skip ahead. Otherwise, we examine our own packet number: if it is one greater than the stored one, we then skip ahead to checking the frame number.

```
check_seq:
      addi   1                  ; add 1 to stored packet number
      pull   0                  ; make copy
      pull   6                  ; my packet number
      eq                        ; compare
      bne    check_frame - pc   ; they match, continue checking
```

If we are not the next greater packet number, it does not indicate lost frame pieces if we are packet number zero (the first packet of a frame). Otherwise, this frame is fragmented and should be dropped:

```
check_seq2:
      popi   2                  ; clear unneeded args
      pull   3                  ; copy of my packet number
      pull   0                  ; extra copy for branch
      bez    finish_loop - pc   ; frame piece 0: don't need to check
                                ; the frame number
      pop
      ji     frag_exit - pc
```

Now that we have verified the packet number matches, we must also verify that our packet carries the correct frame number. Now, we should either match the expected frame number stored in `frame_num` or be the first packet of a frame:

```
check_frame:
      pop
      pull   5                  ; copy of our frame number
      pull   1                  ; stored frame number
      eq                        ; compare
      bne    finish_loop - pc   ; they match, go ahead

check_frame2:
      pull   4                  ; copy of packet number
      bez    finish_loop - pc   ; if zero, first packet of frame, ok
      pop                       ; else, fragmentation: skip
      ji     frag_exit - pc
```

At this point, if the frame number and packet numbers indicate we are the next packet in sequence, we store our values on the outgoing interface for the next packet to examine:

```
finish_loop:
      pop                       ; extra frame copy
      pull   3                  ; copy of packet number
      push   "packet_num"
      pull   2                  ; outgoing interface
      calls  "ifput"            ; store current packet number
      pull   4                  ; copy of frame number
      push   "frame_num"
      pull   2                  ; outgoing interface
      calls  "ifput"            ; store current frame number
      ji     unroll_exit - pc   ; all done
```

If we determine that our frame is missing some packets, all other packets belonging to this frame should be dropped. Therefore, we set up `packet_num` so that only the first packet of a subsequent frame will continue onward.

```
frag_exit:
      push   -1             ; frag, next ok packet number is 0
      push   "packet_num"
      pull   2              ; outgoing interface
      calls  "ifput"        ; packet_num := -1
      popi   2              ; clear extra args
      ji     loopback - pc  ; handle next multicast hop
```

Finally, if the outgoing link is not congested and our frame number and packet numbers match appropriately, we forward ourselves down the multicast tree. Since the current node may have multiple children in the multicast tree, we must divide our resource bound between multiple downstream copies of our packet. We look ahead in the list to see how many remaining children there are (to a maximum of three); this small lookahead loop is actually unrolled:

```
unroll_exit:
      pop
      push   0               ; zero hosts left
      pull   2               ; copy of multicast list
      nth    2               ; next pointer
      pull   0               ; copy of next host
      bez    one_exit - pc   ; if null, done unrolling
      pull   0               ; else copy next tuple
      nth    2               ; get next next tuple
      pull   0               ; make copy
      bez    two_exit - pc   ; if null, done unrolling
three_exit:
      push   3               ; 3 hosts in unroll
      ji     two_exit_2 - pc
two_exit:
      push   2               ; two hosts left to unroll
two_exit_2:
      store  3               ; store unroll number lower in stack
      popi   2               ; clear out extra stuff above it
      store  1               ; stack rearrangement
      ji     sendalong - pc  ; ready for send
one_exit:
      popi   3               ; clear extra
      push   1               ; one host left to unroll
```

Now that we have an estimate of the number of remaining children (including the current child in our outer `m_list` loop), we can divide our resource bound appropriately. First we must do some stack rearrangement to get the appropriate values in the right positions for the **send**, and set the entry point to be `get_nodetype1`, which is the main entry point of the program:

```
;; extensive stack rearrangement
sendalong:
      pull   4              ; mpeg data
      pull   4              ; frame number
      pull   4              ; packet number
      pull   4              ; multicast list
      store  8              ; store list lower in stack
      store  4              ; move packet number
      store  4              ; move frame number
      store  4              ; move mpeg pointer
      push   3              ; carry three stack items (mpeg,
                            ;   frame number, packet number)
      getrb                 ; use rest of resource bound
      pull   2              ; number of hosts to send to
      div                   ; divide resource bound
      pull   6              ; copy of multicast list
      nth    1              ; extract host
      push   get_nodetype1  ; entry point = get_nodetype1
      store  4              ; position entry point in stack
      send                  ; send to host
      pull   3              ; copy of list
```

Finally, we must branch backward to finish processing the rest of the children in `m_list`, if any:

```
loopback:
      nth    2               ; get next host off list
      pull   0               ; copy
      bez    full_exit - pc  ; if null, end of list
      push   congestion_test ; entry point = congestion_test
      push   4               ; carry 4 stack values
      getrb                  ; use rest of resource bound
      here                   ; evaluate here again
      send                   ; loop back
full_exit:
      exit
```

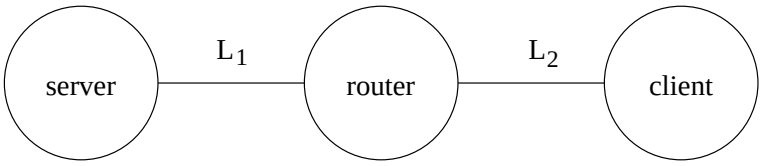

Figure 6.4: Evaluation topology for AVid.

6.4.3 Performance Evaluation

We have evaluated the AVid system in the topology shown in Figure 6.4; we limit our study to a unicast scenario to isolate the effects of congestion adaptation by the `multdeliver` packets. In the following experiments, we vary the bandwidths of links L_1 and L_2 to see how video delivery is affected. We see that in cases where there are multiple congested links on the path from server to client, our active packet approach can achieve substantially better end-to-end performance.

The data rate of our sample MPEG video is under 1.5 Mb/s, so we needed to find a way to limit the 100 Mb/s Ethernet links in our testbed to cause a scarcity of bandwidth. For this, we chose to use the class-based queuing (CBQ) support available in the Linux 2.4.x kernels; rather than porting our kernel implementation to the new kernel, we chose to port our VM to user space and use the 2.4 kernel's `netfilter` interface to capture incoming SNAP packets[6].

We have a sample MPEG video that is just under 22 seconds long, at a 352x240 pixel resolution. This video stream consists of 40 I-frames, 160 P-frames, and 400 B-frames. We compare AVid with its unit frame-dropping active packets to a version that uses straightforward SNAP unicast delivery packets. We present two metrics here: the number of complete frames of each type (I, P, and B) successfully received by the client, as well as the number of "useful" packets received (*i.e.*, not part of an ultimately incomplete frame). While it might be interesting also to count the number of frames ultimately decodable by the MPEG viewing software, our current software does not allow these numbers to be easily

[6]The user space version, although it incurs additional kernel crossings, is nonetheless fast enough to accommodate our video stream without loss.

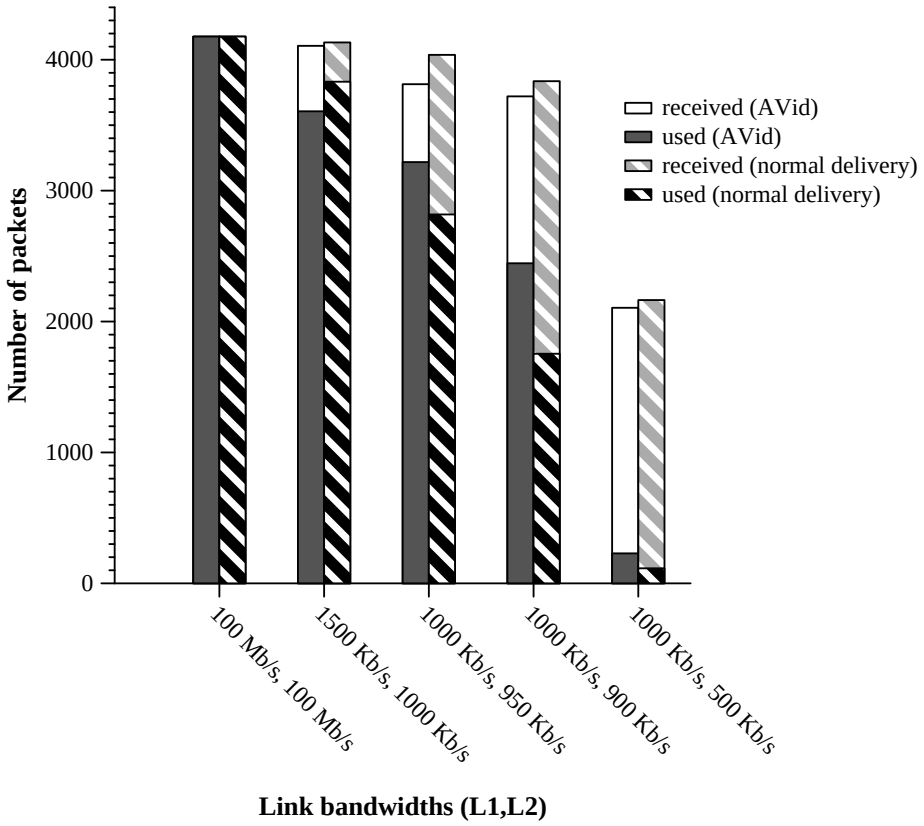

Figure 6.5: Experimental results for AVid, in terms of how many packets received by the client were used.

collected; nonetheless, the metrics we do present achieve our purpose—to demonstrate that AVid is a useful active application.

Our experiments are performed on the same PC cluster used for the microbenchmarks in Chapter 5 and the surveyor benchmarks above. Each machine has a Pentium III (Coppermine) 1 GHz CPU, 256 MB of RAM, and a SuperMicro Super 370 DE6 motherboard with on-board Intel Speedo3 100 Mb/s Ethernet card. These machines run RedHat Linux 7.3, with kernel version 2.4.18-5. All machines are on the same LAN, switched by an Asanté Fast100 Ethernet hub.

The results of our experiments are shown in Figures 6.5 and 6.6. In both graphs, the x-axis shows various configurations of link bandwidths, while the y-axis shows numbers of packets and numbers of frames, respectively. We see that when both links L_1 and L_2

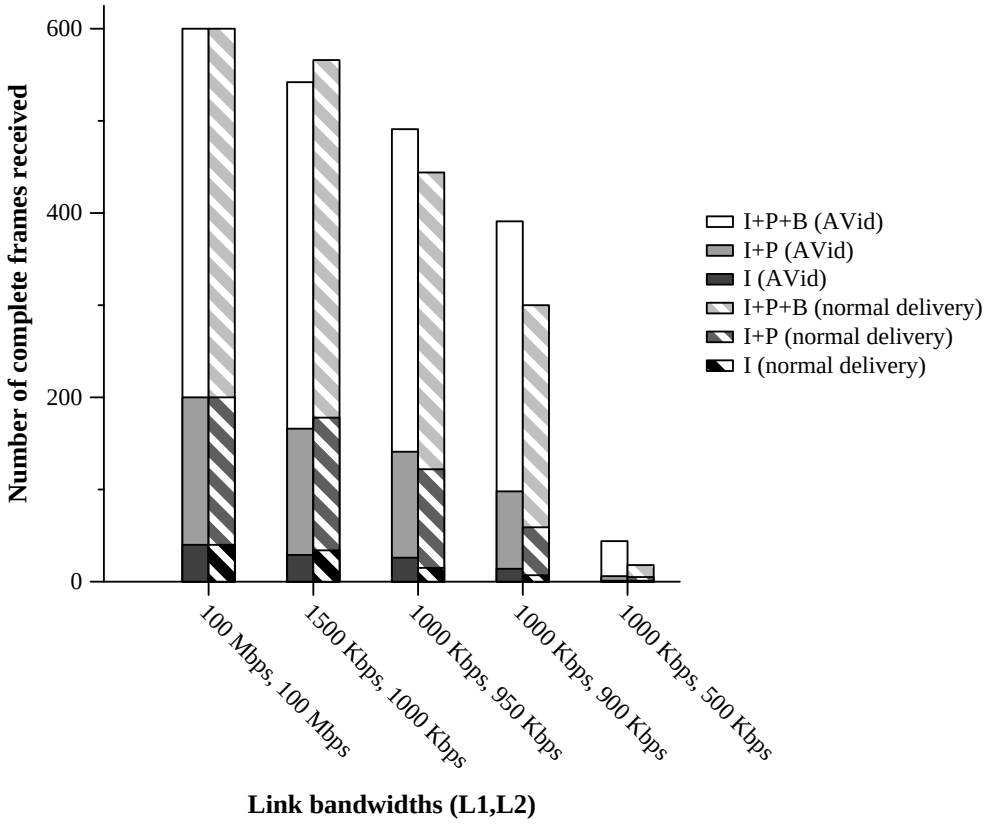

Figure 6.6: Experimental results for AVid, in terms of number of frames received.

are unconstrained, both applications are able to transmit the entire video at full speed without losing any packets. In the second experiment, the bandwidth on L_1 is set to 1500 Kb/s (above the actual data rate of the video), while the bandwidth on L_2 is set to 1000 Kb/s (slightly below the video data rate). We see that both applications experience a few lost packets, although since these losses occur on L_2, the active packets are unable to compensate, and we see that AVid does slightly worse than the normal delivery version.

However, once we begin constraining the bandwidth on L_1 to be 1000 Kb/s, and then set the bandwidth on L_2 even lower, we see the active congestion adaptation taking effect. In these cases, some active packets arriving on the central router discover that part of their frame has already been lost on L_1 and they pre-emptively drop themselves, thus conserving precious L_2 bandwidth. Interestingly enough, in the last three experiments, the AVid client receives *fewer* packets than the normal client yet yields *more* decoded frames: the unicast client receives many more "useless" packets containing pieces of data corresponding to incomplete frames. For example, in the fourth experiment, with L_1 at 1000 Kb/s and L_2 at 900 Kb/s, AVid receives 3720 packets, 2445 (66%) of which belong to complete frames, while the standard delivery receives 3836 packets, only 1754 (46%) of which are part of complete frames.

6.4.4 Discussion

Both `refresh` and `multdeliver` make use of **send** to emulate backward branches by having a packet loop back to the same node. This is accomplished easily by a series of six instructions:

```
push    <entry point>  ; backward branch target
push    -1             ; take the whole current stack
getrb                  ; use remaining resource bound
here                   ; destination is same node
send
exit
```

Note that although this does indeed perform a backward branch, it has two main important properties. The first is that the **send** causes a resource bound to be consumed from the packet, thereby satisfying our global resource predictability properties. The second is

that the **send** creates a new packet, while the **exit** terminates the current packet. Thus, the new packet is re-queued on the loopback device and must wait for service (whereas a simple backward branch instruction in SNAP would allow the original packet to keep executing uninterrupted), thereby satisfying our local node resource usage properties for per-packet execution.

Therefore, backward branches are available at the cost of a resource bound and an interruption of execution. This permits at least a limited amount of looping (such as that required to iterate over a fairly short list, as in our multicast example here) to be easily accomplished. Indeed, the draconian requirement of forward-only branches is not quite as restrictive as it might appear at first glance. Given our experience with the usefulness of this construct, we are considering simply adding backward branches that interrupt execution and cost 1 RB, as detailed in the next chapter.

AVid also uses the resident-state services to provide inter-packet communication to coordinate whole-frame dropping and to allow low-overhead multicast framework maintenance. The general consensus among the AVid developers is that resident-state services drastically increase the number of interesting applications expressible with SNAP. Generally speaking, packets can make more intelligent decisions with more information, so providing packets with a way to record their fate for other packets to read is a powerful tool. Our analysis in Chapter 4 for providing resident state in a resource-safe manner is therefore important.

The final interesting property of our multicast framework is that it allows legacy IP nodes to be freely interspersed between the SNAP-aware active nodes. Although this is an artifact of our Router Alert IP option implementation, it is quite useful, as the multicast framework that is established creates IP tunnels between SNAP nodes *on-the-fly* and without coordination. Thus, while unicast tunnels in the current MBone must be established manually, this administrative overhead is not necessary with SNAP.

As a specific example, a content provider might team with an Internet service provider (ISP) to set up an AVid-like service. With one SNAP node at the server, and SNAP nodes at the ISP's ingress points, the resulting multicast tree would look the same regardless of whether the intervening backbone routers were SNAP nodes or not. Furthermore, the ISP

could incrementally roll out additional internal SNAP nodes to further benefit from the bandwidth savings of multicast over unicast. Most pointedly, however, the content server and ISP need not coordinate with anyone else to make this happen. Indeed, a popular service like this is the most likely path to widespread SNAP deployment.

6.5 Recap

In this chapter, we have established that SNAP is flexible and expressive enough to implement a wide range of active applications. The development of a PLAN to SNAP compiler shows that SNAP is a useful low-level transmission and execution layer for higher-level active networking languages. In addition, we have implemented two native SNAP applications. Our experience with these applications indicates both that some useful active networking applications are expressible in SNAP without backward branches (as in the surveyor packets for DDoS detection) and that limited backward branching is available in a reasonably convenient way where necessary. In short, SNAP does indeed show increased flexibility over traditional passive packet approaches, despite the language restrictions we have made for safety and efficiency reasons.

```
on_client:
        push 1          ; indicate this is a client node by setting
        push "endnode"  ; resident state: endnode := 1.
        calls "put"

        push <servaddr> ; server address
        here            ; this node's address
        push first      ; new entry point = first
        push 2          ; carry two stack vals (server, here)
        getrb           ; use rest of resource bound
        pull 4          ; (server) -> destination
        send            ; move on towards server
        exit

;; store the previous hop in the linkActive variable on the correct
;; outgoing interface, to indicate there is activity on this child
;; link. the stack at this point is []::server::previous_node
first:
        pull    0       ; extra copy of previous node for storing
        push    "linkActive"
        pull    2       ; previous node again
        route           ; find next hop to previous node
        rtdev           ; find outgoing device for previous node
        store   1       ; rearrange stack
        calls   "ifput" ; store previous node address on device

;; next, check for the existence of the multicast list m_list
prep:
        push    "m_list"     ; find out what the current list is
        calls   "get"
        isx                  ; was there an exception from get?
        bne     no_list - pc ; if so, no list installed, we must create it
        pull    0            ; copy first tuple of list
```

Figure 6.7: SNAP code for `refresh`, part 1.

```
;; loop over m_list to see if the previous node is already in it
loop:
        pull 0            ; extra copy of current tuple
        nth     1         ; extract host part
        pull    3         ; copy of previous node address
        eq                ; are the two addresses equal? if so, we are
                          ; done and just need to check if we should
                          ; propagate towards the server
        bne     doWeNeedToUpdateIntro - pc
        nth     2         ; else, hosts don't match, get next tuple in
                          ; list
        pull    0         ; make copy of next pointer
        bez     end - pc  ; if next == 0 (null) we are not in the list

;; otherwise, we must loop back and check the rest of the multicast list
        push    loop   ; entry point = loop
        push    4      ; 4 stack vals (all of current stack)
        getrb          ; use rest of resource bound
        here           ; dest addr = here
        send           ; send packet here
        exit

;; at this point, we have reached the end of the list without finding the
;; previous node address in it, so we must add it.
end:
        pop                       ; pop null pointer
        pull    1                 ; (previous node)
        pull    1                 ; (first tuple in list)
        mktup   2                 ; cons
        store 1                   ; rearrange stack
        pull 0                    ; copy of new cons cell
        push    "m_list"
        calls   "put"             ; store new multicast list
        ji doWeNeedtoUpdate - pc  ; check for propagation
```

Figure 6.8: SNAP code for `refresh`, part 2.

```
;; no multicast list currently stored, we must create one
no_list:
        pop                 ; pop exception
        push 0              ; push null pointer
        mktup 2             ; cons (previous node, null)
        push    "m_list"
        calls   "put"       ; store the new list

;; since we created the list, we definitely need to move up in the
;; multicast tree
        here                ; moving on. this node must be added on
                            ; next node
        push    first       ; entry point = first
        push    2           ; carry 2 stack vals (whole stack)
        getrb               ; use rest of resource bound
        pull    4           ; server's address
        pull 0              ; copy server address
        here                ; get current address
        eq                  ; compare
        bne avid_exit - pc  ; if equal, we are at server, done
        send                ; else, forward towards server
        exit

;; At this point, we know the previous node address is in m_list.
;; Now we need to see if we need to move on. Two different entry
;; points to this section, one of which needs to pop an extra
;; stack value.

doWeNeedToUpdateIntro:
        pop                 ; pop extra tuple copy
doWeNeedtoUpdate:
        store   1           ; stack rearrangement
        push    "wentUp"   ; see if wentUp variable exists
        calls   "get"
        isx                 ; is exception? variable not defined
        bne     next - pc   ; if exception, need to update
        bne avid_exit - pc  ; else, someone else went upwards
                            ; recently; we are done
```

Figure 6.9: SNAP code for `refresh`, part 3.

```
;; Install updated multicast list. For each host in m_list, see what
;; the outgoing device is, and then check if the linkActive variable
;; exists on that device. If so, this host (or someone else on that
;; device) refreshed recently, so we keep them on the list. Otherwise,
;; we skip them, thus pruning the multicast tree.

next:
        pop                 ; pop exception
        pull    0           ; copy of m_list
        push    0           ; null pointer
        store   2           ; stack rearrangement

loopA:
        pull    0           ; copy of m_list
        nth     1           ; first host in list
        pull    0           ; copy of host
        push    "linkActive"    ; boolean variable stored on iface
        pull    1           ; copy of host in list tuple
        route               ; route to host
        rtdev               ; outgoing device towards host
        store   1           ; stack rearrangement
        calls   "ifget"     ; see if linkActive exists on this device
        isx                 ; if exception, linkActive no longer exists
                            ; (timed out)
        bne     loopC - pc  ; in that case, go to loopC; skip this host
        popi    2           ; linkActive exists, get rid of extra args
        pull    1           ; copy current list element
        nth     2           ; get next list element (cdr)
        pull    1           ; copy host
        pull    4           ; copy previous list element
        mktup   2           ; cons
        store   4           ; overwrite previous tuple in stack
        store   2           ; overwrite next tuple in stack
        pop                 ; stack rearrangement

        pull    0           ; if next is null, done
        bez cleanup - pc
```

Figure 6.10: SNAP code for `refresh`, part 4.

```
        ;; loop back to loopA--next tuple is non-null
        push loopA          ; entry point = loopA
        push 3              ; carry three stack values
        getrb               ; use rest of resource bound
        here                ; loop back to here
        send                ; send packet
        exit

;; skip the current host in the list, as it has not refreshed recently
loopC:
        pop                 ; pop exception
        pop                 ; host of tuple
        nth     2           ; get next pointer

        ;; loop back to loopA
        push    loopA       ; entry point
        push    3           ; carry whole stack (3 values)
        getrb               ; use rest of resource bound
        here                ; stay on same host
        send                ; send packet
        exit

cleanup:
        pop                 ; pop null pointer
        push    "m_list"    ; install new list on node
        calls   "put"
        push    1           ; indicate we went upwards recently
        push    "wentUp"
        calls   "put"       ; wentUp := 1
```

Figure 6.11: SNAP code for `refresh`, part 5.

```
        ;; we refreshed/updated, move on towards server
        here                ; current node will be the previous node of
                            ; the next hop towards the server
        push    first       ; entry point
        push    2           ; 2 stack values: server, current host
        getrb               ; use rest of resource bound
        pull    4           ; server address
        pull    0           ; copy of server address
        here                ; current address
        eq                  ; compare addresses
        bne avid_exit - pc  ; already at server, done
        send                ; else move on

avid_exit:
        exit
```

Figure 6.12: SNAP code for `refresh`, part 6.

```
initial:
        push    <mpeg data>
        push    <frame number>
        push    <packet sequence number within frame>

;; check if this node is a client
get_nodetype1:
        push    "endnode"
        calls   "get"           ; check endnode resident state variable
        isx                     ; check for exception
        bne get_nodetype2 - pc  ; if exception, continue
        bne deliver - pc        ; else, deliver data

get_nodetype2:
        pop                     ; exception
        push    "m_list"        ; load current multicast list
        calls   "get"
        isx                     ; if exception, no list present
        bne full_exit - pc      ; in that case, just drop ourselves
        bne tests - pc          ; else move on to congestion tests

deliver:
        pull    2               ; pointer to mpeg data
        demuxi  <portnum>       ; deliver to client
        ; note: demuxi results in a packet exit

;; test congestion and fragmentation on each interface
tests:
        push    "m_list"
        calls   "get"            ; get multicast list
```

Figure 6.13: SNAP code for `multdeliver`, part 1.

```
congestion_test:
        pull    0            ; copy of list
        nth     1            ; extract host element
        pull    0            ; copy of host
        route                ; determine next hop to host
        rtdev                ; determine outgoing interface
        store   1            ; stack rearrangement
        pull    0            ; copy of interface
        calls   "qlen"       ; get outgoing queue length
        pull    0            ; copy of queue length
        subi    5            ; check for congestion
        gti     0            ; is (len-5)>0? then len>5
        bez     frag_check1 - pc    ; if not, no congestion
        push    <frame weight>      ; weighting for different frame types
        div                  ; queue length / frame weight
        gti     2            ; compare to 2
        bez frag_exit - pc   ; too congested, drop
        push    0            ; else continue (push dummy stack val)

;;;; check for fragmentation
;; see if the frame number is the next in sequence
frag_check1:
        pop                         ; queue length
        push    "frame_num"         ; see what frame number was stored
        pull    1                   ; copy of device
        calls   "ifget"             ; stored per-interface
        isx                         ; if exception, nothing stored there
        bez     frag_check2 - pc    ; else move on to next phase

;; no frame number stored
fix1:
        pop                         ; exception
        pull    4                   ; copy of current frame number
        push    "frame_num"
        pull    2                   ; copy of outgoing device
        calls   "ifput"             ; store current frame number
        pull    3                   ; copy of current packet number
        pull    0                   ; duplicate
        bez     finish_loop - pc    ; first packet of the frame, ok
        pop                         ; packet number
        ji      frag_exit - pc      ; otherwise, fragmentation: drop packet
```

Figure 6.14: SNAP code for `multdeliver`, part 2.

```
;; verify packet number is the next in sequence
frag_check2:
        push    "packet_num"
        pull    2               ; copy of outgoing device
        calls   "ifget"         ; get current packet number for device
        isx                     ; if exception, nothing stored
        bez     check_seq - pc  ; not exception, continue checking

;; no packet number stored
fix2:
        popi    2          ; exception and stored frame number
        ji frag_exit - pc  ; done with this address

check_seq:
        addi    1                   ; add 1 to stored packet number
        pull    0                   ; make copy
        pull    6                   ; my packet number
        eq                          ; compare
        bne     check_frame - pc    ; they match, continue checking

check_seq2:
        popi    2                   ; clear unneeded args
        pull    3                   ; copy of my packet number
        pull    0                   ; extra copy for branch
        bez     finish_loop - pc    ; frame piece 0: don't need to check
                                    ; the frame number
        pop
        ji      frag_exit - pc

check_frame:
        pop
        pull    5             ; copy of our frame number
        pull    1             ; stored frame number
        eq                    ; compare
        bne finish_loop - pc  ; they match, go ahead

check_frame2:
        pull    4               ; copy of packet number
        bez finish_loop - pc    ; if zero, first packet of frame, ok
        pop                     ; else, fragmentation: skip
        ji      frag_exit - pc
```

Figure 6.15: SNAP code for `multdeliver`, part 3.

```
finish_loop:
        pop                         ; extra frame copy
        pull    3                   ; copy of packet number
        push    "packet_num"
        pull    2                   ; outgoing interface
        calls   "ifput"             ; store current packet number
        pull    4                   ; copy of frame number
        push    "frame_num"
        pull    2                   ; outgoing interface
        calls   "ifput"             ; store current frame number
        ji      unroll_exit - pc    ; all done

;; fragmented frame. set things up for next frame
frag_exit:
        push    -1              ; frag, next ok packet number is 0
        push    "packet_num"
        pull    2               ; outgoing interface
        calls   "ifput"         ; packet_num := -1
        popi    2               ; clear extra args
        ji      loopback - pc   ; handle next multicast hop

;; loop unrolling, at most 3 outgoing multicast hops at a time
unroll_exit:
        pop
        push    0               ; zero hosts left
        pull    2               ; copy of multicast list
        nth     2               ; next pointer
        pull    0               ; copy of next host
        bez     one_exit - pc   ; if null, done unrolling
        pull    0               ; else copy next tuple
        nth     2               ; get next next tuple
        pull    0               ; make copy
        bez     two_exit - pc   ; if null, done unrolling

three_exit:
        push    3                 ; 3 hosts in unroll
        ji      two_exit_2 - pc
```

Figure 6.16: SNAP code for `multdeliver`, part 4.

```
two_exit:
        push    2               ; two hosts left to unroll
two_exit_2:
        store   3               ; store unroll number lower in stack
        popi    2               ; clear out extra stuff above it
        store   1               ; stack rearrangement
        ji      sendalong - pc  ; ready for send

one_exit:
        popi    3               ; clear extra
        push    1               ; one host left to unroll

;; extensive stack rearrangement
sendalong:
        pull    4               ; mpeg data
        pull    4               ; frame number
        pull    4               ; packet number
        pull    4               ; multicast list
        store   8               ; store list lower in stack
        store   4               ; move packet number
        store   4               ; move frame number
        store   4               ; move mpeg pointer
        push 3                  ; carry three stack items (mpeg,
                                ;   frame number, packet number)
        getrb                   ; use rest of resource bound
        pull    2               ; number of hosts to send to
        div                     ; divide resource bound
        pull    6               ; copy of multicast list
        nth     1               ; extract host
        push    get_nodetype1   ; entry point = get_nodetype1
        store   4               ; position entry point in stack
        send                    ; send to host
        pull    3               ; copy of list
```

Figure 6.17: SNAP code for `multdeliver`, part 5.

```
;; loop back and finish processing multicast list
loopback:
        nth     2               ; get next host off list
        pull    0               ; copy
        bez     full_exit - pc  ; if null, end of list
        push    congestion_test ; entry point = congestion_test
        push    4               ; carry 4 stack values
        getrb                   ; use rest of resource bound
        here                    ; evaluate here again
        send                    ; loop back

full_exit:
        exit
```

Figure 6.18: SNAP code for `multdeliver`, part 6.

Chapter 7

Future Work

In this dissertation, we have demonstrated that it is *possible* to build a practical active packet system. However, the question of what the *best* active packet system might be has barely been touched. There is much work to be done, and experience to be gained, before we can begin to weigh in on the various design tradeoffs this choice entails. In this chapter, we explore two main directions for further work: improvements on the SNAP system itself and experimentation with the SNAP active packet model and its assumptions.

7.1 Improving the SNAP system

One obvious tactic for exploring the design space of active packet systems is to continue to improve upon the SNAP system we have presented here. Here, we subdivide directions for improving SNAP into four main avenues: language improvements, performance enhancements, application development, and the expansion of node-resident services.

7.1.1 Language improvements

There are a few features that would be convenient additions to the SNAP language but that would not break our resource predictability claims. In large part, these are informed by a combination of programming experience with our applications and developing the PLAN-to-SNAP compiler. The three main features we consider here are better exception handling, a loopback instruction, and stricter global hop count guarantees.

Exceptions. Right now, SNAP has a very primitive notion of exceptions: when an exception occurs, an exception value is placed on top of the stack. A program must then use the **isx** instruction to test whether the top stack value is an exception or not. Since this is the only way to detect exceptions, the PLAN-to-SNAP compiler must insert a check at every point where an exception could be raised. This has two main drawbacks: first, it requires inserting extra code, which is always a potential problem for space-limited packet programs. Secondly, it requires additional execution for the common (exception-free) case.

It should be fairly straightforward to add a slightly more convenient (yet still simple) mechanism around two new instructions: **setxh** ("set exception handler") and **raisex** ("raise exception"). The SNAP interpreter would be augmented to contain an exception handler pointer, which would be initially set to point to the end of the code segment. A program could register a block of code via the **setxh** instruction, storing the code address in this handler pointer. Whenever an exception occurs, either during execution or via **raisex**, control would be transferred to the currently registered handler, with the restriction that the resulting branch moves forward through the code (thereby maintaining our linear CPU time guarantee).

Thus, programs that did not wish to handle exceptions would not have to bother checking stack values with **isx** (they would just terminate if an exception was raised), yet the PLAN `try...handle` construct could still be naturally translated, as shown in Figure 7.1. On entering the `try` portion of the expression, the program first registers the block associated with the `handle` portion as the current exception handler. If no exception is raised during execution of the `try` block, then we simply branch beyond the end of the `handle` block and continue. If an exception is raised, the exception value is pushed on the stack, and we branch to the `handle` block. The handler first compares the current exception with the one it expects to handle, and if they match, we enter the `handle` expression proper. Otherwise, we set the handler register to the address of the next enclosing handler[1] and then use **raisex** to throw the exception up to it.

[1]Note that the the "next enclosing handler" is always statically known due to the nature of PLAN exceptions and the fact that the PLAN-to-SNAP compiler is a whole program compiler. Thus, we can reorder the basic blocks as necessary, using the same topological sorting techniques already in use by the compiler.

PLAN:

```
try
  /* block 1 */
handle e =>
  /* block 2 */
```

SNAP:

```
    setxh handler
    /* translation of block 1 */
    ji handle_end - pc

  handler:
    pull   0    ; make copy of exception
    eqi    e    ; is this the exception we
                ;   expect to handle? if
                ;   so, enter handle_body
    bne    handle_body - pc
    setxh  enclosing_handle
    raisex      ; else, throw up to next
                ;   enclosing handler

  handle_body:
    /* translation of block 2 */

  handle_end:
```

Figure 7.1: Translation of PLAN `try...handle` block using **setxh** and **raisex**.

Loopback and other CISC-style abbreviations. As we saw in Chapter 6, it is possible to encode a backward branch as the six-instruction sequence shown in Figure 7.2. We could imagine providing a **lback** ("loopback") instruction as an abbreviation for sending a copy of the packet to the current host with the given branch target as an entry point. This would be in keeping with the **forw** and **forwto** instructions that are also abbreviations for longer instruction sequences based around the more universal **send** primitive.

Generally speaking, as experience with application development using SNAP grows, other common instruction sequences may emerge, also becoming candidates for such "abbreviation" opcodes. Currently, space in the packet is at a premium, and so a CISC-style approach to the SNAP instruction set will help reduce code sizes. Indeed, this is reminiscent of early general-purpose computers with limited main memories. It will be interesting to see if trends toward larger MTU sizes [Cur98] will result in a similar move towards higher-performance RISC-style active packet instruction sets.

```
push    <entry point>  ; backward branch target
push    -1             ; take the whole current stack
getrb                  ; use remaining resource bound
here                   ; destination is same node
send
exit
```

Figure 7.2: Six-instruction sequence for emulating a backward branch.

Better hop count guarantees. Whitaker and Wetherall [WW02] propose a scheme in their Icarus active networking environment to prevent malicious looping or implosion denial-of-service attacks by active packets. The key idea is to guarantee loop-freedom both for unicast packets (thus preventing "ping-pong" attacks) as well as for more general multicast scenarios (such as fanning out and then converging on a target). The approach revolves around carrying a Bloom filter [Blo70] in a packet. An initial empty filter would just be a zeroed-out n-bit field in the packet. Each link in the network would have a Bloom mask where only k of the n-bits are set (with $k << n$). This mask is bitwise ORed with the Bloom filter in the packet, so the packet's filter slowly "fills up" as it traverses the network. If adding in the current interface's mask does not change the filter, it is likely[2] the packet has already visited this interface and it is dropped. Similarly, each node has a "designated ingress interface" for multicast packets, such that only copies of the packet arriving on this interface are allowed to propagate. Whitaker and Wetherall also propose variants to reduce the effect of false positives.

It seems clear that a scheme like this could be adapted to SNAP, thus solving the dilemma of how to treat TTL for multicast while still allowing packets the freedom to route themselves if desired. Looping behavior would be caught by the Bloom filter (although it is necessary to restrict the packet to only one **send** on each interface). Similarly, fanout for multicast would be allowed, with each child getting a copy of the parent's current RB (allowing for easy programmability of a "radius of effect"). However, subsequent implosion would then be prevented as well. One drawback of this scheme is that it destroys some

[2]Here, "likely" depends upon the particular selections of n and k.

of the global resource predictability offered by SNAP for the same reasons that a native SNAP multicast instruction was considered unsafe.

7.1.2 Performance Enhancements

While our current in-kernel implementation is fast enough to prove our efficiency claims for routers for 100 Mb/s links, there are a number of possible improvements we could make to provide better performance. These enhancements might be necessary to achieve full line rates for higher speed networks (*e.g.*, gigabit Ethernet or optical networks).

One-byte instruction representation. The current wire representation for SNAP uses 4-byte instructions, with 7 bits indicating an opcode, and the other 25 bits used for an immediate operand. While this means all instructions are the same size and aligned on a 4-byte boundary, there are two main drawbacks. First, significant space is wasted for instructions without immediate operands, and secondly, the precision of the immediate operands is limited to 25 bits.

One possible solution would be to encode each instruction as a 1-byte opcode, with full 4-byte immediate operands inlined where appropriate. It would even be possible to allow instruction variants using only 1- or 2-byte immediates (*e.g.*, **pushb** for "push byte", **pushhw** for "push half-word"). The main consequence would be that instructions would no longer be uniformly sized, and would not necessarily be aligned on a 4-byte boundary. Thus, branch offsets would have to be expressed in terms of numbers of bytes rather than number of instructions.

SNAP's safety would not be adversely affected by this change, as the length of the code segment still places upper and lower bounds on the number of instructions it can contain, and each instruction can still only be executed once per local packet execution. There is the possibility that an incorrectly calculated branch offset could land the program counter in the middle of an immediate operand, but SNAP must already deal with unrecognized operands for robustness purposes. Thus, the general sandboxing philosophy we have taken with SNAP will still serve in this case.

The main disadvantage of this change is that the loss of 4-byte alignment may make efficient interpreters (particularly hardware implementations) more complex. Fortunately, one would expect that SNAP would have to become fairly widespread before router vendors consider high performance SNAP implementations, so we will likely have more experience to judge whether the code size savings are worth the added implementation complexity.

Tagless stack values. Our current representation for stack values reserves 7 bits as a type tag, leaving 25 bits for the value itself. The type tag is used in three main ways: first, to allow easy implementation of **isx** ("exception" is one of the possible types); second, to perform dynamic type checking of operands; and third, to allow precise garbage collection (GC) of the heap during marshalling for a packet send.

Unfortunately, this means that 32-bit values must be *boxed*, or represented as a pointer to a 32-bit heap object. Most notably for us, this means that IPv4 addresses must be boxed, which means that heap allocation occurs whenever a program manipulates network addresses—not an optimal situation for a packet programming language.

Instead, we could remove the type tags and just have untyped 32-bit stack values. With the addition of the **setxh** and **raisex** instructions described above, we would no longer need to support **isx**. Furthermore, dynamic type checking is not strictly necessary for our notion of safety. It does catch program errors earlier than simple sandboxing does; without type checking, a type error would most likely cause the program eventually to attempt to access an out-of-bounds heap area, branch to an invalid address, or execute an invalid opcode. Without type checking, we would still need dynamic checks in the SNAP interpreter to catch all of these violations and enforce the sandbox.

However, this would mean we could not do exact garbage collection of the heap when marshalling for packet transmission. There are a variety of approaches we could take here. One obvious question is whether we need GC at all; it is not yet clear whether the time spent on this is worth the savings in space in the packet. Another possibility would be to perform some form of conservative garbage collection [BW88].

Currently, heap allocation takes place in a separate area at the end of the packet buffer, and the heap offsets for pointers into this area are calculated as offsets from the beginning

of the original packet heap, as this simplifies pointer arithmetic for heap access. However, we could instead use an additional buffer off to the side for heap allocation, and calculate the new heap offsets as if the new area were contiguous with the original heap.

Now, once this scheme was in place, marshalling for a packet send would simply be a matter of scatter/gather of three memory areas: the start of the buffer through the end of the original heap, the new heap, and then the stack. If we chose to omit GC, at this point the packet would be ready to send. Unfortunately, a conservative GC may not be likely to reclaim much space, as the collector will not be able to distinguish pointers from non-pointers in the stack. Therefore, potentially pointed-to heap objects may not be moved. Thus, even if we detect a lot of garbage in the middle of the heap, we cannot compact it, so the only potential space that could be reclaimed would be in a contiguous region at the end of the heap.

One final possibility would involve reusing the upper 4 bits of a stack value. First, recall that we do not allow native multicast sends in SNAP, so a multicast IPv4 address will not be valid. It just happens that all IPv4 multicast address have their upper 4 bits set to `1111`. Therefore, we can use the following conventions:

1. For most values, we will use 28 bits of precision, and we will use the upper 4 bits as a type tag. Heap offset pointers will have these bits set to `1111`, and all others (integers, exceptions, *etc.*) will take any different value.

2. Addresses use the full 32 bits (however, we know that no valid address will have their upper bits set to `1111`).

Thus, when we scan the stack or the heap during GC, a value will have its upper 4 bits set to `1111` if and only if it is a heap pointer or an invalid IPv4 address. Therefore, this scheme would provide exact GC for correct programs, but may not collect all dead heap objects for invalid ones (namely, ones using multicast IPv4 addresses).

Threaded interpretation. The current structure of the SNAP interpreter can be represented by the pseudo-C code shown in Figure 7.3. For each instruction, we extract the opcode and then enter a large `switch` statement where each opcode has its own `case`.

```
while (pc < code_end) {
  switch (get_opcode(pc)) {
    case op1:
      /* execute opcode 1 */
      pc++;
      break;
    ...
    case op2:
    ...
  }
}
```

Figure 7.3: Structure of the current SNAP interpreter loop.

Now, suppose we have an instruction sequence with **op1** followed by **op2**. After executing **op1**, the interpreter will make three branches: first, to the end of the `switch` statement; second, to the top of the `while` loop; and third, to the `case` for **op2**.

This inefficiency in virtual machine implementations is often addressed by the use of *threaded code interpretation* [ML70, PR98]. In this case, we would construct a jump table mapping opcodes to the pieces of code that execute them. We would then include the decoding of the next instruction at the end of executing the previous one, as shown in Figure 7.4. This would reduce the number of branches to two: one for the `if` statement to detect the end of the code segment, and one to branch to the code for executing the next instruction.

Static approaches. We have operated on the assumption that, especially for ephemeral active packets, large static optimization or safety operations would be unlikely to pay off in terms of reduced execution time. For example, we decided that dynamic sandboxing would be a cheaper way to enforce safety than performing either static type checking or proof-carrying code (PCC) [Nec97] techniques. It would be interesting to actually explore this particular question experimentally.

Another interesting question would be whether some sort of just-in-time (JIT) compilation would be worthwhile. Kind *et al.* have actually implemented such a compiler [KPS02] for a variant of SNAP running on IBM PowerNP 4GS3 network processors [IBM02]. Their

```
jump_table[op1] = &label_op1;
jump_table[op2] = &label_op2;
...

while (pc < code_end) {
  goto *jump_table[get_opcode(pc)];

  label_op1:
    /* execute op1 */
    pc++;
    if (pc < code_end)
      goto *jump_table[get_opcode(pc)];
    else
      break;

  label_op2:
    /* execute op2 */
    pc++;
    if (pc < code_end)
      goto *jump_table[get_opcode(pc)];
    else
      break;
  ...

}
```

Figure 7.4: Structure of a threaded SNAP interpreter.

findings indicated that compiling one SNAP instruction and then executing the resulting native code took slightly longer than just interpreting the same instruction. Thus, JIT compilation would probably not pay off with our current one-shot instructions. However, Kind *et al.* note substantial speedups when the generated code contained loops[3] or was cached between packets of the same flow.

In both of these cases, our intuition has been that the large static overhead would not be overcome: since each SNAP instruction executes at most once, and in some cases, *not at all*, a potentially large part of this static overhead would be wasted work. Before we can evaluate this experimentally, however, we need a larger body of common, useful SNAP programs to use as benchmark cases.

Code caching. Another possibility would be to couple either of the above static approaches with code caching. We could include a code checksum field in the SNAP header to use as a key to the cache. Indeed, having such a code checksum might be desirable anyway, just to avoid executing code segments that were corrupted during transmission (although again, this is not strictly necessary for safety, as our sandboxing techniques will catch these errors).

Then, when a packet arrives, we verify its checksum and see if we have any native code cached for it. If so, we can immediately jump to the native code (perhaps we would need to verify that the first few instructions matched between the bytecode and native versions to avoid checksum collisions). On a cache miss, we can begin the native code compilation in parallel with interpretation, as the code will still be carried in the packet, perhaps waiting for a specified number of misses before compilation so as to avoid compiling "one-shot" diagnostic or control packets. This is in contrast to the ANTS [WGT98] code-caching scheme where a capsule's code *must* be fetched into the cache before being executed, incurring a one-hop round-trip to the previous node to fetch the code, and potentially creating thrashing in the code cache.

[3] Kind *et al.* use a variant of SNAP that allocates an execution budget similar to that of StreamCode [EHH01] (namely, linear in the length of the entire packet including payload) and then permits looping up to this point.

Another possibility would be to cache the bytecode itself, particularly for slow links, in the style of TCP/IP header compression [Jac90]. Here, instead of transmitting the same code segment over and over again for a stream of packets, we would instead just transmit a code segment identifier like the code checksum mentioned above. Two problems spring to mind here: first, we may end up with thrashing in the cache that may result in a packet arriving with no associated code. Secondly, this potentially violates some of our resource guarantees, as the resources an attacker can currently tie up with SNAP are directly tied to his or her access bandwidth. With code caching, an attacker need only send one copy of a large malicious program (particularly one with a small payload) and can then send many more smaller packets containing just payloads, thereby magnifying the effect of the access link.

Scalable Implementations. In Chapter 5, we measured SNAP's efficiency in a software-router setting, which we argued was already ubiquitous enough to make a substantial impact. However, it would be interesting to examine SNAP implementations designed for higher-end routers. One obvious strategy here would be to build hardware SNAP interpreters; given the simplicity of the language and our current virtual machine, this should be straightforward, and we would expect to be able to draw on the experiences of other hardware active packet environments like StreamCode [EHH01].

Another attractive area is implementing SNAP interpreters for the micro-engines of network processors [IBM02, Cor01]. Indeed, Kind *et al.* have already implemented such an interpreter for a variant of the SNAP language [KPS02]. Although they have not made real-time throughput measurements, the architecture they describe involves having network processors interconnected by a switching fabric. Here, the SNAP processing power of such a router would scale with the number of network processors available. Kind *et al.* correspondingly changed the SNAP execution model to have both ingress and egress code entry points to fit more closely the underlying hardware architecture; thus, a SNAP packet executes on its ingress network processor, determines its output port, gets routed across the switching fabric, and finally executes on the corresponding egress network processor.

7.1.3 Application Development

Application development is an exciting direction for future work, as it will enhance our experience with SNAP and inform the community about various design space tradeoffs we have made. Furthermore, we need to develop a suite of benchmark SNAP programs to carry out some of the experimental research mentioned above. Here we will limit ourselves to extensions of the two main applications we described in Chapter 6.

Network management. At the time of this writing, Walter Eaves at University College London has added an SNMP interface to the user-space version of SNAP [ECG+02]. However, we have not yet had a chance to consider the resource implications (as in Chapter 4) of these services. Nonetheless, a combined user-space SNAP interpreter and SNMP daemon would make an attractive drop-in replacement for a simple SNMP daemon. As we mentioned in the previous chapter, we could then roll out mobile-agent-style replacements for various centralized-polling monitoring applications. Indeed, since control-plane operations are often less performance-sensitive than data-plane ones, this would seem to be one of the main avenues for getting SNAP widely deployed in the Internet.

Multimedia. Beyond the AVid system we described here, we could build other applications with active packet programs designed specifically for the type of media we were transporting. For example, for audio streams, perceived sound quality is improved if losses are spread out rather than clustered together. Thus, we could have audio packets that monitored their outgoing queue lengths and aborted themselves *before* the queue filled up entirely, much as in Random Early Detection (RED) [FJ93].

Similarly, interactive voice applications have hard real-time constraints on the delay a packet may encounter. For example, an active packet could maintain a conservative estimate of its cumulative queuing delay. The estimate of the queuing delay at the current node can be calculated based on the length of the outgoing queue and the average rate at which the queue is draining, both of which are reasonable quantities for a node to be maintaining [MK00]. If the packet can determine at any point that it will miss its overall

delay deadline (either definitively or perhaps with a certain reasonable probability), then the packet can drop itself, thereby freeing up resources for other packets in the stream.

In general, multimedia applications are likely to be a great source of new applications for active packets (or active networking in general, for that matter). Each different media type has its own properties, and different network contexts ranging from Ethernet LANs to dialup modems to cellular phones will mean that a variety of new protocols will be needed. An active packet framework, particularly one that is incrementally deployable like SNAP, will provide a convenient way for designers to quickly roll out new protocols without having to worry about packet formats.

7.1.4 Services

As we mentioned in Chapter 4, an active packet language is really only as good as its set of node-resident services, although just a few well-selected ones suffice for a variety of applications. Currently, services are implemented in C and verified by hand for their resource usage properties. Furthermore, they are statically compiled into the kernel. To realize the full flexibility for which the original active networks proposals aimed, a much better service support framework will be necessary.

First, we need a way to dynamically deploy services, whether this takes place via transmitting and loading kernel modules or by a more general form of online software updating [Hic01]. We then need a way to make sure that the newly deployed services are safe. One method involves requiring downloaded code to be cryptographically signed by a central authority [CDD+99]. However, this really only handles authorization and not safety: well-meaning central authorities may still sign software containing bugs.

Instead, we need a way to verify the safety of new services. The use of proof-carrying code techniques [Nec97, MWCG99] would enable code to be shipped with a proof that it satisfied our robustness and resource usage conditions for safety. Even in this case, we would still also want the code to be at least cryptographically signed for access-control purposes.

As we mentioned in Chapter 4, the programming language community has developed several techniques for estimating or asserting the resource usage properties of programs [CW00, NL97, Hof99, HP99, HPS96, RG94]. While it is undecidable to automatically derive the resource usage properties of a program in a general purpose programming language, the process may be easier in a domain-specific language. For example, programs written in PLAN [HKM+98b] and Mercury [SSS97] are guaranteed to terminate, and we have shown even stricter guarantees for SNAP in this work. Therefore, it seems likely that we could design a special-purpose service programming language, in which resource usage proofs could be automatically generated and transmitted in a proof-carrying code framework.

7.2 Expanding the model

One of the main contributions of this work is our model of active packet execution. Perhaps the main concept of this model is its notion of linear resource usage. A main reason for choosing this model is that unicast IP packets satisfy this model; therefore it seems that a linear bound for active packets would also be acceptable to the network community. However, there is some question whether this bound might be too tight. For example, multicast IP packets do not satisfy our global predictability criterion.

It would be worthwhile to take the general model of the resources used for packet processing as presented in Chapter 4 and apply it to current network protocols[4]. This would enable us to get a feel for what bounds are *really* acceptable to the networking community. In particular, it could potentially make a compelling argument that deploying active protocols is no more demanding on a node's resources than current, widely-accepted protocols.

Finally, if it turns out that resource usage greater than a simple linear bound is acceptable, it will open up a new portion of the design space for active packet languages, as we may be able to increase flexibility beyond what SNAP provides without sacrificing safety. For example, it may become possible to remove the backward branch restriction for SNAP, or perhaps to permit statically-bounded loops. Perhaps more likely, it may allow

[4] Personal communication with Michael Hicks.

services that do not run in constant time or space, or ones that allow a packet to "refuel" its resource bound.

7.3 Summary

This dissertation has shown that an active packet system can be practical, something the active networking community has been working towards for several years. However, as we have mentioned above, this does not close the book on active packet systems. Although we have shown SNAP to be practical, there are a number of ways in which it could be improved. Furthermore, one could easily imagine that there are other languages that might satisfy our model of linear resource usage. In this chapter, we have outlined some of the next steps that can be taken in the exploration of the design space for *practical* active packet systems.

Chapter 8

Related Work

This section examines the main existing active packet systems: ANTS [WGT98], PLAN [HKM+98b, HMA+99], PAN [NGK99], Smart Packets [SJS+99], ALIEN [Ale98], MØ [Tsc97], SafetyNet [WJOP00], and StreamCode [HEK00, EHH01]. We describe each system and consider how it fits into our practicality framework. We examine how each system trades off among the three criteria of safety, efficiency, and flexibility and evaluate its strengths and weaknesses.

8.1 ANTS

The Active Network Transport System (ANTS) [WGT98] was one of the first active packet systems developed. Active packets, or *capsules*, logically contain the code that is needed to process them. This code may make calls into a fixed and limited API to gain node-specific information. In addition, related capsules may leave node-resident state for one another in a soft-state cache.

One of the unique features of the ANTS system is an on-demand code loading system. Rather than carry the code itself, the capsules instead contain references to the code. If a node does not have a cached copy of the necessary code, it is loaded, typically from the previous node in the flow, but potentially from the source node. Thus the code for an active application marches 'ant-like' across the network as the caches are loaded.

Safety. The current ANTS prototype is written in Java [GJS96] and relies upon the JVM's bytecode verification and sandboxing facilities [LY96] for the safety features they

provide. Local node resource usage is governed through the use of watchdog timers and memory allocation limits. Capsule types are grouped into protocols, and capsules are restricted to only access soft state belonging to their own protocol. The reference to the code actually takes the form of a MD5 cryptographic hash of the actual code, thus preventing code spoofing. Thus, a misbehaving capsule is isolated from other capsules and the node itself, and if it consumes too many resources it is terminated. However, Hawblitzel *et al.* [HCC+98] note that terminating a misbehaving thread or process safely in the JVM is, in fact, far from simple, as node data structures may be left in an inconsistent state or node threads may be "hijacked" by the misbehaving capsule.

To control network-wide resource use, ANTS provides a TTL field, which is decremented at each hop, and duplicated when a packet creates a child packet. Since packets can create any number of child packets, the TTL limits the distance a packet's children can travel, but not the total work in the network. Currently ANTS requires that capsules be certified by some authority before being deployed to limit the possibility of malicious packets. This significantly reduces the speed at which new capsule programs can be deployed, and of course raises the question of the correctness of the certification.

Efficiency. Wetherall [Wet99] provides an in-depth look at the costs involved in processing ANTS capsules on a Sun Ultra 1 running Solaris 2.6 (SPECINT95 rating of 6). Per-capsule processing latency ranges from 500 μs for zero-payload capsules to 700 μs for capsule payloads resulting in maximum Ethernet frames (1500 bytes, resulting in 16 Mb/s application throughput). For 512-byte capsules, Wetherall measures that of the 615 μs latency, 42% may be attributed to kernel crossings and thus could be eliminated from an in-kernel ANTS implementation. An additional 32% is attributed to marshalling costs (casting capsule byte arrays into Java objects and vice versa). All of these numbers are for capsules that do not require demand loading (*i.e.*, the capsule code is already cached); when loading is required the latency increases by an order of magnitude [WGT98]. However, Wetherall argues that capsules in a given network flow will likely require similar processing, so at 16 Mb/s for the in-cache case, this early ANTS prototype already achieves performance adequate to T1 line rates.

Flexibility. Perhaps the best demonstration of the flexibility of the ANTS system is the number and variety of active networking applications that have been built using the ANTS toolkit. These include specialized web caching [LG98], reliable multicast [LGT98], and distributed auction services [LWG98]. ANTS nodes may be scattered throughout the network; legacy routes may just perform default IP forwarding, thus permitting incremental ANTS deployment. One scalability drawback is that the MD5 fingerprint of the protocol code makes capsule code re-use difficult: in order to combine capsules from two different protocols and have them cooperate, their code must be "glued" into a new protocol, with a corresponding new MD5 fingerprint. Thus, there could be multiple copies of the same code residing in a given cache, which would be problematic if a proliferation of application protocols resulted in widespread cache contention.

8.2 PLAN

The Packet Language for Active Networks (PLAN) [HKM+98b] was another early active packet system and is, in part, the work of the current author. This system features a two-level architecture consisting of limited (but not fixed) node-resident services (similar to the ANTS node API) and active packets written in a domain-specific language, PLAN. A key design goal was to restrict the expressiveness of the PLAN language so that it was safe to execute unauthenticated PLAN programs.

PLAN is a simple, strongly typed, functional language with syntax resembling that of Standard ML [MTH90]. It adds facilities for spawning new packets as well as a unique language construct called a *chunk* [MHN99], which permits packets to be treated as data, thus enabling fragmentation and encapsulation.

Safety. PLAN is strongly typed; this is enforced dynamically by the PLAN interpreter. However, PLAN programs may be typechecked statically so that programmers may know *a priori* that their programs will not create type errors in the network. A key feature of the language is that general recursion is disallowed, thus guaranteeing that all PLAN programs will terminate. Unfortunately, it is still possible to write PLAN programs that consume CPU or memory resources which are exponential in the length of the program [HKM+98b],

so a simple termination guarantee is not strong enough to provide the resource predictability we would like.

Network bandwidth is governed by a strictly decreasing *resource bound* carried by each packet. When a packet spawns a child packet, it must donate some of its remaining resource bound to the child, thus bounding the total number of network hops which can be taken by a packet and its descendants. Unfortunately, it is difficult to know how to set this bound, and there is currently no provision for preventing the bound being set unreasonably high when the packet is first injected. Finally, PLAN's service namespace may be dynamically restricted or expanded based upon cryptographic credentials [HKS02].

Efficiency. Hicks *et al.* implemented an active internetwork in OCaml [LRW99], PLANet [HMA+99], in which all of the packets contain PLAN programs and data. PLANet was able to sustain 48 Mb/s of routing throughput[1], through the use of a special "routing function" pointer in each packet that allowed packets to evaluate their PLAN programs on selected nodes while simply being forwarded through others. The main savings here was being able to avoid packet marshalling costs on the routers where PLAN evaluation did not occur. The main overheads identified in PLANet were kernel crossings, thread scheduling, and garbage collection.

Flexibility. Despite being a language with limited expressiveness, PLAN has been found useful as a "network scripting language" for combining node-resident services. Examples of PLAN-based applications include application-specific routing [HMA+99], multicast (where each packet carries a list of all recipients) [HKM+98a], and virtual private networks [MHN99]. One awkward point, however, is that the resource bound conservation properties make it difficult (from a programmer's point of view) to apportion a packet's resource bound among multiple children.

[1] Measurements taken on 300 MHz dual Pentium IIs with 256 MB RAM.

Packet type	processing latency 128 bytes	1500 bytes
kernel IP	50 μs	160 μs
kernel active x86	70 μs	180 μs
user-space active x86	133 μs	290 μ s
user-space active Java	448 μs	N/A

Table 8.1: Packet processing latencies for 128-byte PAN capsules.

8.3 PAN

PAN [NGK99, Nyg98] is a follow-on project to ANTS, also developed at MIT. The main question Nygren *et al.* address is: are the computational overheads of providing active processing too high to ever achieve practical performance? Fortunately for us, the answer was "no." PAN achieves its high performance through the use of in-kernel packet execution, code caching, and standard network performance tuning such as minimizing data copies. A unique feature of PAN is its support of multiple mobile code systems, including Java and x86 object code.

Safety. PAN segregates its soft state on a per-code-object basis, thus providing better flow isolation than the global unified store of ANTS. Furthermore, the use of module thinning permits code objects to export narrowed interfaces and supports *guardian objects* to mediate access to shared data structures (*e.g.*, routing tables). For its Java-based mobile code objects, PAN inherits the type safety properties of the JVM, although dynamically linking x86 object code into the kernel is completely unsafe and hence impractical. This is in fact unfortunate, as the performance numbers presented below seem to imply that safety must be sacrificed in order to achieve efficiency.

Efficiency. By residing in-kernel and minimizing data copies, PAN avoids the kernel crossing overheads suffered by most other active packet systems. Indeed, when using the x86 native code, PAN is able to effectively achieve 100 Mb/s throughput. Nygren *et al.* identify overheads associated with various parts of their system, as shown in Table 8.1[2].

[2]Measurements taken on 200 MHz PPro, 64 MB RAM.

PAN incurs as little as 13% overhead over IP for active processing with unsafe native code. However, switching to Java to provide even basic type safety results in much poorer performance (a significant part of this overhead is attributed to garbage collection costs). Nonetheless, PAN demonstrates that active processing *in itself* does not imply poor performance.

Flexibility. PAN's support for multiple mobile code systems is unique among first-generation active packet systems. The use of native object code provides essentially unlimited flexibility (including, unfortunately, the subversion of the node itself). However, code objects are restricted to a fixed API, the PAN Node Interface, and the use of a type-safe language ensures code cannot bypass this interface, thus having essentially the same flexibility as ANTS in this respect.

8.4 Smart Packets

The Smart Packets project [SJS+99] from BBN targets active packets for network management tasks. One of the goals of their mobile agent-style approach is to cut down on reporting traffic by allowing Smart Packets to digest the information retrieved from nodes to customize the reports that are sent back to the management center. Since many network management functions can already be expressed as simple algorithms, a Smart Packet could diagnose a problem "on-site" and take immediate action to correct it, thus improving management response time.

Because Smart Packets are meant to be deployed in potentially misconfigured or failed networks, they must be extremely robust. In particular, they have been designed to be self-contained, so that no new router state is required. Furthermore, useful programs should be encodable within a single link layer frame so that fragmentation may be avoided. Smart Packets are coded using two equivalent languages: Sprocket is a safe subset of C extended with primitives for MIB access; Sprocket, in turn, may be compiled to Spanner, a compactly-representable CISC-like assembly language.

Safety. The Smart Packets project, like many of the other systems we have discussed, relies on its virtual machine execution environment to provide basic type and memory safety. In addition, the VM enforces limits on the number of instructions executed, amount of memory used, and access to MIB variables. Cryptographic authentication is used to ensure program integrity and origin and to set resource limits. These certificates may be omitted, in which case the packet runs in a very resource-limited environment that is nonetheless sufficient to perform simple diagnostic functions such as *ping* or *traceroute*.

Efficiency. As we will see in our discussion of ALIEN and SANE below, the use of cryptography for authentication can be quite expensive. However, as the main application domain of Smart Packets is network management, being able to operate at the control plane rather than the transport plane may mean that Smart Packets may nonetheless be fast enough. Unfortunately, basic performance numbers are not available at the time of this writing.

Flexibility. Unlike ANTS, PLAN, and PAN, Smart Packets do not have a general packet-spawning capability. Instead, Smart Packets can create Message Packets to report data back to their source. Smart Packets are, however, able to control their course through the network, having both hop-by-hop and end-to-end modes of execution. Finally, providing MIB operations means that Smart Packets should be able to perform most SNMP-style management tasks.

8.5 ALIEN and SANE

ALIEN [Ale98] is an active networking architecture developed at the University of Pennsylvania that supports both active packets and active extensions (node updates). Here we will discuss one particular instantiation of ALIEN, called SANE (Secure Active Network Environment) [AAKS98]. SANE active packets are written in OCaml [LRW99], a dialect of ML supporting dynamic linking of bytecodes.

SANE uses public key cryptography to establish security associations between neighboring nodes. Using these associations, it is possible to guarantee the origin and integrity of the OCaml bytecodes that are the code for the active packets.

Safety. OCaml bytecodes are known to be typesafe [LR98]. ALIEN provides additional safety by using *module thinning* to restrict the interface to a node as seen by the active packets being dynamically linked against it. However, ALIEN does not address resource predictability issues such as limiting the CPU, memory, or bandwidth used by an active packet.

Efficiency. One of the main results of the SANE work is that using cryptographic authentication on every packet in a software implementation results in unacceptably low performance: roughly 5 Mb/s throughput for 1500-byte packets [Ale98][3]. The latency for a one-hop ping is roughly 10 ms. Alexander measures that roughly 25% of the latency is attributable to authentication, 20% to OCaml-induced overheads such as the OCaml thread scheduler and garbage collector, and 16% to marshaling costs. The avoidance of these costs was an important goal in SNAP's design.

Flexibility. Because OCaml is a general-purpose programming language, SANE's active packets are quite expressive. Indeed, the ability to support both active packets and active extensions makes ALIEN a good platform for exploring several different approaches to active networking. However, ultimately the flexibility of the ALIEN system depends upon exactly which functions are made available in the module-thinned API. Because ALIEN relies heavily upon the security of this interface, it is a very fine line to walk when considering whether a useful function is safe enough to be exposed in the API.

8.6 MØ

The MØ [Tsc97] mobile agent environment is under development by Tschudin at Uppsala University. MØ agents are called *messengers* and are meant to provide a basis for developing

[3]Measurements taken on 433 MHz DEC Alphas.

distributed systems. The idea is that hosts should only provide services that do not require synchronization with other hosts; everything else should be built up using messengers.

M∅ is a stack-based language similar to Postscript. Messengers are designed to be lightweight, as it is expected they should be numerous and perhaps short-lived. Messenger threads are anonymous; they may only interact or synchronize through the use of a per-node shared memory which implements a non-browsable dictionary [Muh98]: only a messenger knowing the required key may access data stored there.

Safety. As mentioned above, the M∅ environment provides active packet isolation, and the M∅ interpreter provides basic integrity safety. Resource safety is provided through the use of a market-based system; each messenger which arrives is given some amount of "startup money". Messengers may then bid in a lottery system for CPU time slices, and must pay periodic "rent" on global memory. Messengers may cooperate by donating funds to one another. Threads are killed when they run out of money, and memory is reclaimed when there is no money left for rent. Unfortunately, there is no way to kill an out-of-control messenger, other than by letting its resources run out. It would be possible under this scheme, then, for an application to continually insert new messengers which donate their startup cash towards a malicious, long-running messenger. Furthermore, application designers may be encouraged to take this approach of generating additional bandwidth just to keep their services alive, even if no one is using them.

Efficiency. M∅ has support for running native code, including primitives for querying the available CPU types, setting registers, and defining address spaces. In addition, the M∅ language is designed to be quite compact, leaving more room in active packets for payload. The effectiveness of this platform can be seen from an experiment in which an election service deployed itself around the world over 5 ABONE nodes (6 active hops, 160 Internet hops) in 455 ms [Tsc99b, Tsc99a]. Unfortunately, not enough details are available about this experiment to allow us to directly compare M∅ with the other systems.

Flexibility. Because messengers may send arbitrary strings across communication channels, they essentially have control over their own wire formats. In fact, code caching can

actually be coded up within the MØ environment itself. Furthermore, messengers can express their own security policies, as they have access to cryptographic functions, as well as a special primitive called *elaunch*, which takes a string, decrypts it with the node's private key, and then executes it as an MØ program. Thus it is possible to ensure that certain MØ code is only executed upon the nodes desired.

8.7 SafetyNet

Wakeman *et al.* at University College London propose a new programming language for active networks called SafetyNet [WJOP00]. As implied by its name, this system holds as a primary goal that active networking programs should not be able to disrupt the basic connectivity of the network. Since routing tables are the key component of maintaining connectivity, SafetyNet programs do not have direct, unlimited access to them but rather must access them through a mediated interface. Although SafetyNet is very much a work in progress, we present here the major design goals as another tradeoff point in the design space.

Safety. The approach taken with SafetyNet is to rely on a specialized static type system to guarantee various properties about programs written in it. Besides simple type safety, which provides node integrity from dangling or forged pointers, the SafetyNet type system is also able to express resource usage through the use of *linear types*. Simply put, variables of linear type can only be used once, thus permitting a kind of resource accounting. Wakeman *et al.* claim that limits on network hops, thread spawning, memory allocation, CPU slices, *etc.* can all be encoded in the type system.

Efficiency. There are no performance measurements available for SafetyNet. However, Wakeman *et al.* hope to get a performance boost from the static type system by using a code caching strategy similar to that of ANTS. By performing the static typecheck of a piece of code when it is first downloaded, the system may safely eliminate dynamic runtime checks when the code is executed. Thus, the cost of the static check is amortized across all of the code's invocations. Other projects have encountered high overheads from a threaded

model of execution [HMA+99, Ale98], so care will have to be given to the thread system implementation.

Flexibility. A large part of the flexibility of the language will be dependent upon the primitives provided for access to node data structures or services. However, constructs are provided for node-resident state, remote thread spawning (equivalent to sending an active packet), and data delivery; Wakeman *et al.* demonstrate how to provide a basic multicast service using the SafetyNet language.

8.8 StreamCode

StreamCode [HEK00, EHH01] is a system being developed at NEC and ETH-Zürich. The StreamCode team has focused on high performance active packets and have designed their language to promote a fast hardware implementation in the form of a StreamCode processor. The idea is to execute a packet program *as it is read in off the network*; straight line code is thus executed at line speed (as well as backward branches to program portions that have already been received), although forward branches stall until the target code has been received.

Safety. The StreamCode approach to resource usage is really quite simple; a packet program may only execute for as much time as it takes the network interface to receive all the bits of the packet. After that point, the hardware processor resets, and the packet program is aborted, thus guaranteeing that a packet cannot use an overly high amount of processing time. However, it is unclear whether node memory usage is bounded (although a program's state is cleared when execution terminates, much as in SNAP). Finally, it appears that network resource bounding is provided by a TTL-like mechanism, but the presence of a native multicast primitive means that a StreamCode packet has the potential for exponential fanout (thus permitting a fairly simple denial of service attack initiated by a single packet, where the packet simply multiplies itself out for several hops, and then all descendents converge on the victim).

As a side note, Egawa *et al.* [EHH01] suggest that end users pad their packets with extra payload to permit the programs to complete their execution. This potentially encourages StreamCode programmers to waste bandwidth by attaching extra padding "just to be sure" their programs will have enough time to finish.

Efficiency. As mentioned above, packet programs may only execute for as long as it takes to receive all the packet bits, so StreamCode executes at line speeds almost by definition. Egawa *et al.* [EHH01] report 829 Mb/s throughput on 933 MHz Pentium III machines connected by Gigabit Ethernet interfaces.

Flexibility. The StreamCode instruction set is modeled on a generic MIPS instruction set with additional primitives for table lookup and burst copies between packet buffers. Unfortunately, the onus is on programmers to be sure that their programs complete in the necessary time so they do not get prematurely terminated. Egawa *et al.* [EHH01] use StreamCode to implement content-sensitive multicast to differentiate between premium-service versus best-effort flows or entertainment data versus commercial advertisements to do "smart" preferential dropping at congestion points.

Chapter 9

Conclusions and Contributions

The primary contribution of this work, is, of course, the demonstration of our thesis that active packet systems can be practical. A further contribution is SNAP itself, as the concrete example that we use to prove our claim. The active networking community has been moving towards practicality for active packets for several years now, and we have finally accomplished it.

The establishment of our practicality framework with our criteria of safety, efficiency, and flexibility, is also a contribution to the community, as it provides a means to evaluate subsequent active packet systems. Our in-depth evaluation in Chapter 8 of existing active packet systems with respect to this framework also provides a useful survey of the technology to date.

Furthermore, our model of SNAP execution, using lightweight "network script" packets with linear resource usage, is important for understanding active packets in a more global, network-wide context. In particular, different active packet schemes and implementations that satisfy the model can be interchanged wherever there is only a dependency on the predictable resource use the model provides. Also, since IP satisfies the model as well, this model serves to unite IP and active packets in one practical family.

In the course of demonstrating that SNAP fits the model and meets our requirements for safety, efficiency, and flexibility, we have also made the following contributions:

Safety. SNAP is the first active packet system to provide adequate resource predictability. Our establishment of local and global predictability properties can also be applied to

subsequent active packet systems. Finally, our safety proofs (although themselves simple) are the first such proofs for an active packet system.

Efficiency. SNAP is the first active packet system to provide IP-like functionality at IP-like overheads in a software router setting, while still maintaining robustness. Our C code base will be easily adaptable to other operating systems kernels, permitting other researchers to deploy SNAP in their own active networks. Also, our techniques for avoiding marshalling costs and permitting in-place execution may be applied to increase the efficiency of other active packet systems.

Flexibility. In the development of our two main applications, we also use SNAP to demonstrate new and interesting algorithms. In our Distributed Denial-of-Service detection application, SNAP provides a lightweight mobile agent platform for network management, where previous mobile agent infrastructures have been notoriously heavyweight. This permits network monitoring algorithms to see actual measurable gains over centralized polling techniques, rather than just theoretical asymptotic complexity advantages.

In addition, the style of monitoring we use in this application shows a novel way to reduce management traffic by avoiding having to query every managed node. This is inherently enabled by the mobile agent approach of embedding domain-specific control logic in the querying packets. Furthermore, because we take an active packet polling approach, there is no need for additional resident monitoring daemons at the managed nodes; the SNAP interpreter will only run when active packets are present.

Finally, our AVid video-on-demand system shows a way to achieve an incrementally-deployable multicast framework through the use of the Router Alert IP option to set up transparent tunnels through legacy IP nodes. In addition, we combine media-specific congestion adaptation in our packets to provide a usable system even in the absence of guaranteed quality of service.

Appendix A

SNAP Instruction Reference

This appendix contains a complete enumeration of the current SNAP instruction set. We will use the notation $s[i]$ to refer to the ith stack value, with $s[0]$ being the top stack value. The program counter will be abbreviated *pc*. Branch offsets will be abbreviated *off*.

A.1 Control Flow Instructions

- **exit**. Terminates the current packet execution.
- **paj** n. "Pop and jump." Computes an offset $off := n + s[0]$. Pops $s[0]$ from the stack, verifies that $off > 0$, and then sets $pc := pc + off$. Thus, **paj 1** with a zero on top of the stack is the same as a **pop**.
- **tpaj** n. "Test, pop, and jump." If $s[0] = 0$, then sets $off := n + s[1]$, else $off := 1$. Pops $s[0]$ and $s[1]$, verifies that $off > 0$, and then sets $pc := pc + off$. If $s[0] \neq 0$, **tpaj** n has the same effect as **popi 2**.
- **ji** n. "Jump immediate." Verifies that $n > 0$, and sets $pc := pc + n$.
- **bez** n. "Branch if equal to zero." If $s[0] = 0$, then sets $off := n$, else $off := 1$. Pops $s[0]$, verifies that $off > 0$, and then sets $pc := pc + off$. **bez 1** has the same effect as **pop**.
- **bne** n. "Branch if not equal to zero." If $s[0] \neq 0$, then sets $off := n$, else $off := 1$. Pops $s[0]$, verifies that $off > 0$, and then sets $pc := pc + off$. **bne 1** has the same effect as **pop**.

A.2 Stack Manipulation Instructions

- **pop**. Removes $s[0]$ from the stack, then increments pc.
- **popi** n. Removes the top n values from the stack, then increments pc.
- **pull** n. Pushes a copy of $s[n]$ on top of the stack (so **pull 0** duplicates the top stack value), then increments pc.
- **store** n. Overwrites $s[n]$ with the value stored at $s[0]$, pops $s[0]$, then increments pc. **store 0** is the same as a **pop**.
- **pint** n. Pushes n onto the stack with an "integer" type tag, then increments pc.
- **paddr** n. Pushes n onto the stack with an "address" type tag, then increments pc.
- **ptup** n. Pushes n onto the stack with a "tuple" type tag, then increments pc.
- **pexc** n. Pushes n onto the stack with an "exception type tag, then increments pc.
- **pstr** n. Pushes n onto the stack with a "string" type tag, then increments pc.
- **pflt** n. Pushes n onto the stack with a "float" type tag, then increments pc.

A.3 Heap Manipulation Instructions

- **mktup** n. Allocates a length-n tuple in the heap, initializes the values with $s[0], \ldots, s[n-1]$, pops $s[0], \ldots, s[n-1]$, and then pushes a tuple value on the stack which is a heap offset indicating the location of the newly allocated tuple. Finally, **mktup** increments pc.
- **nth** n. If $s[0]$ is a tuple value, then **nth** pops $s[0]$, extracts the nth value from the heap-allocated tuple indicated by the heap offset in $s[0]$ and pushes it on the stack. Finally, **nth** increments pc.
- **len**. If $s[0]$ is a tuple value, **len** pops $s[0]$ and pushes the length of the tuple to which it points, then increments pc.

- **istup**. If $s[0]$ is a tuple value, pushes a 1 on top of it, else pushes 0. Then **istup** increments *pc*. Note that unlike the other heap manipulation instructions, **istup** does not pop any values off the stack before pushing its result.

A.4 Relational Operators

For the purposes of the relational operators indicated here, two stack values are equal if they are bit-wise equal (note that this implies that their type tags must be equal as well). For results, all relational operators push a 1 if their predicate is true, 0 if it is false. All relational operator instructions increment *pc* afterwards. Relative order operations such as greater-than or less-than are only valid for integers and floating point numbers.

- **eq**. Pops $s[0]$ and $s[1]$ and pushes $(s[0] = s[1])$.
- **eqint** n. Pops $s[0]$ and compares it to n interpreted as an "integer" constant, pushing 1 if they are equal, 0 otherwise.
- **eqadr** n. Pops $s[0]$ and compares it to n interpreted as an "address" heap offset pointer, pushing 1 if they are equal, 0 otherwise.
- **eqtup** n. Pops $s[0]$ and compares it to n interpreted as a "tuple" heap offset pointer, pushing 1 if they are equal, 0 otherwise.
- **eqexc** n. Pops $s[0]$ and compares it to n interpreted as an "exception" constant, pushing 1 if they are equal, 0 otherwise.
- **eqstr** n. Pops $s[0]$ and compares it to n interpreted as a "string" heap offset pointer, pushing 1 if they are equal, 0 otherwise.
- **eqflt** n. Pops $s[0]$ and compares it to n interpreted as a "float" heap offset pointer, pushing 1 if they are equal, 0 otherwise.
- **neq**. Pops $s[0]$ and $s[1]$ and pushes $(s[0] \neq s[1])$.
- **nqint** n. Pops $s[0]$ and compares it to n interpreted as an "integer" constant, pushing 1 if they are not equal, 0 otherwise.

- **nqadr** n. Pops $s[0]$ and compares it to n interpreted as an "address" heap offset pointer, pushing 1 if they are not equal, 0 otherwise.
- **nqtup** n. Pops $s[0]$ and compares it to n interpreted as a "tuple" heap offset pointer, pushing 1 if they are not equal, 0 otherwise.
- **nqexc** n. Pops $s[0]$ and compares it to n interpreted as an "exception" constant, pushing 1 if they are not equal, 0 otherwise.
- **nqstr** n. Pops $s[0]$ and compares it to n interpreted as a "string" heap offset pointer, pushing 1 if they are not equal, 0 otherwise.
- **nqflt** n. Pops $s[0]$ and compares it to n interpreted as a "float" heap offset pointer, pushing 1 if they are not equal, 0 otherwise.
- **gt**. Pops $s[0]$ and $s[1]$ and pushes $(s[1] > s[0])$.
- **gti** n. Pops $s[0]$ and pushes $(s[0] > n)$, where n is an integer constant.
- **geq**. Pops $s[0]$ and $s[1]$ and pushes $(s[1] \geq s[0])$.
- **geqi** n. Pops $s[0]$ and pushes $(s[0] \geq n)$, where n is an integer constant.
- **lt**. Pops $s[0]$ and $s[1]$ and pushes $(s[1] < s[0])$.
- **lti** n. Pops $s[0]$ and pushes $(s[0] < n)$, where n is an integer constant.
- **leq**. Pops $s[0]$ and $s[1]$ and pushes $(s[1] \leq s[0])$.
- **leqi** n. Pops $s[0]$ and pushes $(s[0] \leq n)$, where n is an integer constant.
- **fgti** n. Pops $s[0]$ and pushes $(s[0] > n)$, where n is a "float" heap offset pointer.
- **fgeqi** n. Pops $s[0]$ and pushes $(s[0] \geq n)$, where n is a "float" heap offset pointer.
- **flti** n. Pops $s[0]$ and pushes $(s[0] < n)$, where n is a "float" heap offset pointer.
- **fleqi** n. Pops $s[0]$ and pushes $(s[0] \leq n)$, where n is a "float" heap offset pointer.

A.5 Arithmetic Operators

We divide our discussion of arithmetic operators by the type of arguments they take. All instructions increment *pc* afterwards.

A.5.1 Integer/Float Operators

1. **add**. Pops $s[0]$ and $s[1]$ and pushes $(s[1] + s[0])$.
2. **addi** n. Pops $s[0]$ and pushes $(s[0] + n)$, where n is an "integer" constant.
3. **faddi** n. Pops $s[0]$ and pushes $(s[0] + n)$, where n is a "float" heap offset pointer.
4. **sub**. Pops $s[0]$ and $s[1]$ and pushes $(s[1] - s[0])$.
5. **subi** n. Pops $s[0]$ and pushes $(s[0] - n)$, where n is an "integer" constant.
6. **fsubi** n. Pops $s[0]$ and pushes $(s[0] - n)$, where n is a "float" heap offset pointer.
7. **mult**. Pops $s[0]$ and $s[1]$ and pushes $(s[1] * s[0])$.
8. **multi** n. Pops $s[0]$ and pushes $(s[0] * n)$, where n is an "integer" constant.
9. **fmuli** n. Pops $s[0]$ and pushes $(s[0] * n)$, where n is a "float" heap offset pointer.
10. **div**. Pops $s[0]$ and $s[1]$ and pushes $(s[1]/s[0])$.
11. **divi** n. Pops $s[0]$ and pushes $(s[0]/n)$, where n is an "integer" constant.
12. **fdivi** n. Pops $s[0]$ and pushes $(s[0]/n)$, where n is a "float" heap offset pointer.

A.5.2 Integer-only Operators

1. **mod**. Pops $s[0]$ and $s[1]$ and pushes $(s[1] \bmod s[0])$.
2. **modi** n. Pops $s[0]$ and pushes $(s[0] \bmod n)$.
3. **neg**. Pops $s[0]$ and pushes $(0 - s[0])$.
4. **not**. Pops $s[0]$, pushes 1 if $(s[0] = 0)$, 0 otherwise.

5. **lnot**. Pops $s[0]$, and pushes ($s[0]$) (bitwise negation).

6. **and**. Pops $s[0]$ and $s[1]$ and pushes ($s[1]$ & $s[0]$) (bitwise and).

7. **andi** n. Pops $s[0]$ and pushes ($s[0]$ & n).

8. **or**. Pops $s[0]$ and $s[1]$ and pushes ($s[1]$ | $s[0]$) (bitwise or).

9. **ori** n. Pops $s[0]$ and pushes ($s[0]|n$).

10. **lshl**. Pops $s[0]$ and $s[1]$ and pushes ($s[1] << s[0]$) (left shift logical).

11. **lshli**. Pops $s[0]$ and pushes ($s[0] << n$).

12. **rshl**. Pops $s[0]$ and $s[1]$ and pushes ($s[1] >> s[0]$) (right shift logical).

13. **rshli**. Pops $s[0]$ and pushes ($s[0] >> n$).

14. **rsha**. Pops $s[0]$ and $s[1]$ and pushes ($s[1] >>> s[0]$) (right shift arithmetic).

15. **rshai**. Pops $s[0]$ and pushes ($s[0] >>> n$).

16. **xor**. Pops $s[0]$ and $s[1]$ and pushes ($s[1]$ xor $s[0]$).

17. **xori** n. Pops $s[0]$ and pushes ($s[0]$ xor n).

A.5.3 Address operators

1. **snet**. Pops $s[0]$ and $s[1]$. Pushes the subnet address computed by using the value pointed to by $s[0]$ as an IPv4 address and the value pointed to by $s[1]$ as a subnet mask.

2. **sneti** n. Pops $s[0]$ and pushes the subnet address computed by using the value pointed to by $s[0]$ as an IPv4 address and the value pointed to by interpreting n as an "address" heap offset pointer as a subnet mask.

3. **bcast** n. Pops $s[0]$ and $s[1]$. Pushes a broadcast address computed by using the value pointed to by $s[0]$ as an IPv4 address and the value pointed to by $s[1]$ as a subnet mask.

4. **bcasti** n. Pops $s[0]$ and pushes the broadcast address computed by using the value pointed to by $s[0]$ as an IPv4 address and the value pointed to by interpreting n as an "address" heap offset pointer as a subnet mask.

A.6 Environment Query Instructions

All environment query instructions also increment *pc* afterwards.

- **isx**. Pushes a 1 onto the stack if $s[0]$ is an exception value, 0 otherwise. Note that $s[0]$ is not popped from the stack.
- **getrb**. Pushes the current packet's resource bound count onto the stack.
- **getsrc**. Pushes a copy of the packet's source address header field onto the stack.
- **getdst**. Pushes a copy of the packet's destination address header field onto the stack.
- **getspt**. Pushes a copy of the packet's source port header field onto the stack.
- **here**. Pushes a copy of one of the current node's addresses onto the stack.
- **ishere**. Pops $s[0]$, and if it is one of the current node's addresses, pushes a 1, else pushes a 0.
- **route**. Pops the address in $s[0]$, and pushes the address of the next hop towards that address.
- **rtdev**. Pops the address in $s[0]$, pushes the address of the next hop towards that address, then pushes the number of the corresponding outgoing interface.

A.7 Networking Instructions

- **send**. Pops $s[0]$, $s[1]$, $s[2]$, and $s[3]$. Sends a new packet with destination address $s[0]$, resource bound $s[1]$, copies of the top $s[3]$ stack values (after popping $s[0-3]$ above), and entry point $s[3]$. If $s[2]$ is -1, then the entire stack is taken (again, after popping $s[0-3]$ above). All other header fields for the new packet are inherited

from the current packet. The current packet's resource bound is decremented by $s[1]$. Finally, *pc* is incremented.

- **hop**. Pops $s[0]$, $s[1]$, $s[2]$, and $s[3]$. Sends a new packet with destination address $s[0]$ (where $s[0]$ is one of the current node's neighbors), resource bound $s[1]$, copies of the top $s[3]$ stack values (after popping $s[0-3]$ above), and entry point $s[3]$. If $s[2]$ is -1, then the entire stack is taken (again, after popping $s[0-3]$ above). All other header fields for the new packet are inherited from the current packet. The current packet's resource bound is decremented by $s[1]$. Finally, *pc* is incremented.

- **forw**. If the packet's destination address is an address of the current node, simply increments *pc*. Else, sends a copy of the current packet towards the destination address and then terminates the current packet's execution. **forw** is equivalent to the following sequence of instructions:

```
    getdst              ; find destination of packet
    ishere              ; are we there?
    bne       nosend-pc ; if so, fall through
    getep               ; same entry point as current packet
    pint      -1        ; take all the current stack
    getrb               ; use all remaining resource bound
    getdst              ; use same destination address
    send                ; send packet
    exit
  nosend:
    ...
```

- **forwto**. Pops $s[0]$. If $s[0]$ is one of the current node's addresses, simply increments *pc*. Else, sends a copy of the current packet but with destination address $s[0]$ and then terminates the current packet's execution.

- **demux**. Pops $s[0]$ and $s[1]$ and delivers the string indicated by the heap offset pointer $s[1]$ to the application port number $s[0]$. Then the current packet terminates.

- **demuxi** n. Pops $s[0]$ and delivers the string indicated by the heap offset pointer $s[0]$ to the application port number n. Then the current packet terminates.

A.8 Debugging Instructions

- **print**. Causes $s[0]$ to be printed to the system log, then advances pc.

A.9 Service Interface Instructions

- **calls** n. Invokes the node-resident service named by the string pointed to by the heap offset pointer n. Further behavior is service-dependent.

Bibliography

[AAH+98] D. Scott Alexander, William A. Arbaugh, Michael Hicks, Pankaj Kakkar, Angelos Keromytis, Jonathan T. Moore, Carl A. Gunter, Scott M. Nettles, and Jonathan M. Smith. The SwitchWare Active Network Architecture. *IEEE Network*, 12(3):29–36, 1998.

[AAKS98] D. Scott Alexander, William A. Arbaugh, Angelos D. Keromytis, and Jonathan M. Smith. A Secure Active Network Environment Architecture: Realization in SwitchWare. *IEEE Network*, 12(3):37–45, 1998.

[Ale98] D. Scott Alexander. *ALIEN: A Generalized Computing Model of Active Networks*. PhD thesis, University of Pennsylvania, 1998.

[AWY96a] C. C. Aggarwal, J. L. Wolf, and P. S. Yu. On Optimal Batching Policies for Video-on-Demand Storage Servers. In *Proceedings of the 1996 International Conference on Multimedia Systems*, pages 253–258, June 1996.

[AWY96b] C. C. Aggarwal, J. L. Wolf, and P. S. Yu. A permutation-based pyramid broadcasting scheme for video-on-demand systems. In *Proceedings of the IEEE International Conference on Multimedia Computing and Systems*, pages 118–126, June 1996.

[BBC+98] S. Blake, D. Black, M. Carlson, E. Davies, Z. Wang, and W. Weiss. An architecture for differentiated services. RFC 2475, IETF, December 1998.

[BBC00] Yahoo brought to standstill. *BBC News*, February 2000.

[BBDS97] Richard Black, Paul Barham, Austin Donnelly, and Neil Stratford. Protocol Implementation in a Vertically Structured Operating System. In *Proceedings*

of the 22nd Annual IEEE Conference on Local Computer Networks (LCN'97), pages 179–188, November 1997.

[BCZ96] Samrat Bhattacharjee, Kenneth L. Calvert, and Ellen W. Zegura. On active networking and congestion. Technical Report GIT-CC-96-02, Georgia Tech, 1996.

[BGP97] M. Baldi, S. Gai, and G. Picco. Exploiting Code Mobility in Decentralized and Flexible Network Management. In *Proceedings of the First International Workshop on Mobile Agents*, pages 13–26, April 1997.

[Blo70] Burton H. Bloom. Space/Time Trade-offs in Hash Coding with Allowable Errors. *Communications of the ACM*, 13(7):422–426, July 1970.

[BM95] Scott Bradner and Allison Mankin. The Recommendation for the IP Next Generation Protocol. RFC 1752, IETF, January 1995.

[BM98] Markus Breugst and Thomas Magedanz. Mobile Agents—Enabling Technology for Active Intelligent Network Implementation. *IEEE Network*, 12(3):53–60, May/June 1998.

[BM99] Yitzhak Birk and Ron Mondri. Tailored transmissions for efficient near-video-on-demand service. In *Proceedings of the IEEE International Conference on Multimedia Computing and Systems*, pages 226–231, June 1999.

[BP98] Mario Baldi and Gian Pietro Picco. Evaluating the Tradeoffs of Mobile Code Design Paradigms in Network Management Applications. In *Proceedings of the 20th International Conference on Software Engineering (ICSE'98)*, pages 146–155, April 1998.

[BW88] Hans-Juergen Boehm and Mark Weiser. Garbage Collection in an Uncooperative Environment. *Software Practice and Experience*, 18(9):807–820, September 1988.

[CDD+99] Sumi Choi, Dan Decasper, John DeHart, Ralph Keller, John Lockwood, Jonathan Turner, and Tilman Wolf. Design of a Flexible Open Platform for

High Performance Active Networks. In *Proceedings of the 37th Annual Allerton Conference on Communication, Control, and Computing*, pages 157–165, September 1999.

[CER96] Denial-of-Service Attack via ping. CERT Advisory CA-1996-26, December 1996.

[CER99] Denial of Service Tools. CERT Advisory CA-1999-17, December 1999.

[CER00] "Mstream" Distributed Denial of Service Tool. CERT Incident Note IN-2000-05, May 2000.

[CER01] "Carko" Distributed Denial-of-Service Tool. CERT Incident Note IN-2001-04, April 2001.

[CFSD90] Jeffrey D. Case, Mark Fedor, Martin Lee Schoffstall, and James R. Davin. A Simple Network Management Protocol (SNMP). RFC 1157, IETF, May 1990.

[Cor01] Intel Corporation. Intel® IXP1200 Network Processor Datasheet, May 2001.

[CS01] Ariana Eunjung Cha and David Streitfield. Microsoft Web Sites Attacked. *Washington Post*, January 2001.

[Cur98] John Curtis. In Defense of Jumbo Frames. *Network World*, August 1998.

[CW00] Karl Crary and Stephanie Weirich. Resource Bound Certification. In *Proceedings of the 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'00)*, pages 184–198, January 2000.

[DH98] Stephen E. Deering and Robert M. Hinden. Internet Protocol, Version 6 (IPv6) Specification. RFC 2460, IETF, December 1998.

[DHS96] A. Dan, Y. Heights, and D. Sitaram. Generalized Interval Caching Policy for Mixed Interactive and Long Video Workloads. In *Proceedings of SPIE's Conference on Multimedia Computing and Networking*, pages 344–351, January 1996.

[DSS94] Asit Dan, Dinkar Sitaram, and Perwez Shahabuddin. Scheduling Policies for an On-Demand Video Server with Batching. In *Proceedings of the ACM Multimedia Conference*, pages 15–23, October 1994.

[DSS96] Asit Dan, Dinkar Sitaram, and Perwez Shahabuddin. Dynamic batching policies for an on-demand video server. *Multimedia Systems*, 4(3):51–58, June 1996.

[ECG+02] Walter Eaves, Lawrence Cheng, Alex Galis, Thomas Becker, Toshiaki Suzuki, Spyros Denazis, and Chiho Kitahara. SNAP Based Resource Control for Active Networks. In *Proceedings of IEEE GLOBECOM 2002*, November 2002. To appear.

[EFV99] Derek L. Eager, Michael C. Ferris, and Mary K. Vernon. Optimized Regional Caching for On-Demand Data Delivery. In *Proceedings of Multimedia Computing and Networking (MMCN'99)*, pages 301–316, January 1999.

[EHH01] Takashi Egawa, Koji Hino, and Yohei Hasegawa. Fast and secure packet processing environment for per-packet QoS customization. In *Proceedings of the IFIP-TC6 Third International Working Conference on Active Networks (IWAN'01)*, September/October 2001.

[EV98] D. L. Eager and M. K. Vernon. Dynamic skyscraper broadcasts for video-on-demand. In *Proceedings of the 4th International Workshop on Multimedia Information Systems (MIS'98)*, September 1998.

[Fen97] William C. Fenner. Internet Group Management Protocol, Version 2. RFC 2236, IETF, November 1997.

[FGM+99] Roy T. Fielding, James Gettys, Jeffrey C. Mogul, Henrik Frystyk Nielsen, Larry Masinter, Paul J. Leach, and Tim Berners-Lee. Hypertext Transfer Protocol—HTTP/1.1. RFC 2616, IETF, June 1999.

[FJ93] Sally Floyd and Van Jacobson. Random early detection gateways for congestion avoidance. *IEEE/ACM Transactions on Networking*, 1(4):397–413, August 1993.

[For00] Net vs. Norm: The Slashdot Effect. *Forbes Magazine*, February 2000.

[GJS96] James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification*. Addison Wesley, 1996.

[GLM95] Leana Golubchik, John C.-S. Lui, and Richard R. Muntz. Reducing I/O Demand in Video-on-Demand Storage Servers. In *Proceedings of ACM SIGMETRICS*, pages 25–36, 1995.

[GT99] Lixin Gao and Donald F. Towsley. Supplying Instantaneous Video-on-Demand Services using Controlled Multicast. In *Proceedings of the IEEE International Conference on Multimedia Computing and Systems*, pages 117–121, June 1999.

[GVK+95] Jim Gemmell, Harrick M. Vin, Dilip D. Kandlur, P. Venkat Rangan, and Lawrence A. Rowe. Multimedia Storage Servers: A Tutorial. *IEEE Computer*, 28(5):40–49, 1995.

[HB99] Gísli Hjálmtýsson and Samrat Bhattacharjee. Control on Demand. In *Proceedings of the First International Working Conference on Active Networks (IWAN'99)*, pages 315–329, June/July 1999.

[HCC+98] Chris Hawblitzel, Chi-Chao Chang, Grzegorz Czajkowski, Deyu Hu, and Thorsten von Eicken. Implementing Multiple Protection Domains in Java. In *Proceedings of the 1998 USENIX Annual Technical Conference*, June 1998.

[Hed88] C. Hedrick. Routing Information Protocol. RFC 1058, IETF, June 1988.

[HEK00] Koji Hino, Takashi Egawa, and Yoshiaki Kiriha. Open Programmable Layer-3 Networking. In *Proceedings of the Sixth IFIP Conference on Intelligence in Networks (SmartNet 2000)*, September 2000.

[Hic01] Michael Hicks. *Dynamic Software Updating.* PhD thesis, University of Pennsylvania, 2001.

[HKM+98a] Michael Hicks, Pankaj Kakkar, Jonathan T. Moore, Carl A. Gunter, and Scott Nettles. Network Programming with PLAN. In *Proceedings of the IEEE Workshop on Internet Programming Languages*, pages 127–143, May 1998.

[HKM+98b] Michael Hicks, Pankaj Kakkar, Jonathan T. Moore, Carl A. Gunter, and Scott Nettles. PLAN: A Packet Language for Active Networks. In *Proceedings of the 1998 International Conference on Functional Programming (ICFP'98)*, pages 86–93, September 1998.

[HKS02] Michael Hicks, Angelos D. Keromytis, and Jonathan M. Smith. A Secure PLAN (Extended Version). In *DARPA Active Networks Conference and Exposition (DANCE'02)*, pages 224–237, May 2002.

[HMA+99] Michael Hicks, Jonathan T. Moore, D. Scott Alexander, Carl Gunter, and Scott Nettles. PLANet: An Active Internetwork. In *Proceedings of the 18th Annual Joint Conference of the IEEE Computer and Communications Societies*, pages 1124–1133, March 1999.

[HMN01] Michael Hicks, Jonathan T. Moore, and Scott Nettles. Compiling PLAN to SNAP. In *Proceedings of the IFIP-TC6 Third International Working Conference on Active Networks (IWAN'01)*, pages 134–151, September/October 2001.

[HMWN02] Michael Hicks, Jonathan T. Moore, David Wetherall, and Scott Nettles. Experiences with capsule-based active networking. In *Proceedings of the 2002 DARPA Active Networks Conference and Exposition (DANCE'02)*, pages 16–24, May 2002.

[Hof99] Martin Hoffman. Linear types and non-size-increasing polynomial time computation. In *Proceedings of the 14th IEEE Symposium on Logic in Computer Science (LICS'99)*, July 1999.

[Hor00] Luke Hornof. Self-Specializing Mobile Code for Adaptive Network Services. In *Proceedings of the 2nd International Working Conference on Active Networks (IWAN'00)*, pages 102–113, October 2000.

[HP99] John Hughes and Lars Pareto. Recursion and Dynamic Data-structures in Bounded Space: Towards Embedded ML Programming. In *Proceedings of the 1999 International Conference on Functional Programming (ICFP'99)*, pages 70–81, September 1999.

[HPS96] John Hughes, Lars Pareto, and Amr Sabry. Proving the Correctness of Reactive Systems Using Sized Types. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'96)*, pages 410–423, January 1996.

[HS97] Kien A. Hua and Simon Sheu. Skyscraper broadcasting: A new broadcasting scheme for metropolitan video-on-demand systems. In *Proceedings of the ACM SIGCOMM'97 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, pages 89–100, September 1997.

[IBM02] IBM PowerNP NP4GS3 Network Processor Datasheet. `http://www.ibm.com/chips/techlib/techlib.nsf/prodcts/IBM_PowerNP_NP4GS3`, February 2002.

[Jac90] Van Jacobson. Compressing TCP/IP Headers for Low-Speed Serial Links. RFC 1144, IETF, February 1990.

[Jai91] Raj Jain. *The Art of Computer Systems Performance Analysis.* Wiley, 1991.

[Kat97] Dave Katz. IP Router Alert Option. RFC 2113, IETF, February 1997.

[KPS02] Andreas Kind, Roman Pletka, and Burkhard Stiller. The potential of just-in-time compilation in active networks based on network processors. In *Proceedings of the 5th Workshop on Open Architectures and Network Programming (OPENARCH'02)*, pages 79–90, June 2002.

[Law01] George Lawton. Is IPv6 Finally Gaining Ground? *IEEE Computer*, 34(8):11–15, August 2001.

[Lem01] Robert Lemos. Hackers storm White House Web site. *ZDNet News*, May 2001.

[LG98] Ulana Legedza and John V. Guttag. Using Network-level Support to Improve Cache Routing. *Computer Networks*, 30(22–23):2193–2201, November 1998.

[LGT98] Li-wei H. Lehman, Stephen J. Garland, and David L. Tennenhouse. Active Reliable Multicast. In *Proceedings of the 17th Annual Joint Conference of the IEEE Computer and Communications Societies*, pages 581–589, April 1998.

[LR98] Xavier Leroy and François Rouaix. Security Properties of Typed Applets. In *Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'98)*, pages 391–403, January 1998.

[LRW99] Xavier Leroy, Didier Remy, and Pierre Weis. Objective Caml—a General Purpose High-level Programming Language. *ERCIM News*, (36), January 1999.

[LWG98] Ulana Legedza, David Wetherall, and John Guttag. Improving the Performance of Distributed Applications Using Active Networks. In *Proceedings of the 17th Annual Joint Conference of the IEEE Computer and Communications Societies*, April 1998.

[LY96] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. Addison Wesley, 1996.

[MCG+99] Greg Morrisett, Karl Crary, Neal Glew, Dan Grossman, Richard Samuels, Frederick Smith, David Walker, Stephanie Weirich, and Steve Zdancewic. TALx86: A realistic typed assembly language. In *Proceedings of the Second Workshop on Compiler Support for System Software (WCSSS'99)*, May 1999.

[Men99] Paul Menage. RCANE: A Resource Controlled Framework for Active Network Services. In *Proceedings of the First International Working Conference on Active Networks (IWAN'99)*, pages 25–36, June/July 1999.

[MHN99] Jonathan T. Moore, Michael Hicks, and Scott M. Nettles. Chunks in PLAN: Language support for programs as packets. In *Proceedings of the 37th Annual Allerton Conference on Communication, Control, and Computing*, September 1999.

[MK00] Keith McCloghrie and Frank Kastenholz. The Interfaces Group MIB. RFC 2863, IETF, June 2000.

[ML70] Charles H. Moore and Geoffrey C. Leach. FORTH—A Language for Interactive Computing. Technical report, Mohasco Industries, Inc., 1970.

[MTH90] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML*. MIT Press, 1990.

[Muh98] Murhimanya Muhugusa. Implementing Distributed Services with Mobile Code: The Case of the Messenger Environment. In *Proceedings of the IASTED International Conference on Parallel and Distributed Systems (Euro-PDS'98)*, July 1998.

[MWCG99] Greg Morrisett, David Walker, Karl Crary, and Neal Glew. From System F to Typed Assembly Language. *ACM Transactions on Programming Languages and Systems*, 21(3):527–568, May 1999.

[Nec97] George C. Necula. Proof-Carrying Code. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'97)*, pages 106–119, January 1997.

[NGK99] Erik L. Nygren, Stephen J. Garland, and M. Frans Kaashoek. PAN: A High-Performance Active Network Node Supporting Multiple Mobile Code Systems. In *Proceedings of the 2nd Workshop on Open Architectures and Network Programming (OPENARCH'99)*, pages 78–89, March 1999.

[NL97] George C. Necula and Peter Lee. Safe, Untrusted Agents using Proof-Carrying Code. In *Special Issue on Mobile Agent Security*, volume 1419 of *Lecture Notes in Computer Science*. Springer, October 1997.

[Nyg98] Erik L. Nygren. The Design and Implementation of a High Performance Active Network Node. Master's thesis, Massachusetts Institute of Technology, February 1998.

[Plu82] David C. Plummer. An Ethernet Address Resolution Protocol. RFC 826, IETF, 1982.

[Pos80] Jonathan B. Postel. User Datagram Protocol. RFC 768, IETF, August 1980.

[Pos81a] Jonathan B. Postel. Internet Control Message Protocol. RFC 792, IETF, September 1981.

[Pos81b] Jonathan B. Postel. Internet Protocol. RFC 791, IETF, September 1981.

[Pos82] Jonathan B. Postel. Simple Mail Transfer Protocol. RFC 821, IETF, August 1982.

[PR85] J. Postel and J. Reynolds. File Transfer Protocol (FTP). RFC 959, IETF, October 1985.

[PR98] Ian Piumarta and Fabio Riccardi. Optimizing Direct-threaded Code by Selective Inlining. In *Proceedings of the 1998 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'98)*, pages 291–300, June 1998.

[PT00] Antonio Puliafito and Orazio Tomarchio. Using Mobile Agents to Implement Flexible Network Management Strategies. *Computer Communications Journal*, 23(8):708–719, April 2000.

[RD99] Marcelo G. Rubinstein and Otto Carlos M. B. Duarte. Evaluating Tradeoffs of Mobile Agents in Network Management. *Networking and Information Systesm Journal*, 2(2):237–252, 1999.

[RD01] Danny Raz and Mark Dilman. Efficient Reactive Monitoring. In *Proceedings of the 20th Annual Joint Conference of the IEEE Computer and Communications Societies*, pages 1012–1019, April 2001.

[RG94] Brian Reistad and David K. Gifford. Static Dependent Costs for Estimating Execution Time. In *Proceedings of the 1994 ACM Conference on Lisp and Functional Programming (LFP'94)*, pages 65–78, June 1994.

[RLLS98] Raj Rajkumar, Chen Lee, John Lehoczky, and Dan Siewiorek. Practical Solutions for QoS-based Resource Allocation Problems. In *Proceedings of the IEEE Real-Time Systems Symposium (RTSS'98)*, pages 296–306, December 1998.

[RRG01] Sridhar Ramesh, Injong Rhee, and Katherine Guo. Multicast with Cache (Mcache): An Adaptive Zero-Delay Video-on-Demand Service. In *Proceedings of the 20th Annual Joint Conference of the IEEE Computer and Communications Societies*, pages 85–94, April 2001.

[RRV93] Srinivas Ramanathan, P. Venkat Rangan, and Harrick M. Vin. Frame-Induced Packet Discarding: An Efficient Strategy for Video Networking. In *Proceedings of the 4th International Workshop on Network and Operating System Support for Digital Audio and Video (NOSSDAV'93)*, pages 173–184, 1993.

[SBP98a] C. Schramm, A. Bieszczad, and B. Pagurek. Application-Oriented Network Modeling with Mobile Agents. In *Proceedings of the IEEE/IFIP Network Operations and Management Symposium (NOMS'98)*, February 1998.

[SBP98b] G. Susilo, A. Bieszczad, and B. Pagurek. Infrastructure for Advanced Network Management based on Mobile Code. In *Proceedings of the IEEE/IFIP Network Operations and Management Symposium (NOMS'98)*, February 1998.

[SJS+99] Beverly Schwartz, Alden W. Jackson, W. Timothy Strayer, Wenyi Zhou, R. Dennis Rockwell, and Craig Partridge. Smart Packets for Active Networks. In *Proceedings of the 2nd Workshop on Open Architectures and Network Programming (OPENARCH'99)*, pages 90–97, March 1999.

[SJS+00] Beverly Schwartz, Alden W. Jackson, W. Timothy Strayer, Wenyi Zhou, R. Dennis Rockwell, and Craig Partridge. Smart packets: Applying active networks to network management. *ACM Transactions on Computer Systems*, 18(1):67–88, February 2000.

[SMB97] Akhil Sahai, Christine Morin, and S. Billiart. Intelligent agents for a Mobile Network Manager. In *Proceedings of the IFIP/IEEE International Conference on Intelligent Networks and Intelligence in Networks (2IN'97)*, September 1997.

[SN02] Pisai Setthawong and Scott Nettles. Resource Use Tracking for Active Networks. Submitted to the 2002 International Working Conference on Active Networking (IWAN'02), 2002.

[SP01] Dawn Song and Adrian Perrig. Advanced and Authenticated Marking Schemes for IP Traceback. In *Proceedings of the 20th Annual Joint Conference of the IEEE Computer and Communications Societies*, pages 878–886, April 2001.

[SPI00] Hackers disrupt Web sites. *Seattle Post-Intelligencer*, February 2000.

[SSS97] Chris Speirs, Zoltan Somogyi, and Harald Soendergaard. Termination Analysis for Mercury. In *Proceedings of the Fourth Static Analysis Symposium*, pages 160–171, September 1997.

[SSZ98] Ion Stoica, Scott Shenker, and Hui Zhang. Core-Stateless Fair Queueing: Achieving Approximately Fair Bandwidth Allocations in High Speed Networks. In *Proceedings of the ACM SIGCOMM'98 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, pages 118–130, August/September 1998.

[SWKA00] Stefan Savage, David Wetherall, Anna Karlin, and Tom Anderson. Practical Support for IP Traceback. In *Proceedings of the ACM SIGCOMM 2000 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, pages 295–306, August/September 2000.

[TMT01] Seiichiro Tani, Toshiaki Miyazaki, and Noriyuki Takahashi. Adaptive Stream Multicast Based on IP Unicast and Dynamic Commercial Attachment Mechanism: An Active Network Implementation. In *Proceedings of the IFIP-TC6 Third International Working Conference on Active Networks (IWAN'01)*, pages 116–133, September/October 2001.

[Tsc97] Christian F. Tschudin. The Messenger Environment M0—a Condensed Description. In J. Vitek and C. Tschudin, editors, *Mobile Object Systems—Towards the Programmable Internet*, number 1222 in LNCS. Springer, April 1997.

[Tsc99a] Christian Tschudin. A Self-Deploying Election Service for Active Networks. In *Proceedings of the 3rd International Conference on Coordination Models and Languages (COORDINATION'99)*, pages 183–195, April 1999.

[Tsc99b] Christian Tschudin. An Active Networks Overlay Network (ANON). In *Proceedings of the First International Working Conference on Active Networks (IWAN'99)*, pages 156–163, June/July 1999.

[TW96] David L. Tennenhouse and David J. Wetherall. Towards an Active Network Architecture. *Computer Communication Review*, 26(2), April 1996.

[VI96] S. Viswanathan and Tomasz Imielinski. Metropolitan Area Video-on-Demand Service using Pyramid Broadcasting. *Multimedia Systems*, 4(4):197–208, August 1996.

[Wet99] David Wetherall. Active Network Vision and Reality: Lessons from a Capsule-based System. *Operating Systems Review*, 34(5):64–79, December 1999.

[WGT98] David J. Wetherall, John Guttag, and David L. Tennenhouse. ANTS: A Toolkit for Building and Dynamically Deploying Network Protocols. In *Proceedings of the First Workshop on Open Architectures and Network Programming (OPENARCH'98)*, March 1998.

[Wil92] Paul R. Wilson. Uniprocessor Garbage Collection Techniques. In *Proceedings of the 1992 International Workshop on Memory Management (IWMM'92)*, number 637 in Lecture Notes in Computer Science, pages 1–42. Springer-Verlag, September 1992.

[WJOP00] Ian Wakeman, Alan Jeffrey, Tim Owen, and Damyan Pepper. SafetyNet: A Language-Based Approach to Programmable Networks. In *Proceedings of the 3nd Workshop on Open Architectures and Network Programming (OPENARCH'00)*, March 2000.

[WLAG93] Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham. Efficient Software-Based Fault Isolation. In *Proceedings of the 14th ACM Symposium on Operating Systems Principles (SOSP'93)*, pages 203–216, December 1993.

[WW02] Andrew Whitaker and David Wetherall. Forwarding Without Loops in Icarus. In *Proceedings of the 5th Workshop on Open Architectures and Network Programming (OPENARCH'02)*, pages 63–75, June 2002.

[Yas01] Rutrell Yasin. DoS Attack Storms Weather Channel's Routers. *InternetWeek*, May 2001.