Practical Programmable Packets

Jonathan T. Moore* Michael Hicks* Scott Nettles†

*Computer and Information Science
University of Pennsylvania

†Electrical and Computer Engineering
The University of Texas at Austin

Abstract— **We present SNAP (Safe and Nimble Active Packets), a new scheme for programmable (or *active*) packets centered around a new low-level packet language. Unlike previous active packet approaches, SNAP is *practical*: namely, adding significant *flexibility* over IP without compromising *safety and security* or *efficiency*. In this paper we show how to compile from the well-known active packet language PLAN [7] to SNAP, showing that SNAP retains PLAN's flexibility; give proof sketches of its novel approach to resource control; and present experimental data showing SNAP attains performance very close to that of a software IP router.**

Keywords—**Active networks, active packets, capsules, resource control.**

I. Introduction

The explosive growth of the Internet has placed new and increased demands on the network infrastructure. Applications now have varied service requirements such as high bandwidth, low delay, low jitter, etc. The one-size-fits-all, single-service model of IP [19], which has certainly contributed to its success, often no longer fits user or application needs.

To meet these new demands, the network infrastructure must evolve to encompass new service models and protocols. Unfortunately, IP is difficult to change, due to its centralized, committee-controlled nature (*cf.* IPv6 [3]). *Active networks* seek to avoid the committee bottleneck by making the network programmable and thus easier to change on the fly. Perhaps the most radical way to make the network programmable is to make packets carry (or be) programs. In such *active packets* (*a.k.a. capsules* [25]), the traditional packet header is replaced with a program that is executed to determine its handling and effect.

Previous research efforts on active packets, such as PLAN [7] and ANTS [25] have demonstrated that the flexibility provided by active packets can be used to improve application performance and functionality. Examples of active packet applications include application-specific routing [9], transparent redirection of web requests to nearby caches [12], distributed on-line auctions [13], reliable multicast [14], mobile code firewalls [8], and reduced network management traffic [21]. Unfortunately, most of the active packet platforms have either restricted themselves to the control plane [21], had unacceptably low performance [25], [9], or have achieved reasonable performance only by sacrificing safety and security [17]. None has done an effective job of providing safe resource control. This has led to a widespread belief that active network processing in the data plane is fundamentally impractical.

In this paper we show that, to the contrary, active packets can be of general use. To support this claim, we drew from our own extensive experience with PLAN [7], and from the successes and failures of first-generation active packet languages in general, to design SNAP (Safe and Nimble Active Packets). The design of SNAP focuses on three dimensions of the active packet design space: *safety and security*, where existing systems have made good progress on preserving node integrity and providing protection but fail to cope adequately with resource allocation; *flexibility*, where existing systems excel; and *performance*, where existing systems have key limitations, especially for data transport. By demonstrating that SNAP adds significant flexibility over IP without sacrificing either safety or efficiency, we establish SNAP as the first *practical* active packet system.

This work was supported by DARPA under Contract #N66001-06-C-852 and by the NSF under Contracts ANI 00-82386 and ANI 98-13875.

II. Our Approach

Before proceeding with the body of the paper, which consists of SNAP's design, implementation, and performance characteristics, we examine more closely the design-space axes of *safety and security*, *flexibility*, and *efficiency*, and explain how SNAP's approach improves on prior approaches.

A. Safety and Security

Most active packet systems [7], [21], [25] have provided good protection against packets damaging nodes or other packets. The general strategy is to use type-checking and/or dynamic monitoring. With minor variations discussed in Section VI, SNAP uses the same techniques with the same benefits.

Where existing systems generally fail is in controlling the resource utilization of active packets. Several systems use time-to-live (TTL) counters to limit packet proliferation [7], [17], [25] and watchdog timers and allocation limits to terminate packets using too many local resources [21], [25]. These approaches sacrifice safety because forced termination can be unsafe [5]. PLAN [7] limits packet execution by restricting its expressibility so that all programs must terminate. However, bounding this termination time is problematic as packets may run into time exponential to their length. In general, we would prefer that packet resource usage be predictable, so that the network can decide how to process packets most effectively.

SNAP takes a rather dramatic approach to these problems. The language is designed with limited expressibility so that a SNAP program uses bandwidth, CPU, and memory resources in linear proportion to the packet's length. Furthermore, the constant of proportionality is known and small. This means that a node can trivially predict a strict upper bound on the resources that will be used by a packet. Furthermore SNAP also uses a TTL-like mechanism drawn from PLAN called the *resource bound*. Taken together, these two mechanisms place a bound on resource utilization that can be computed when the packet first enters the network. This strict resource control is SNAP's most novel advance.

How is linear resource use achieved? Bandwidth use is inher-

ently linear in packet length, but linear CPU and memory use is achieved by restricting the bytecodes of the SNAP language. Only forward branches are allowed, so that each bytecode is executed at most once. Also, bytecodes (with a few key exceptions, see Section III) must execute in constant time. These two restrictions imply that execution time is linear in the number of bytecodes. Bounding memory works similarly: each instruction can push at most one item on the stack and add at most one element to the heap. Since a linear number of bytecodes are executed, only a linear amount of additional space may be used.

B. Flexibility

The principal feature active networks add to existing networks is flexibility. Existing active packet systems are perhaps the best example of this, allowing custom protocols, flows, and packets. These systems clearly meet the initial flexibility goals of active networking, especially when coupled with extensible routers, as in PLANet [9]. However, even these systems somewhat compromise flexibility by limiting expressibility to aid in achieving safety; for example, with PLAN we designed the language to ensure that all packet programs terminate. Experience has shown this to be a good compromise.

With SNAP, we take this compromise further, restricting flexibility so as to achieve even greater safety: namely, linear per-packet bounds on resource usage. The question is "Have we gone too far?" We argue in Section V that the answer is "No" by presenting a compiler that translates PLAN to SNAP. Although it cannot (and should not, for safety reasons) effectively translate all PLAN programs to SNAP, it does translate a large useful subset. Therefore, since PLAN is one of the more flexible existing systems, SNAP also achieves high flexibility.

C. Efficiency

With one exception, existing systems achieve only mediocre performance, being unable to saturate a 100 Mb/s Ethernet using relatively fast CPUs. The exception, PAN [17], achieves its performance in two key ways: first, it is implemented in-kernel, thus avoiding the overheads imposed by user-space implementations; second, its packet language is unrestricted x86 machine code—thus abandoning any guarantees of node safety. We do not believe that high performance without security is an acceptable design alternative.

SNAP is designed for good performance; it provides basic operators and control flow, thus enabling a lightweight interpreter. SNAP's linear safety properties make it possible to compute a maximum size for a packet buffer, thus avoiding high memory management costs during execution. Furthermore, a compact wire representation improves throughput by leaving more room for payload data. This wire representation makes possible "in-place" execution of packet programs, in many cases avoiding expensive marshalling and unmarshalling. Finally, our in-kernel implementation avoids unnecessary domain crossings.

In this paper, we compare SNAP's performance relative to IP implemented in a similar environment. Our experiments show that active processing imposes negligible overhead above normal IP processing. In other words, with SNAP a user only has to pay for the amount of "activeness" used. We feel that such a result is important because active packets will only be viewed as practical if they impose only a small penalty or no penalty at all for IP-like service. In addition, we show that SNAP is considerably faster than its predecessor system, PLAN.

In the remainder of the paper, we expand on the themes introduced in this section by presenting the details of the SNAP language and its implementation. We begin with an overview of the SNAP bytecode language, and in the following four sections we demonstrate its practicality: in Section IV, we sketch network *safety* proofs which are made possible by SNAP's design; in Section V, we show that SNAP achieves the *flexibility* of previous active packet systems by describing a PLAN-to-SNAP compiler; and in Sections VI and VII, we show that SNAP can have an *efficient* implementation compared to a software IP router. Before concluding, we discuss previous active packet systems in Section VIII.

III. SNAP

We now present an overview of SNAP, introducing the features that will be important through the rest of the paper. SNAP was designed as the successor to our first packet language, PLAN, and shares much of PLAN's design philosophy; we make comparisons between the two throughout the paper. As with PLAN, SNAP has been specifically designed to permit formal proofs, particularly about the safety of its programs. Readers seeking a detailed formal description of the language are directed to [15].

SNAP executes on a stack-based bytecode virtual machine, much like other such VMs except that it supports SNAP's communication primitives and safety approach. A SNAP program consists of a sequence of bytecode instructions, a stack, and a heap. Stack values are "small" data, like integers or addresses, while heap values are "large" data, such as byte arrays or tuples[1], which can be pointed to from small values. Each SNAP instruction consists of an opcode and an optional immediate argument.

SNAP's network semantics are that a packet program is executed by every SNAP-enabled node and simply forwarded by legacy IP routers. We describe in Section VI how this is accomplished. In addition, each SNAP packet has a *resource bound* field similar to the IPv4 time-to-live (TTL) field: this field is decremented at each hop, and a packet must donate some of its resource bound to any child packets that are sent.

SNAP instructions fall into seven classes, listed in Table I. In general, SNAP instructions:

1. execute in constant time, and
2. allocate a constant amount of stack and heap space.

These points are crucial to our safety claims, as discussed in Section IV. As discussed below, a few exceptions to these points exist.

A. Example: Ping

We illustrate SNAP's basic features concretely by using a version of *ping* coded in SNAP, shown in Figure 1. We use *ping* because it allows for a simple example (only 7 SNAP instructions)

[1] A *tuple* can be thought of as an array of small values.

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

TABLE I
SNAP INSTRUCTION CLASSES

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

Fig. 1. SNAP code for ping.

with which the reader should be familiar. Furthermore, this is the program we use for our benchmarks in Section VII.

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 sender's top s stack values; an entry point n, the index of the first instruction to execute; r resource bound; and a destination field of d. Executing **send** results in r resource bound being given to the child packet and thus deducted from the packet that executes the send. **Forw** is just shorthand for a **send** keeping 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. Network control operations are exceptional in that they operate in time linearly related to the packet's length.

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 draconian restriction, we will see in Section V that special compilation techniques still allow useful programs.

The next bytecode executed is **push**, 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**, **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 effectively sent with its source and destination fields swapped, 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: $[port :: payload]$.

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

B. Other Instructions

The instruction classes from Table I not appearing in the ping program are "simple computation," "heap manipulation," and "service access." The "simple computation" instructions 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, relational operators, as well as some special-purpose instructions on addresses (*e.g.*, subnet masks).

The "tuple 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. 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 upload new service routines at runtime to add new functionality. This is essentially the same model of services supported in PLAN [7].

IV. Safety

Safety and security are important issues in a shared internetworking infrastructure—it should not be possible for users, either maliciously or accidentally, to crash the network or otherwise make it unusable to others. In particular, the following properties should hold of *any* packet scheme, let alone an active one:

- **Node integrity**: It should be impossible for the processing of a packet to result in a node crash or subversion.
- **Packet isolation**: It should be impossible for an active packet to affect other packets without permission.
- **Resource safety**: It should be possible to predict and strictly bound the amount of local or network resources consumed by an active packet.

At a high-level, SNAP follows the basic design philosophy of PLAN: namely, that with a limited-expressibility domain-specific language, we can know that a program is safe to execute *without even examining it*. We accomplish this by language design, safe interpretation techniques, and formal proof.

To ensure node integrity and packet isolation, our first line-of-defense is that SNAP simply does not contain any primitives to exert control over the local node or other packets; SNAP programs may query the node for information but may otherwise only affect themselves. We complete our guarantee by employing a form of dynamically-enforced software fault isolation [22]. This essentially prevents an attacker from using an ill-formed packet (*e.g.* by exploiting buffer overruns) to gain control of the node itself or to tamper with another user's packets. For example, pointer dereferences are verified to be within the packet's heap. This approach is much like that of other active packet systems.

A. Resource safety

SNAP's novel contribution is how it achieves resource safety. The limited expressibility of SNAP makes it possible to compute *a priori* bounds on the running time and memory usage of a program, giving significant leverage over controlling resources.

Consider the following resource safety requirements:

1. **CPU safety**: on any one node, processing a packet p should take $O(|p|)$ time, where $|p|$ is the length of p.
2. **Memory safety**: on any one node, processing a packet p should require $O(|p|)$ memory.
3. **Bandwidth safety**: the overall network bandwidth consumed by a packet p should be $O(n|p|)$ where n is some resource bound associated with p at its creation.

These requirements were derived from properties of unicast IPv4 packets: examining the header (including any options) and forwarding the packet take $O(|p|)$ time and space. Similarly, the amount of network bandwidth consumed by the packet is bounded by the product of the size of the packet and the TTL contained in its header[2].

SNAP has these properties and it is possible to prove so formally. Here, we outline the basic ideas of such proofs; the reader interested in the formal details is referred to the technical report [15].

A.1 CPU safety

Because all branches in SNAP go forward, each instruction in the program is executed at most once. Furthermore, with the exception of the network control instructions, all SNAP instructions execute in constant time. Thus, aside from the network control instructions, a SNAP program runs in time linear in its length (and therefore in the length of the containing packet).

However, sending a packet takes $O(|p|)$ time, as does delivering data with **demux**. In the pathological case of a program consisting only of **send** instructions, the total time would be $|p| \times O(|p|) = O(|p^2|)$. To gain the desired linear bound, we restrict the number of network operations allowed per packet to some constant n, resulting in $n \times O(|p|) = O(|p|)$. The most conservative case would be to set $n = 1$, allowing only a single send or delivery per packet, matching unicast semantics but prohibiting multicast-style programs, and even reasonable implementations of traceroute [6], [20]. A more flexible bound, used in our current implementation, is the least n such that multicast may be programmed: n varies per node to be the number of network interfaces on that node. For **demux**, we permit only one delivery per packet, and force the packet to exit following the delivery. As we gain more experience with SNAP, we expect to develop more insight into reasonable policies.

Note that if we know the most expensive constant-time instruction, the maximum number of sends, and the maximum packet size, we can precisely compute an upper bound on a program's runtime.

A.2 Memory safety

To prove memory safety, we show that for each instruction (using amortized analysis) at most one small value may be allocated, on each of the heap and the stack. For all of the SNAP instructions except the **mktup** instruction, zero or more arguments are consumed from the stack, and at most one result value is added, while the heap is not affected. Furthermore, the requirement that **mktup** *n* must be followed by at least $n-1$ instructions that are not **mktup** means that each instruction allocates at most one amortized small value on the heap.

Given that each instruction of a SNAP program executes at most once, we see that the maximum number of small values allocated by a program is twice its length. As a result, the SNAP VM need only allocate a buffer of constant size for each packet; in our current implementation this is $4 \times MTU$ due to additional per-heap-object overheads. This buffer can be immediately recycled upon program termination, thus avoiding memory-management overheads, such as garbage collection costs, which have plagued previous systems [9], [23], [17].

[2]Multicast IPv4 packets do not comply with this standard, but as multicast resource usage is an open problem, our "bandwidth safety" property is a conservative goal.

A.3 Bandwidth safety

Finally, we can prove bandwidth safety by observing that a SNAP packet's resource bound is decremented upon reception, and that any child packets must be given some of their parent's resource bound (*i.e.*, conservation of resource bound). Thus a packet p with initial resource bound of n can cause at most n transmissions of $|p|$, whether directly or through its offspring.

B. Services and safety

The above safety guarantees do not necessarily apply if a SNAP program invokes a node-resident service via **calls**. Services have general-purpose functionality, so we cannot make the same *a priori* guarantees about their resource usage. Nonetheless, as services are node-resident, the same review processes currently used for protocol deployment may be used. The proof sketches above do, however, provide a guideline for "SNAP-safe" services: they must execute in constant time and space. Naturally, more complex services are feasible (and probably desirable), but they are not covered by our existing safety framework.

V. FLEXIBILITY

We have seen how restrictions to SNAP's flexibility imply several important safety properties. To demonstrate that SNAP still retains enough expressibility to be useful, we developed a compiler that translates PLAN into SNAP. PLAN's flexibility is well-documented in the literature [6], [7], [8], [9]; our compiler thus ensures that SNAP remains useful. Indeed, of the six active applications mentioned in the introduction, two are currently implemented in PLAN, while at least three others could be.[3] Perhaps PLAN's most important application is its internetwork, PLANet [9]. In PLANet, *all* internetworking functionality is implemented using PLAN packets and router services, including address resolution, dynamic routing, encapsulation and fragmentation, error reporting, *etc*. Thus, our compiler demonstrates that we could likewise perform all of these internetworking functions with SNAP.

Since SNAP provides stronger safety guarantees through language restrictions, it is not feasible to translate *all* PLAN programs into SNAP. However, the PLAN programs that are ruled out are the ones with problematic resource usage, so we do not see this as an obstacle in practice. Indeed, all of the sample programs shipped with the current release of PLANet [9] may be encoded in SNAP.

In this section, we present the compilation techniques which make it possible to use SNAP as a compilation target. We also provide an example of the compiler's output for a PLAN *ping* program, for comparison against the hand-coded SNAP shown in Figure 1 in Section III.

A. Compiling PLAN to SNAP

PLAN is a purely functional programming language augmented with primitives for remote evaluation. It supports standard features, such as functions and arithmetic, and features common to functional programming, like lists and the list iterator *fold*[4]. A notable restriction is that functions may not be recursive; this is part of a guarantee that all PLAN programs terminate. Like SNAP, PLAN programs are packet-resident, and may call general-purpose *service routines* that are node-resident.

SNAP would be a fairly straightforward target language for PLAN if not for its lack of backward branches. Backward branches are typically used by compilers in three ways:

1. in returning to the caller after completing a function call
2. in calling a (mutually) recursive function
3. in returning to the head of a loop body

To deal with the first two points, we eliminate function calls to PLAN functions through inlining; this is straightforward because PLAN prohibits recursive functions (service function calls are translated directly to use the **calls** opcode). In PLAN, the only looping construct is the list iterator *fold*. In this case, we unroll the *fold*, and inline each call to the iterator function. Because the number of times the iterator is called depends on the length of the list, we cannot, generally speaking, know how many times to unroll the *fold*. Therefore, the user provides a conservative upper bound to the compiler. We do not expect this to be a problem in practice, as most uses of *fold* are on short lists of addresses, *e.g.*, for the multicast program in [6] or the flow-based routing program in [9].

PLAN differs slightly from SNAP in its execution model. PLAN programs do not evaluate on every active router they traverse, as SNAP programs do, but on the destination only. On the intervening PLAN nodes, a packet-specified "routing function" is evaluated instead, which determines the next hop and forwards the packet there. This routing function is specified as an argument to the PLAN packet transmission function `OnRemote`. To translate this model to SNAP, we add a piece of SNAP code to be evaluated on every hop: it checks if the packet has reached its destination, and if not looks up the next hop using the specified function and forwards the packet. If the packet has arrived at its destination, the code jumps to the entry point, stored on the top of the stack. For the special `defaultRoute` routing function, this bit of code is simply the **forw** instruction, followed by a **paj** ("pop and jump").

B. Compilation example: ping

To illustrate PLAN compilation, we present a simple example. The top of Figure 2 shows a PLAN version of ping. This program is sent into the network, and first evaluates the `p` function on the destination it wishes to ping. This function calls `OnRemote` to send a packet that will evaluate the function `r` back on the sender. This packet will use the `defaultRoute` routing function to get it back to the source (the routing function used to get to `p` is specified at send-time). When `r` is executed, it delivers some data (in the variable `w`) to the application listening on port numbered `n`.

The translation of this program by our compiler is shown in the lower portion of Figure 2. The `defaultRoute` routing function appears first, followed by code sequences for `r` and `p`.

[3] These applications are implemented in ANTS, which could easily be implemented as a set of PLAN services, and vice versa.

[4] Intuitively, *fold* executes a given function f for each element of a given list, accumulating a result as it goes.

```
fun r(w:blob,n:int) =
  deliverUDP(w,n)

fun p(w:blob,n:int) =
  OnRemote(|r|(w,n),getSrc(),
           getRB(),defaultRoute)
```

```
defaultRoute:
     forw                      ; forward to dest
     paj      -1               ; jump to top stack
                               ; value - 1
r:   pull     0                ; get port num
     pull     2                ; get data
     demux                     ; deliver data & exit
p:   pull     0                ; pkt stack: port
     pull     2                ; pkt stack: data
     push     r                ; pkt stack: paj offset
     push     defaultRoute     ; entry point
     push     3                ; how much stack
     getrb                     ; how much rb
     getsrc                    ; return to src
     send                      ; send return packet
     pop                       ; pop send return val
     popi     3                ; pop extra stack
     exit                      ; done
```

Fig. 2. Ping in PLAN and as compiled into SNAP

`r` simply copies the top two stack values using **pull**[5] and calls **demux** (which exits immediately after delivering its data). `p` constructs a packet to evaluate `r` on the source. It first constructs the stack for the call to `r` consisting of arguments `w` and `n`. Then it pushes the offset `r`, used by **paj** in the `defaultRoute` function. The remaining opcodes push the arguments to **send**: `defaultRoute` is the entry point, 3 specifies the amount of stack to take, **getrb** pushes all of the current packet's resource bound, and **getsrc** pushes the current packet's source address. Following the **send**, we pop the three arguments used in the sent packet's stack (**popi 3**), as well as the return value of **send**.

In general, our compiler does a decent job of translating PLAN to SNAP, and as shown in Section VII, the compiled code performs well. In fact, the program in Figure 2, while longer than the hand-coded 7 instruction version presented in Figure 1 in Section III, achieves about the same performance, as we show experimentally in Section VII. The compiled code is not fully optimized, especially for space; for example, all of the instructions following the **send** above could be eliminated. For the near term, we expect to use the compiler to achieve initial translations which we can then tune by hand as necessary.

VI. Implementation

We have implemented a SNAP VM in C, with both kernel-space and user-space versions. For the user-space implementation we have a daemon, *snapd*, that communicates with other daemons and user applications via UDP. The kernel version has been implemented for Linux kernel version 2.2.12 as part of RedHat Linux 6.1, and is accessible to user applications by a special socket type. SNAP is currently positioned as a transport layer protocol, although we ultimately intend for it to reside as a shim layer between layers three and four, using the IP Router Alert option [11] to flag the packets as active ones.

[5]This is not strictly necessary since the stack is already properly arranged for the call to **demux**; however our compiler does not yet do this optimization.

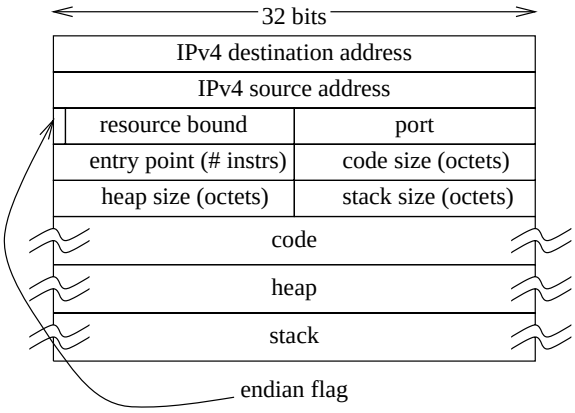

Fig. 3. SNAP packet format

We followed three implementation principles:

1. make execution as fast as possible
2. minimize the size of SNAP program representations
3. trade large initial fixed costs for incremental ones

In total, our intention was to reduce the overhead of SNAP, especially for common case programs like data delivery and diagnostics. In the remainder of this section, we discuss how we achieved these goals, covering the SNAP packet format, SNAP program representations, the structure of the interpreter, and the implementation of key operations, such as sending packets and checking well-formedness.

A. Packet Format

Our packet format is shown in Figure 3. The first portion of the header contains some standard header fields such as source, destination, and port, as well as the resource bound field that is used for bandwidth safety as described earlier.

The second portion of the header describes the SNAP program. Preceding the resource bound field is a flag to indicate the endianness of values in the packet program (all header fields are in network byte order), so that they may be converted if need be. The next header field is the "entry point", which indicates at which instruction execution is to begin. We then have fields that delineate the three main portions of the program: the code, heap, and stack. The program 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 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 [9] and ANTS [23] that require extensive unmarshalling before execution.

B. 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, allowing the packet to be arbitrarily relocated in memory (but 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 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 addresses and floating point values have to be allocated in the heap. We believe this is a minor limitation, as floats and addresses are used infrequently, and integers almost never require high precision in the context of simple packet programs. Instructions are similarly an n-bit opcode and a $32-n$-bit immediate corresponding to the data part of a stack value. We have 100 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.

C. 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.

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, situated at the end of the buffer, growing towards the stack. In our user-space implementation, we allocate a single buffer of size $4 \times MTU$, the maximum possible size required (as per Section IV), and receive all incoming packets into that buffer. In our kernel implementation, the buffer we receive from the kernel is not much bigger than the packet itself. Rather than immediately copy the packet into a maximally sized buffer, we do so only if needed. Execution proceeds in the given buffer until either a heap allocation takes place, or the current stack is about to overrun, at which point the copy is done. This permits simple executions to avoid the copy; *e.g.*, when the ping program in Figure 1 is forwarded, no copy is needed, but when it executes at the destination, the **getsrc** instruction causes an address to be allocated on the heap, resulting in a copy when the return packet is sent.

D. Implementing **send**

Send creates a new packet containing subsets of its parent's code, stack, and heap, and some of the parent's resource bound. Creating this new packet presents two difficulties. First, if any 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 [26]. This process ensures 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. We currently copy all of the code into the new packet; we could similarly employ a control-flow analysis to *prune* the code, as occurs in PLANet [9].

While this approach is general, it is both computationally and memory intensive. Fortunately, we can take more optimal approaches 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 or unmarshalling costs.

E. Well-formedness checking

Before a program may be executed, the interpreter must verify it is well-formed. Many systems that dynamically load code, notably Java [4] and proof-carrying code [16], verify that the *entire* code body is well-formed before allowing it to be executed. Depending on the size and complexity of the program, this may result in a large up-front cost, which we prefer to avoid. Instead, we intermix dynamic checks with interpretation, resulting in a performance improvement when the packet only executes a fraction of its instructions. This is often the case: most packets require frequent routing (implemented by the **forw** instruction), but only occasional computation; the programs in Figures 1 and 2 exhibit this property.

To reduce the number of dynamic checks, we are willing to verify as little as possible while still ensuring node integrity. As a result, we can avoid many of the checks that would normally be associated with *type-safety*. For instance, most interpreters would check that the argument to **not** is an integer and would signal a type error otherwise. However, we can omit this check, and *assume* that it is. This may result in "incorrect" program behavior but it will not compromise any of the safety properties.

VII. Efficiency

Having seen that the SNAP implementation is tailored for efficient execution, we now present experimental evidence that SNAP is efficient enough to be practical in many settings. To do so, we examine SNAP's performance in two areas. First, using our in-kernel SNAP implementation, we compare SNAP to IP in a software router setting, both for bandwidth and latency. Second, using the user-space implementation of SNAP and PLAN, we compare the latency of SNAP to that of PLAN, as well as examine the cost of using the SNAP to PLAN compiler.

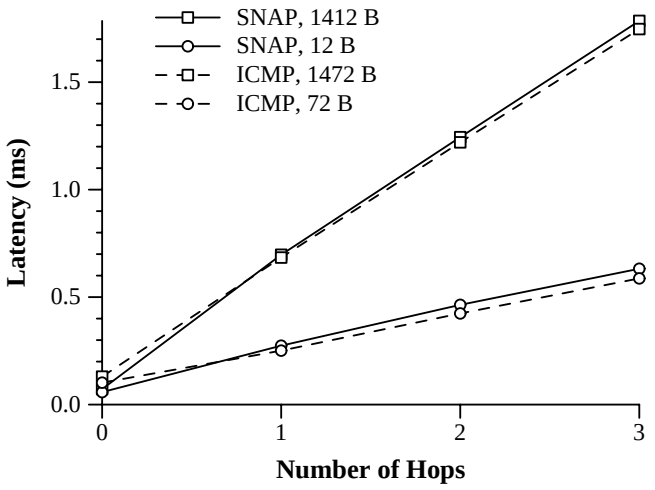

Fig. 4. Ping latencies

SNAP is competitive with IP and shows greatly reduced evaluation overheads compared to PLAN. SNAP latencies are no more 10% slower than IP's, depending on payload size and hop count. SNAP can saturate a 100 Mb/s Ethernet link using roughly 900 byte or greater sized packets, and can switch as many as 16,800 packets per second. These measurements compare favorably with IP on the same platform, which can switch as many as 17,300 packets per second, and saturate with roughly 800 byte packets. SNAP *ping* is between 2 and 3 times faster and 60% smaller than the comparable PLAN program. Thus, SNAP can be considered practical in domains where software routers are already practical[6].

All experiments were run on dual-CPU 300-MHz Pentium II systems with 256 MB of RAM. These machines have 16 KB split first-level caches and unified 512 KB second level caches and rate 11.7 on SPECint95. The machines run Linux kernel 2.2.12 and are connected by 100 Mb/s Ethernet links. Due to slightly skewed distributions, we report the medians of 21 trials, as per Jain [10]. The times are measured on a clock with a 4 μs granularity.

A. Comparing SNAP to IP

To compare SNAP latency to that of IP, we measured the round-trip times of SNAP ping, as shown in Figure 1, with the standard IP-based ICMP ECHO-REPLY [18]. The program overhead—namely, the size of the packet minus the IP header and payload—for using the SNAP-based ping was 68 bytes, while for ICMP it is 8 bytes. We adjusted the payloads for each program so that we had 100 byte (essentially minimal) and 1500 byte (maximal) Ethernet frames.

The results are shown in Figure 4; the y-axis shows latency in milliseconds (ms), and the x-axis presents the number of hops, *i.e.*, network links traversed (0 hops is a machine pinging itself). For maximally-sized packets, SNAP's latencies are at most 2% slower than those for IP, depending on the hop count, while they are as much as 10% slower for minimally-sized packets. For the 0-hop case, SNAP is actually faster for both packet sizes by more than 40%, implying that the cost of SNAP execution for the ping program is cheaper than the kernel's ICMP code.

[6]We believe that SNAP can also be efficiently supported in high-speed core router environments, but this is a topic for future research

	IP	SNAP
Per-packet (μs)	71	95
Per-byte (μs)	0.134	0.135

TABLE II
PER-NODE SWITCHING COSTS

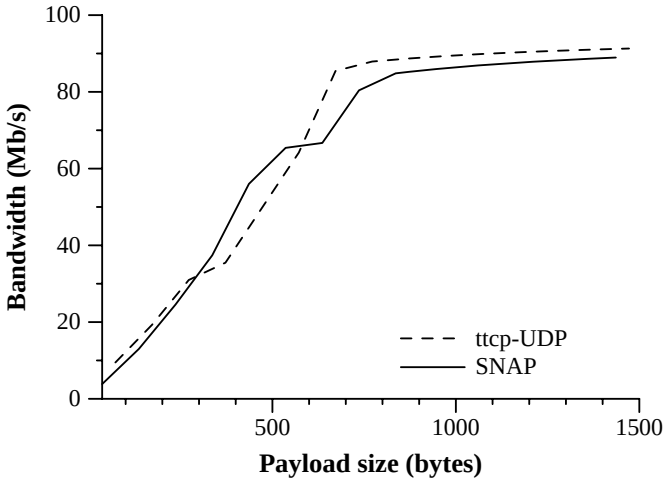

Fig. 5. Throughput measurements.

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 are shown in Table II. We see that the fixed cost per packet are about 24 μs (27%) higher for SNAP, while the per-byte costs are only 0.001 μs ($< 1\%$) higher. We believe the higher per-packet costs include the fact that IP must demultiplex the packet to SNAP, as well as the overheads of actual SNAP evaluation. We intend to instrument our kernel to measure these costs more carefully.

For bandwidth, we compared SNAP's equivalent of UDP (**forw** followed by **demux**) to the `ttcp` [1] load generator sending UDP packets. SNAP UDP has an overhead of 44 bytes as compared to 8 bytes for UDP. We performed a series of measurements on a three-machine configuration, varying the payload to result in Ethernet frame sizes of 100 byte increments, calculating the throughput for 5000 packets. The intermediate router is the system bottleneck.

The results are shown in Figure 5. The y-axis plots throughput in Mb/s (based on payload), while the x-axis plots the payload size in bytes. Both SNAP and UDP level off to saturate the link; SNAP's maximal bandwidth is slightly less than UDP's because of the additional 36 bytes of overhead. In both cases, the curves "ramp up" and level off, saturating the link. For UDP, this happens with roughly 800 byte packets, while for SNAP it occurs with 900 byte packets. An interesting feature of the graph is that between 300 and 500 byte packets, SNAP actually

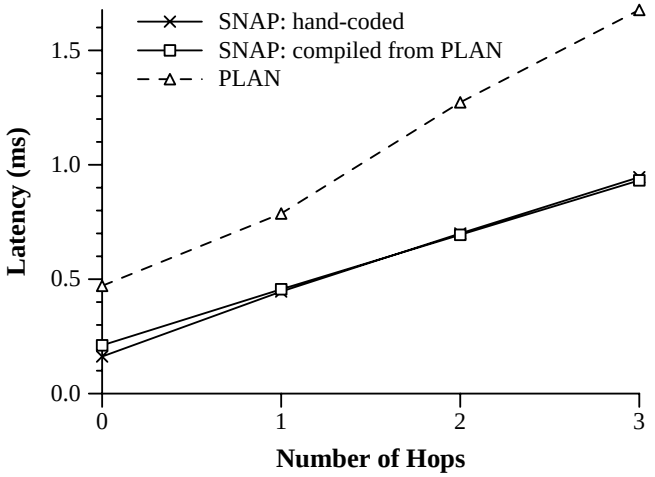

Fig. 6. Comparing SNAP ping to PLAN ping.

appears to outperform UDP. However, we believe this is more due to peculiar scheduling in the kernel as opposed to some improvement of SNAP over UDP. In particular, we have noticed that both SNAP and UDP are unintuitively able to switch mid-sized packets faster than small-sized ones. We suspect this is due to achieving synchrony with the device when packets are sent at a certain rate. We intend to investigate this behavior more thoroughly.

B. Comparing SNAP to PLAN

Not only is SNAP competitive with IP, but it significantly outperforms comparable active packet systems. To illustrate this, we compared SNAP to the PLANet [9] implementation of PLAN.

For our experiments, we used PLANet compiled to native code (bytecode is also an option), running on top of UDP. Because PLANet is a user-space implementation, we compared it to SNAP's user-space implementation, which also runs on top of UDP. PLAN's ping program (see Figure 2) has an overhead of 178 bytes. We used both the hand-coded version of SNAP ping (see Figure 1), which has an overhead of 68 bytes, and the version compiled into SNAP from the PLAN version (see Figure 2), which has an overhead of 112 bytes. In all cases we used 4-byte payloads.

The results are shown in Figure 6. Both of the SNAP versions perform essentially identically: about 1.8 times faster than PLAN for 1-3 hops, and nearly 3 times faster for 0 hops. In addition, the space overhead of SNAP is significantly less than that of PLAN: hand-coded SNAP ping is 62% smaller than PLAN ping, and compiled ping is 37% smaller. We are encouraged by the fact that our compiler's code, while more verbose, performs competitively with code produced by hand. More experience is needed to see if this holds true in general.

VIII. Related Work

In this section, we discuss previous research on active packets. We have summarized these systems with respect to flexibility, safety, and efficiency in Table III. While all of the projects have demonstrated utility derived from the flexibility of active packets, none of them has achieved a completely satisfying degree of safety *and* efficiency.

Project	Flexibility	Safety	Efficiency	
			Space	Speed
SPkts	fair	fair	good	*?*
ANTS	excellent	good	excellent	fair
PAN	excellent	poor	excellent	excellent
PLAN	excellent	good	fair	good
SNAP	excellent	excellent	good	excellent

TABLE III
Comparison of active packet systems.

BBN's SmartPackets [21] are used as mobile network management agents, using an SNMP-like [2] interface to query and configure nodes. Safety is largely ignored; the authors mention that an authorization-based scheme should be used. No execution data is available, but the program representation is optimized to be very compact.

ANTS [25] active packets, called *capsules*, contain a "pointer" to the code needed to handle them; this code is dynamically loaded on demand from previous nodes in a flow, essentially eliminating per-packet space overheads. Such a scheme may benefit SNAP as well. ANTS relies on its implementation language, Java, to provide safety and on watchdog timers to regulate resource usage. However, Hawblitzel *et al.* [5] have shown the practice of abruptly terminating subprograms in the same address space to be generally unsafe without adding significant overhead. Owing to Java, ANTS exhibits low throughput [24].

A follow-on project to ANTS is the PAN mobile-code platform [17], which essentially implements the ANTS model in-kernel, using native code. Not unexpectedly, PAN can achieve IP-like performance, but at the cost of no safety guarantees.

Finally, the PLAN project [7] is SNAP's direct predecessor, and similarly attempts to address safety concerns via language design. While all PLAN programs are guaranteed to terminate, it is possible to write exponentially long-running programs. Experimental results [9] show reasonable performance, but slow PLAN evaluation leads to the conclusion that active processing is unsuitable in the data plane.

IX. Conclusions

We have presented our second-generation active packet system, SNAP (Safe and Nimble Active Packets), which has three contributions. First, SNAP provides provable resource safety—linear bounds on bandwidth, CPU, and memory usage—through novel language restrictions. Second, despite these language restrictions, SNAP still retains the flexibility of first-generation systems, as demonstrated by our PLAN to SNAP compiler. Third, an efficient wire format and implementation achieve performance extremely close to that of an IP software router. Taken together, these three contributions establish that active packet systems can be practical for general use (at least where software routers are already practical). We expect to make our implementation available in the near future.

ACKNOWLEDGMENTS

The authors would like to thank Luke Hornof and Jessica Kornblum for comments on early drafts of this paper. We would also like to thank the anonymous referees for their helpful comments.

REFERENCES

[1] Network performance testing with TTCP. *The Network Monitor*, 3(1), 1997.

[2] J. Case, M. Fedor, M. Schoffstall, and J. Davin. A Simple Network Management Protocol (SNMP). RFC 1157, IETF, May 1990.

[3] S. Deering and R. Hinden. Internet Protocol, Version 6 (IPv6) Specification. RFC 2460, IETF, December 1998.

[4] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*. Addison Wesley, 1996.

[5] C. Hawblitzel, C.-C. Chang, G. Czajkowski, D. Hu, and T. von Eicken. Implementing multiple protection domains in Java. In *USENIX Annual Technical Conference*, June 1998.

[6] M. Hicks, P. Kakkar, J. T. Moore, C. A. Gunter, and S. Nettles. Network programming with PLAN. In *IEEE Workshop on Internet Programming Languages*, May 1998.

[7] M. Hicks, P. Kakkar, J. T. Moore, C. A. Gunter, and S. Nettles. PLAN: A Packet Language for Active Networks. In *ACM SIGPLAN ICFP*, September 1998.

[8] M. Hicks and A. D. Keromytis. A secure PLAN. In *International Working Conference on Active Networks (IWAN)*, June 1999.

[9] M. Hicks, J. T. Moore, D. S. Alexander, C. A. Gunter, and S. Nettles. PLANet: An Active Internetwork. In *IEEE INFOCOM*, March 1999.

[10] R. Jain. *The Art of Computer Systems Performance Analysis*. Wiley, New York, 1991.

[11] D. Katz. IP router alert option. RFC 2113, IETF, February 1997.

[12] U. Legedza and J. Guttag. Using network-level support to improve cache routing. In *Proceedings of the 3rd International WWW Caching Workshop*, June 1998.

[13] U. Legedza, D. Wetherall, and J. Guttag. Improving the Performance of Distributed Applications Using Active Networks. In *IEEE INFOCOM*, March 1998.

[14] L. Lehman, S. Garland, and D. Tennenhouse. Active Reliable Multicast. In *IEEE INFOCOM*, March 1998.

[15] J. T. Moore. Safe and efficient active packets. Technical Report MS-CIS-99-24, Department of Computer and Information Science, University of Pennsylvania, October 1999.

[16] G. Necula. Proof-Carrying Code. In *ACM SIGPLAN-SIGACT POPL*, January 1997.

[17] E. Nygren, S. Garland, and M. F. Kaashoek. PAN: A High-Performance Active Network Node Supporting Multiple Mobile Code Systems. In *IEEE OPENARCH*, March 1999.

[18] J. Postel. Internet Control Message Protocol. RFC 792, IETF, September 1981.

[19] J. Postel. Internet Protocol. RFC 791, IETF, September 1981.

[20] D. Raz and Y. Shavitt. An active network approach for efficient network management. In *International Working Conference on Active Networks*, July 1999.

[21] B. Schwartz, A. W. Jackson, W. T. Strayer, W. Zhou, R. D. Rockwell, and C. Partridge. Smart packets: Applying active networks to network management. *ACM Transactions on Computer Systems*, 18(1), February 2000.

[22] R. Wahbe, S. Lucco, T. Anderson, and S. Graham. Efficient Software-Based Fault Isolation. In *ACM SOSP*, December 1993.

[23] D. Wetherall. Active network vision and reality: lessons from a capsule-based system. *Operating Systems Review*, 34(5):64–79, December 1999.

[24] D. Wetherall. *Service Introduction in an Active Network*. PhD thesis, MIT, February 1999.

[25] D. J. Wetherall, J. Guttag, and D. L. Tennenhouse. ANTS: A Toolkit for Building and Dynamically Deploying Network Protocols. In *IEEE OPENARCH*, April 1998.

[26] P. R. Wilson. Uniprocessor Garbage Collection Techniques. In *Proceedings of International Workshop on Memory Management*, September 1992.