Safe and Efficient Active Packets

Jonathan T. Moore

University of Pennsylvania
Safe and Efficient Active Packets

Abstract
We present a new scheme for active, or programmable, packets based upon a new packet language, SNAP (Safe Networking with Active Packets). SNAP’s semantics permit us to prove that all SNAP programs are safe with respect to network resource usage and evaluation isolation. Furthermore, we describe an implementation of a SNAP interpreter, snapd, which achieves high performance for standard networking tasks. This work represents the first active packet system that is demonstrated to be both safe and efficient.

Comments

This technical report is available at ScholarlyCommons: https://repository.upenn.edu/cis_reports/181
Safe and Efficient Active Packets

Jonathan T. Moore *
jonm@dsl.cis.upenn.edu
Distributed Systems Laboratory
University of Pennsylvania

Abstract

We present a new scheme for active, or programmable, packets based upon a new packet language, SNAP (Safe Networking with Active Packets). SNAP’s semantics permit us to prove that all SNAP programs are safe with respect to network resource usage and evaluation isolation. Furthermore, we describe an implementation of a SNAP interpreter, snapd, which achieves high performance for standard networking tasks. This work represents the first active packet system that is demonstrated to be both safe and efficient.

1 Introduction

Increasingly widespread use of the Internet has placed new demands on the networking infrastructure. In particular, 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 [13], although certainly a key to its success as an internetworking protocol, often no longer fits users’ needs.

These new demands imply the need for the evolution of the networking infrastructure. However, IP is notoriously difficult to change (consider that IPv6 still has not been adopted), due to its centralized, committee-controlled nature. Active, or programmable, networks seek to address this problem by making the network extensible and thus more flexible. Active packets are one method of providing a programmable network interface; here, the traditional packet header is replaced with a program that is executed to determine its handling. Previous research [4, 17] has demonstrated the utility of active packets for new protocols or network configuration to improve application performance.

In order to be a usable shared resource, an active packet-based network must balance the opposing notions of safety and efficiency. It should not be possible for malicious or buggy active packets to crash network nodes or carry out denial-of-service attacks (safety), yet the overhead of executing per-packet programs must not be so great that the network becomes unusable due to its poor performance (efficiency).

We present an active packet scheme that addresses both safety and efficiency with the design of a network bytecode language, SNAP (Safe Networking with Active Packets). The SNAP language is designed with limited expressibility, thus making it possible to assert safety theorems about all SNAP programs. Furthermore, SNAP has been designed for efficient execution. This paper presents the relevant safety proofs as well as an experimental demonstration of the efficiency of a SNAP implementation.

In the next section, we provide an overview of previous related research, motivating the need for a new active packet scheme. In Section 3, we present a description of the SNAP bytecode language and outline the safety theorems for the language in Section 4. We then proceed in Section 5 to describe a SNAP im-

---

*This work was supported by DARPA under Contract #N66001-96-C-852.
plementation, with experimental results presented in Section 6. We give our concluding remarks in Section 7.

2 Related Work

In this section, we discuss previous research on active packets. In particular, we will focus upon the different tradeoffs that the designers have made in balancing safety versus efficiency. While all of the projects we will touch upon have demonstrated utility derived from the flexibility of active packets, none of them has achieved a completely satisfying degree of both safety and efficiency. Nonetheless, we draw from their experience in the design of our new active packet scheme.

First, we consider the SmartPackets project [14] from BBN. Here, network packets contain code in a language called Spanner, which is a CISC-style assembly language. These packets are used for network management and diagnostics via an SNMP-like [2] interface to the nodes. The SmartPackets, when used as mobile agents to move about and configure the network, reduce network management-related traffic by reacting to local conditions without having the communication overhead of a centralized remote management center. Schwartz et al. stress the need for a compact Spanner representation so that a meaningful program can fit within a single link-layer frame and be more robust in the face of an overloaded or failing network. Unfortunately, the design relies upon the use of heavyweight cryptography for authentication and access control, thus making them unsuitable for general payload transport.

The ANTS [17] system from MIT uses active packets called capsules, which conceptually contain both a data payload and the code for handling that payload. In practice, the packets merely contain a pointer to the code, which itself is dynamically loaded on demand from predecessor nodes. There have been numerous simulation-based demonstrations [7, 6] showing that ANTS active protocols can enhance application performance. However, the ANTS implementation was done in Java, relying heavily upon the safety properties of the language and requiring heavyweight verification at each hop. Indeed, this user-space interpretation allowed under 20 Mb/s throughput in practice [16].

A follow-on project to ANTS was the PAN mobile-code platform [9], which provides in-kernel support for mobile code. Nygren et al. demonstrate line rates (over 90 Mb/s) when dynamically loading x86 object code. However, this loading is done in an unsafe manner, thus making it possible for an active packet to crash the node. While achieving high enough performance to test active applications, this lack of safety makes PAN unsuitable for a shared network infrastructure.

Finally, the PLAN project [3] from the University of Pennsylvania attempts to address safety concerns via language design. The PLAN packet language has limited expressibility, so that authentication (e.g., by cryptography) is not required to execute packet programs. Notably, all PLAN programs are guaranteed to terminate, although it is still possible to write exponentially long-running programs. To balance the lack of expressiveness, PLAN programs may access node-resident services that provide more heavyweight functionality; since these services are more long-lived than ephemeral packets, the costs of safety verification may be amortized across many packets. Hicks et al. have built an active internetwork based on this two-level scheme called PLANet [4]. Experimental results show application benefits from using active packets in the control plane, yet slow PLAN evaluation leads to the conclusion that active processing cannot be done at every hop if reasonable performance is to be attained.

In summary, prior active packet implementations have either been too inefficient for general use as a replacement for an IP header, or they have achieved performance at the cost of node or network safety. In the next section, we describe our new active packet scheme, SNAP, which achieves safety via language design (like PLAN or ANTS), is suitably compact for use in packets (like SmartPackets), yet is amenable to high-speed interpretation (like PAN). SNAP is in many ways a descendent of PLAN, especially in terms of programming expressibility, yet SNAP offers stronger safety claims and addresses some of the performance concerns experienced in PLANet. In-
packet $P ::= (r, C_0, C, S)$ where:  
$r =$ resource bound  
$C_0 =$ full program  
$C =$ sequence of commands  
$S =$ stack of values

network $N ::= \{P_1, \ldots, P_n\}$

\[
\begin{align*}
\langle r, C_0, (\text{push } v) :: C, v :: S \rangle & \longrightarrow_{\text{local}} \langle r, C_0, C, v :: S \rangle  \\
\langle r, C_0, (\text{pop } v) :: C, v :: S \rangle & \longrightarrow_{\text{local}} \langle r, C_0, C, S \rangle  \\
\langle r, C_0, (\text{pull } n) :: C, v_1 :: \ldots :: v_n :: S \rangle & \longrightarrow_{\text{local}} \langle r, C_0, C, v_1 :: \ldots :: v_n :: S \rangle  \\
\langle r, C_0, (\text{bez } n) :: c_1 :: \ldots :: c_n :: C, 0 :: S \rangle & \longrightarrow_{\text{local}} \langle r, C_0, C, S \rangle  \\
\langle r, C_0, (\text{bez } n) :: C, v :: S \rangle & \longrightarrow_{\text{local}} \langle r, C_0, C, v :: S \rangle \text{ if } v \neq 0  \\
\langle r, C_0, (\text{op}P_i,j[k]) :: C, v_1 :: \ldots :: v_i :: S \rangle & \longrightarrow_{\text{local}} \langle r, C_0, C, v_1 :: \ldots :: v_i :: S \rangle \text{ if } v \neq 0  \\
\langle r, C_0, \text{send} :: C, n :: s :: r' :: d :: S \rangle & \longrightarrow_{\text{local}} \langle r - r', C_0, C, S \rangle  \\
N \cup \{\langle r, C_0, (\text{send} :: C, \text{null}) :: S \rangle\} & \longrightarrow_{\text{net}} N  \\
N \cup \{\langle r, c_1 :: c_n :: C, \text{send} :: C, n :: s :: r' :: d :: v_1 :: v_s :: S \rangle\} & \longrightarrow_{\text{net}} N \cup \{\langle r - r', c_1 :: c_n :: C, v_1 :: v_s :: S \rangle, \langle r', c_1 :: c_n :: C, v_1 :: v_s \rangle\}  \\
N \cup \{p\} & \longrightarrow_{\text{net}} N \cup \{p'\} \text{ if } p \longrightarrow_{\text{local}} p'
\end{align*}
\]

Figure 1: SNAP abstract machine.

deed, the performance results we present in Section 6 dispute the commonly-held belief that active packets are too slow to be evaluated at every node.

3 SNAP

In this section we describe our new active packet language, SNAP (Safe Networking with Active Packets). SNAP has been designed with limited expressibility, to promote the safety properties we will discuss in Section 4. In particular, all SNAP programs are guaranteed to terminate, and, more specifically, will run in time linear in their length. We have chosen a bytecode-style language to permit the application of current techniques in fast interpretation and compact representation (e.g., for Ocaml [1, 10]).

For purposes of space and simplicity, we present six main classes of SNAP bytecodes. Although there are actually far more bytecodes, many of them are simple primitive operations (e.g., arithmetic) and can be grouped together to specify the semantics. There are also some bytecodes that can be viewed simply as abbreviations for some sequence of these basic bytecodes, these sequences occurring frequently enough to justify the additional bytecode for the code space savings. Figure 1 describes the basic SNAP operations as reductions in a stack-based abstract machine.

In the SNAP abstract machine, we model the network as a multiset\(^1\) of packets, where each packet contains four fields: a resource bound counter (which will behave similarly to IPv4's time-to-live (TTL) field), a full program, a sequence of bytecodes left to execute, and a stack of values. Figure 1 describes two reduction relations: $\longrightarrow_{\text{local}}$, which describes the ex-

\(^1\)A semantics that distinguished nodes from one another might be more descriptive, but is not necessary for the safety proofs in the next section.
execution of one packet on a given node, and \( \rightarrow_{\text{net}} \), which describes the behavior of packets in a network. As a matter of notation, \( \rightarrow \) is a single-step reduction, \( \rightarrow^n \) is a sequence of \( n \) reductions, and \( \rightarrow^* \) represents an arbitrary number of reductions.

The first three rules specify the behavior of the stack manipulation bytecodes: push adds a value to the stack, pop removes the top value, and pull copies a value from within the stack to the top of the stack. The next two rules govern the behavior of the control-flow primitive bez ("branch if equal to zero"), which, if the top stack value is zero, skips the next \( n \) instructions. The sixth rule encapsulates most of the rest of the SNAP bytecodes (like arithmetic, comparison, etc.), where \( \text{op}_{i,j}[k] \) is the \( k \)-th primop with \( i \) stack value inputs and \( j \) stack value outputs.

The most novel instruction, send, creates a new packet that is to be sent elsewhere for execution. Concisely, \( \text{send}(n, s, r', d) \) sends a packet to destination \( d \) that will begin execution at the \( n \)-th instruction and will take the top \( s \) stack values as its initial stack. Furthermore, the parent packet will donate \( r' \) of its resource bound to the new packet, with one resource bound being consumed during transport. The grayed-out \( \rightarrow_{\text{local}} \) reduction (7) for send is provided only to support proofs in the next section; the \( \rightarrow_{\text{net}} \) reduction (9) more accurately describes the full effect.

Finally, rule (8) simply describes the normal termination of a packet program; when there are no instructions left to execute, the packet is discarded. Any transitions not covered in the above rules (e.g., running out of resource bound, type errors, stack underflow) are "stuck," and would result in some sort of error in an actual implementation.

### 4 Safety

In this section, we define more precisely what we mean for an active packet scheme to be safe to run in a network. These properties will be expressed in terms of theorems we would like to prove about an active packet programming system. We will then prove these theorems for SNAP, using the abstract machine description we provided in the last section.

#### 4.1 Desired properties

Safety is an important issue in a shared internetworking infrastructure—it should not be possible for malicious users to crash the network or otherwise make it unusable to others. Furthermore, the network must be robust to ill-formed packets (in the case of active packets, buggy packet programs). We follow the approach of Wetherall [16] and guarantee safety similar to that provided by IPv4.

There is an additional notion of safety that states that an active packet cannot crash or take over a node. This is typically implemented using techniques like software fault isolation [15], proof-carrying code [8], or dynamic interpretation checks. The SNAP semantics we presented earlier could be straightforwardly extended to include error transitions that encapsulate this kind of behavior, although we do not cover this extension here. Instead, we address some of the less well-understood aspects of node safety.

Firstly, we want to limit the amount of denial-of-service damage that can be inflicted. In other words, we would like to have some guarantees about the amount of network resources that can be used by active packets. There are three main resources that concern us: CPU time, memory, and bandwidth. For processing an IPv4 packet, a node spends \( O(\text{length}) \) CPU time (header processing, including options, plus the time to forward the packet out the correct interface), and requires \( O(\text{length}) \) memory per packet (which can be reclaimed after the packet has been processed). Finally, the maximum bandwidth consumed by a unicast IPv4 packet is \( O(\text{length} \times \text{TTL}) \) where \( \text{TTL} \) is the time-to-live field in the packet.\(^2\)

For an active packet scheme to have similar resource consumption behavior to that of the Internet, then, we require the following properties:

1. On any one node, processing a packet \( p \) should take \( O(|p|) \) time, where \( |p| \) is the length of \( p \).

2. On any one node, processing a packet \( p \) should require \( O(|p|) \) memory.

\(^2\)Note that this is not the case for multicast packets, which have even worse global behavior.
(3) The overall network bandwidth consumed by a packet \( p \) should be \( O(nlp) \), where \( n \) is some resource bound associated with \( p \) at its creation.

Secondly, it should not be possible for an active packet to directly affect the processing of other packets. Namely, we would like to ensure packet isolation:

(4) The processing of a packet \( p \) should be independent of other packets in the network.

### 4.2 Safety proofs

We will now proceed to show that SNAP has the four safety properties we detailed above, namely, bounded resource usage and processing isolation. Due to space restrictions, we present here only the high-level details; the full proofs are straightforward.

**Theorem 1 (CPU safety):** Let \( p = (r, C_0, C, S) \) be a packet, and let \( |C| = k \). Then for some \( n \leq k \), \( p \rightarrow^{n}_{\text{local}} (r', C_0, \emptyset, S') \).

*Proof.* By induction on \( \rightarrow_{\text{local}} \): each local reduction removes at least one instruction from the sequence of commands left to execute. \( \Box \)

**Theorem 2 (Memory safety):** Let \( p = (r, C_0, C, S) \) and \( p' = (r', C_0', C', S') \) be packets. If \( p \rightarrow^{*}_{\text{local}} p' \) then \( |S'| \leq |S| + |C| \).

*Proof.* This proof actually depends upon a property of the primitive operators \( \text{op}_{i,j}[k] \). Let \( \text{growth}(\text{op}_{i,j}[k]) = j - i \). Then each local reduction adds at most

\[
m = \max_{i,j,k} \{ \text{growth}(\text{op}_{i,j}[k]) \}
\]

values to the stack. Thus, we can limit the growth of the stack to a constant multiple \( m \) of the length of the packet. In the case of our particular set of bytecodes, \( m = 1 \). \( \Box \)

**Theorem 3 (Bandwidth safety):** Let \( p = (r, C_0, C, S) \) be a packet, and suppose \( \{p\} \rightarrow^{*}_{\text{net}} N \). Then \( |N| \leq r + 1 \).

*Proof.* By induction on \( \rightarrow_{\text{net}} \): only the send instruction creates a new packet; the other two \( \rightarrow_{\text{net}} \) reductions either decrease or maintain the number of packets in the network. Furthermore, with each send reduction, the total amount of resource bound in the network decreases by 1, therefore \( p \) (or \( p \)'s descendents) can cause the creation of at most \( r \) new packets. \( \Box \)

**Theorem 4 (Isolation):** If \( \{p\} \rightarrow^{*}_{\text{net}} N_p \) and \( N \rightarrow^{*}_{\text{net}} N_N \), then \( \{p\} \cup N \rightarrow^{*}_{\text{net}} N_p \cup N_N \).

*Proof.* Since the \( \rightarrow_{\text{net}} \) reduction affects one packet at a time,

\[
\{p\} \cup N \rightarrow^{*}_{\text{net}} N_p \cup N \rightarrow^{*}_{\text{net}} N_p \cup N_N. \Box
\]

Thus, the SNAP bytecodes adhere to the four safety properties presented in the previous subsection.

### 5 Implementation

We have written an implementation in C of a SNAP interpreter, snapd, that follows the semantics presented in Section 3. Currently, snapd runs as a Linux user-space application, with packets transmitted via UDP [11]. In this section, we describe how we have added detail to the general SNAP description to provide a working active packet system.

#### 5.1 Representation

First, we define the types of values supported by snapd\(^3\): 32-bit integers, 32-bit IPv4 addresses, and variable-length byte arrays. In order to support byte arrays (needed for payload delivery), we have added a "heap" to the packets. The actual byte arrays are stored in the heap, and their "stack values" are 32-bit offsets into the heap. Thus all of our stack values are the same size, permitting a straightforward and efficient stack implementation. Currently, stack values are 8 bytes (a 1 word type tag and a 1 word value); we are exploring ways to reduce this space overhead.

The addition of a heap does not violate our memory safety theorem from Section 4; when a byte array is copied via the pull instruction, only the

\(^3\)Note that SNAP does not specify the types of values, as the safety proofs rely only on the number of values in the stack, not their type.
Our packet format is shown in Figure 2. We have some standard header fields such as source, destination, and flow identifier, as well as the resource bound field that is used for bandwidth safety as described earlier. We then have fields that delineate the three main portions of the active packet: the code, heap, and stack. These three packet sections are laid out in this order to permit minimal copying during execution; the code and heap do not grow during execution, so as long as the containing buffer is big enough, the initial stack may grow and shrink as needed. The last header field is the "entry point" field, which indicates at which instruction execution is to begin; this permits the full generality of the send instruction without requiring that we retain two copies of the packet code.

The interpretation of most of the instructions is extremely straightforward, with the exception of the send bytecode. As the reader may recall from Section 3, send creates a new packet for execution that gets a subset of its parent's full program, part of the parent's stack, and some of the parent's resource bound. The marshalling for the new packet is very straightforward; the parent's code is copied (and the entry point is set appropriately), the resource bound is deducted from the parent, and the top few stack values are copied, along with the portions of the heap (if any) to which they refer. These copies, however, are expensive, as we will see in the next section; reducing or eliminating them is a topic for future research.

6 Performance

In this section, we present our experimental experience with snapd. In particular, we investigate its latency and bandwidth characteristics to support our claim that SNAP can be efficiently implemented. All the experiments in this section 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.

---

32 bits

<table>
<thead>
<tr>
<th>IPv4 destination address</th>
<th>IPv4 source address</th>
</tr>
</thead>
<tbody>
<tr>
<td>resource bound</td>
<td>flow identifier</td>
</tr>
<tr>
<td>entry point</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>code size (in octets)</td>
<td></td>
</tr>
<tr>
<td>heap size (in octets)</td>
<td></td>
</tr>
<tr>
<td>stack size (in octets)</td>
<td></td>
</tr>
</tbody>
</table>

Figure 2: snapd packet format

stack-resident offset is copied. Furthermore, none of our primitive operators return or modify byte array types, so a packet's heap cannot grow during execution.

We provide support for 53 bytecodes in snapd, including those for stack manipulation, control-flow, value operations, packet header access, routing table access, packet creation, and data delivery. All of these bytecodes can be modeled in the semantic framework of Section 3 either directly, as a member of the operators \( op_{1,j}[k] \), or as an abbreviation for sequence of other instructions. Currently, each bytecode instruction is 3 words (1 word opcode, 2 words for an immediate stack value). Uniform-sized instructions permit the straightforward implementation of branches and code entry points, as we are about to describe.

5.2 Packet Format

Our packet format is shown in Figure 2. We have some standard header fields such as source, destination, and flow identifier, as well as the resource bound field that is used for bandwidth safety as described earlier. We then have fields that delineate the three main portions of the active packet: the code, heap, and stack. These three packet sections are laid out in this order to permit minimal copying during execution; the code and heap do not grow during execution, so as long as the containing buffer is big enough, the initial stack may grow and shrink as needed. The last header field is the "entry point" field, which indicates at which instruction execution is to begin; this permits the full generality of the send instruction without requiring that we retain two copies of the packet code.

The interpretation of most of the instructions is extremely straightforward, with the exception of the send bytecode. As the reader may recall from Section 3, send creates a new packet for execution that gets a subset of its parent's full program, part of the parent's stack, and some of the parent's resource bound. The marshalling for the new packet is very straightforward; the parent's code is copied (and the entry point is set appropriately), the resource bound is deducted from the parent, and the top few stack values are copied, along with the portions of the heap (if any) to which they refer. These copies, however, are expensive, as we will see in the next section; reducing or eliminating them is a topic for future research.

6 Performance

In this section, we present our experimental experience with snapd. In particular, we investigate its latency and bandwidth characteristics to support our claim that SNAP can be efficiently implemented. All the experiments in this section 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.

---

4 Recall that our memory safety theorem allows us to put an upper bound on this size.
5 snapd is single-threaded, so we do not take advantage of this parallelism.
forw ; move on if not at dest
bne 4 ; skip 4 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

Figure 3: SNAP code for ping.

6.1 Latency

First we measured snapd’s latency, using round-trip times. The code for our ping packet is shown in Figure 3, and it is sent with the initial stack (0 :: portnum :: payload). We will now walk through the execution of this packet as an illustration of some of snapd’s bytecode set.

6.1.1 Ping benchmark

Figure 3 shows the SNAP instructions used for our ping program. The first instruction executed is the forw bytecode. This compares the packet’s destination header field to the current host’s address. If they are different, then the packet is forwarded towards its destination. More specifically, a send is executed that donates all the remaining resource bound, takes the same code and entry point, and copies the whole stack and heap; the original packet then exits.

When the packet reaches its destination, forw simply drops through to the next instruction. The bne checks whether the top stack value is nonzero. In our example, since 0 is on top of the stack, the branch falls through and the 0 gets popped. Next, we push a 1 onto the stack—essentially replacing the earlier 0, indicating we are now on the way back. The getsrc instruction pushes a copy of the source header field onto the stack. The return packet is then generated via the forwto instruction, which sets the source header field to the current host and the destination header field to the top stack value (in our case, now the original source), pops this latter argument, keeps the same entry point, donates all remaining resource bound, and carries the whole stack (now 1 :: portnum :: payload).

On the return path, as before, most hops will just execute the forw instruction, although since the destination header field is now set back to the original source, the packet will progress in the correct direction. Upon reaching the source, the forw falls through, the bne skips 4 instructions6 (since the 1 is on top of the stack), and the demux instruction delivers the payload to the correct application port.

6.1.2 Measurements

We now present the results of executing the above ping program across several snapd nodes connected linearly. The size of the ping program is 148 bytes, so since we are encapsulated inside UDP and IP, the minimum Ethernet frame size is 176 bytes. We also measured the latencies for payloads resulting in 750 byte and 1500 byte frames. We contrasted these results with those obtained by running the standard ICMP ECHO REPLY [1121] ping over the Linux kernel router. The payloads for the ICMP pings were set to result in the same 176, 750, and 1500 byte Ethernet frame sizes.

The results are shown in Figure 4; the y-axis shows latency in microseconds, and the x-axis presents the number of hops, i.e., network links traversed (so 0 hops is a machine pinging itself). Most strikingly, the snapd latencies are on the same order of magnitude as those for the Linux kernel version.

Based on the slope of these lines for the two implementations at multiple payload sizes, we can estimate

---

The pop instruction is only used in the case where the destination and source are the same host; then when the packet is “at the destination,” the forwto will fall through, so we have to pop the 1 we just pushed before demuxing.
per-hop and per-byte costs for both snapd and the Linux kernel as shown in Table 5. The higher per-byte cost for snapd can be attributed to the fact that forw requires the marshalling of a new copy of the packet. Most of the difference, though, comes from per-packet costs; since snapd is in user space, there are two kernel crossings per packet. We instrumented snapd with timers to verify these estimates—for maximal packets, the main receive-interpret-send loop takes 129 µs, of which 98 µs is spent in the recvfrom and sendto system calls. This suggests that an in-kernel implementation of SNAP would have significantly lower latency.

6.2 Bandwidth

For the bandwidth measurements, we use three machines connected linearly: a load generator, a forwarder, and a receiver. The load generator and receiver do not run snapd, but rather send SNAP packets directly to the forwarder so that we can directly measure routing throughput. We use a 2-instruction SNAP program (forw and demux) for payload delivery. This permitted us a minimum Ethernet frame size of 108 bytes, and we take measurements for Ethernet frame sizes in multiples of 100 bytes. We use ttcp to measure the throughput of the Linux kernel router for comparison, setting the buffer sizes to result in the same sized Ethernet frames as for snapd. The bandwidths presented are in terms of useful payload delivered, i.e., not including header or program size, so snapd is penalized for tunneling inside UDP.

We took four sets of measurements: ttcp sending UDP (which often had high packet loss), ttcp using TCP with the NODELAY option set, our SNAP sender running full blast, and our SNAP sender sending just slowly enough to ensure at most a 1% packet loss. The results are shown in Figure 6. The y-axis plots throughput in Mb/s, while the x-axis plots the resulting Ethernet frame size.

Notice that the ttcp curves have knees in them where the payload sizes become large enough that the bottleneck switches from CPU to I/O. There is no such obvious knee in the snapd curves, however. Since snapd must interpret its packets, the higher computation overhead results in a lower-sloped “ramping up” section. Unfortunately, this slope is low enough that we reach the maximal Ethernet frame size before levelling off; high payload sizes would result in fragmentation that would cloud the issues. Nonetheless, at the higher payloads, snapd with a loss rate under
1% sustains a bandwidth higher than that of TCP (though not yet quite as high as UDP). By reducing our kernel crossing overheads as mentioned in the latency subsection, we could hope to see a reduced CPU overhead, perhaps seeing a more pronounced knee similar to that of the Linux kernel router.

7 Conclusions

We have designed a new active packet language, SNAP (Safe Networking with Active Packets), and presented a formal semantics for it. We have used this semantic framework to prove that SNAP programs are safe in terms of resource usage and isolation. To our knowledge, this is the first formal proof of safety for an active packet scheme.

Furthermore, we have implemented a SNAP interpreter, snapd, that achieves high performance for standard networking tasks, despite running in user space over UDP. In fact, these results challenge the commonly-held belief that active packet processing must be relegated to the control plane. Indeed, the current implementation is already fast enough for all but the fastest backbone routers in the Internet. A future, in-kernel implementation promises to reduce our overhead, improving snapd’s performance even further.

To be usable in a shared network infrastructure, an active packet system must be both safe and efficient. Previous active packet schemes have either not adequately addressed safety and security, or they have done so at the cost of performance. SNAP and snapd provide the first active packet system that is demonstrably both safe and efficient.

Acknowledgments

The author would like to thank Michael Hicks, Luke Hornof, Jessica Kornblum, and Scott Nettles for comments on early drafts of this paper. Special thanks to Michael Hicks for providing the Pentium cycle counter timing package and providing insight for performance measurement and tuning.

References


