# Smart Packets: Applying Active Networks to Network Management

BEVERLY SCHWARTZ, ALDEN W. JACKSON, W. TIMOTHY STRAYER,
WENYI ZHOU, R. DENNIS ROCKWELL, and CRAIG PARTRIDGE
BBN Technologies

Smart Packets is an Active Networks project focusing on applying active networks technology to network management and monitoring. Messages in active networks are programs that are executed at nodes on the path to one or more target hosts. Smart Packets programs are written in a tightly-encoded, safe language specifically designed to support network management and avoid dangerous constructs and accesses. Smart Packets improves the management of large complex networks by (1) moving management decision points closer to the node being managed, (2) targeting specific aspects of the node for information rather than exhaustive collection via polling, and (3) abstracting the management concepts to language constructs, allowing nimble network control.

This paper introduces Smart Packets and describes the Smart Packets architecture, the packet formats, the language and its design goals, and security considerations.

Categories and Subject Descriptors: C.2.1 [**Computer Communications Network**]: Network Architecture and Design

## 1. INTRODUCTION

Active Networks [Tennenhouse and Wetherall 1996; Wetherall and Tennenhouse 1996; Bhattacharjee et al. 1996; Tennenhouse et al. 1997] is a framework within which users inject programs contained in messages into a network capable of performing computations and manipulations on behalf of the user. Nodes along the network receive these messages, execute the programs within, and (possibly) return values or forward the messages along to other nodes. The Active Networks Program is a DARPA-sponsored research program seeking to sharply increase the programmability of computer networks and network components. The program seeks to increase user and application control over how packets are handled, and to increase the flexibility of computer networks and the ability to upgrade them.

A major motivation behind Active Networks was the theory that there is an exponential growth of computing power in the network suggested by Moore's Law, which states that the speed of electronic components doubles every 18 months. Unfortunately, in most parts of the Internet, the traffic growth rates far exceed the growth rate of Moore's Law. As a result, there is typically *less* effective computing power per packet every year.

Authors' address: BBN Technologies, 10 Moulton St, Cambridge MA 02138
This work was sponsored by the Defense Advanced Research Projects Agency and issued by the NCCOSC under contract N66001-96-C-8517.

### 1.1 Why Do Active Network Management?

There are places, however, where Moore's Law is winning. One place is network management and monitoring. The average device is not generating, processing or receiving drastically more network management traffic than it was a year or two ago. We can hope, therefore, that there is more per-device processing power available for network management than there was in the past, especially since modern router architectures tend to place network management functions on a distinct processor [Partridge et al. 1998].

At the same time, the exponential growth of the Internet is overwhelming management centers. Currently, network management is achieved by having management stations routinely poll the managed devices [Rose 1994; Stallings 1996] looking for anomalies [Paxson 1997b]. As the number and complexity of nodes increase, management centers become points of implosion, inundated with large amounts of redundant information when components are in the same state they were in previously. This passive network solution does not scale and is not cost effective. Furthermore, a component can suffer multiple state changes in less than one round-trip time and, indeed, can oscillate per packet [Paxson 1997a]. It is essential that network management employ techniques that require less communication and permit more effective action on the managed node itself.

Smart Packets seeks to exploit the increase in processing power within the control side of the router to help provide network management. Smart Packets puts active networks technology into the management of the network to make managed nodes programmable. Management centers can then send programs to the managed nodes. This approach has three advantages. First, the information content returned to the management center can be tailored (in real-time) to the current interests of the center, thus reducing the back traffic as well as the amount of data requiring examination. Second, many of the management rules employed at the management center can now be embodied in programs which, when sent to managed nodes, automatically identify and correct problems without requiring further intervention from the management center. Third, Smart Packets shortens the monitoring and control loop—measurements and control operations are taken during a single packet's traversal of the network, rather than through a series of set and get operations from a management station.

The Smart Packets architecture consists of four parts: (1) a specification for smart packet formats and their encapsulation into some network data delivery service, (2) the specification of a high level language, its assembly language, and a compressed encoding representing that portion of a smart packet that gets executed, (3) a virtual machine resident in each networking element to provide a context for executing the program within the smart packet, and (4) a security architecture.

### 1.2 Prior Work

Earlier projects have attempted to put a programming language into a network management system. In the late 1980's, the High-level Entity Management System (HEMS) [Partridge and Trewitt 1988] used a query language [Trewitt and Partridge 1988] tuned to the monitoring and control of network entities. While the HEMS query language provides insight into methods to request and modify host data, it does not meet our needs. The language is really an extended database query language, designed to extract large amounts of data from a node. It does not allow general programming. It gives the packet no control over where it is sent, and places no limits on the size of a query. The current Internet network management standard, the Simple Network Management Protocol [Davin et al. 1987; Case et al. 1990], was developed as a competitor to HEMS and because of concerns about complexity chose to put each extraction operation in a separate packet.

More recently, a set of platform-independent programming languages, most notably Java [Gosling et al. 1996; Arnold and Gosling 1997], PLAN [Hicks et al. 1998] and CAML [Leroy 1996a; Leroy and Mauny 1993; Leroy 1996b], have been created. These program languages are designed to be transmitted across the network, between machines. However, if the HEMS language was too simple, the newer languages are too rich for reasons discussed in more detail in section 4.1.

## 2. SYSTEM ARCHITECTURE

The Smart Packets project is designed to demonstrate that network management is a fruitful target for exploiting active networks technology. As a result, there is a temptation to provide the richest and most flexible programmable environment possible. At the same time, we are concerned that if we made the environment too rich we could still overload the computing power of the managed node and, further, create an environment so rich that it would be hard to secure.

In our attempts to balance these concerns, we made two important design decisions. First, there should be no new persistent state in routers across packets. Keeping persistent state in network nodes, especially routers, is expensive and creates management and consistency problems. Consequently, programs sent in smart packets must be completely self contained. This goal implies the transport service should be connectionless; even fragmentation of the smart packet is not permitted. The Smart Packets programming language must be able to express meaningful programs in under 1 Kbyte in length.

Second, there should be a virtual machine used to provide surety of safety while executing the programs carried in smart packets. Remote execution of code is dangerous, so to mitigate the possibility of damage to the executing host, an insular environment is established where operations are controlled. The language that is interpreted by the virtual machine is also designed to avoid dangerous (and for network management, superfluous) features, like file system access and memory management.

Figure 1 shows the Smart Packets system architecture. User-written network management and monitoring programs generate smart packets—encapsulated in Active Network Encapsulation Protocol (ANEP) [Alexander et al. 1997] frames—and give them to the ANEP Daemon process. The daemon injects the smart packet into the network, where the smart packet is sent in either an end-to-end or a hop-by-hop mode. In end-to-end mode, the program is executed only at the destination. In hop-by-hop mode, the program is executed at the source, destination, and all hops in between. The program can contain directives to send results to the source from any host it is executed on. Directives also exist to exclude program execution at the source and destination nodes.

The ANEP Daemon process has two responsibilities: it is the injection and recep-

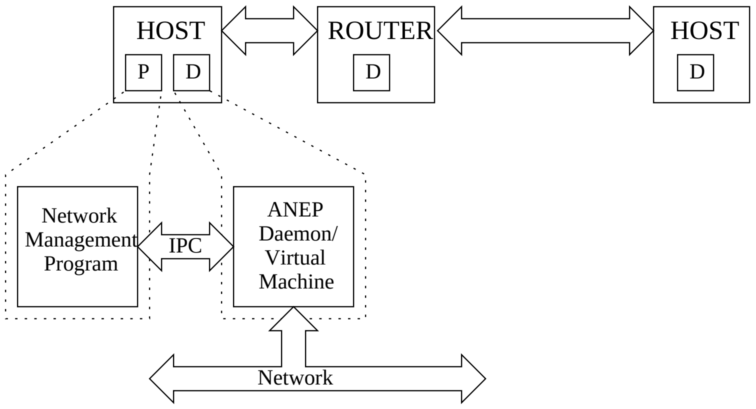

Fig. 1. IP and ANEP encapsulation

tion point for smart packets, and it also contains the virtual machine for executing the programs received. The virtual machine provides a clean, safe context with a well-defined, securable interface into the rest of the system.

## 3. TRANSMISSION OF SMART PACKETS

A smart packet consists of a Smart Packets header followed by payload. The smart packet is encapsulated within an ANEP packet which, in turn, is carried within IPv4, IPv6, or, in some Active Networks testbed situations, UDP.

### 3.1 Transmission Issues

One challenge in implementing Smart Packets is that IP does not have a notion of a datagram whose contents are processed at intermediate nodes. An IP router simply examines the datagram header and forwards the datagram. For a smart packet, however, the router must process the contents of the datagram before forwarding it. As a further complication, the router should examine the contents of the datagram only if the router supports Smart Packets. Otherwise, the router should pass the datagram through.

We achieved this goal by modifying an IP option—Router Alert [Katz 1997; Partridge and Jackson 1999]—to achieve the desired behavior. Based on a tag in the option, and possibly an examination of some of the higher-layer headers, the router can determine if it should process the datagram contents. If the router doesn't support Active Networks, it ignores the option and forwards the datagram. If the router supports Active Networks, it examines the ANEP message, learns the message is a Smart Packets packet and, if the router supports Smart Packets, it processes the packet.

### 3.2 Packet Format

Smart packets contain either a program, resulting data, or messages wrapped within a common Smart Packets header and encapsulated within ANEP. Figure 2 shows the format of a smart packet. The Smart Packets header has four fields: version number, type, context and sequence number. The version number is used to identify language upgrades and packet format changes. The type field identifies the message as one of four types: a Program Packet, a Data Packet, an Error Packet, or a Message Packet.

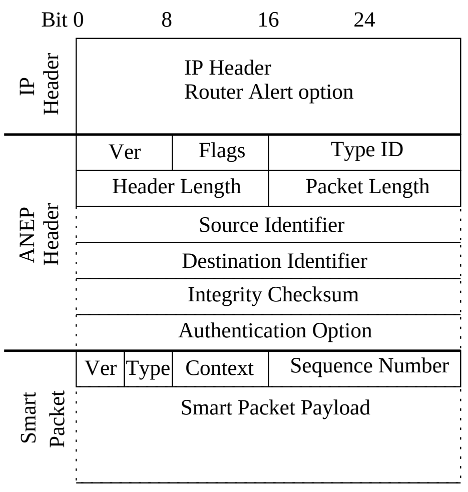

Fig. 2. A smart packet with IP and ANEP encapsulation

The Program Packet carries the code to be executed at the appropriate hosts. The Data Packet carries the results of the execution back to the originating network management program. The Message Packet carries informational messages rather than executable code. Error Packets return error conditions if the transport of a Program Packet or the execution of its code encounters exceptions. Only Program Packets use the IP Router Alert option.

The context field holds a value that identifies the originator of the smart packet. The context value is generated by the ANEP Daemon for each client, and is unique for each client within a particular host. The value is placed into outgoing Program Packets. As Program Packets traverse the network and generate one or more responses (Data, Error, and Message Packets), the context value is used to identify the client to which the responses must be delivered.

The sequence number field holds a value that is used to differentiate between messages from the same context. This value allows a client to match response packets with injected programs. Like the context field value, response packets echo the sequence number value of the Program Packet.

## 3.3 Encapsulation

The Active Networks Encapsulation Protocol was developed for the DARPA Active Networks Program to facilitate portability and interoperability with other Active Networks projects. Smart packets are encapsulated within ANEP packets.

The layout of an ANEP Header is shown in the middle of Figure 2. Smart packets use four optional features of ANEP: the source address, the destination address, the checksum (which computes an IP checksum over the entire ANEP header and payload), and the authentication option.

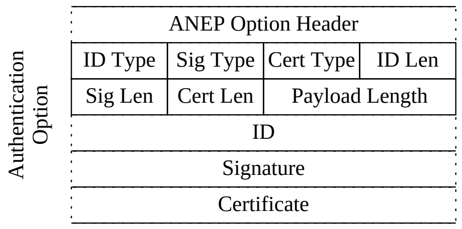

Fig. 3. ANEP Authentication Option

The authentication option, modified from that proposed by Spatscheck [Spatscheck 1997], is shown in Figure 3. It identifies the sender of the message, includes a digital signature over the entire ANEP header and selected portions of the smart packet, and contains a public-key certificate.

The ANEP option header is common header for all ANEP options. It holds fields that identify the option type, any sub-type information needed to parse the option, and the length of the option:

(1) The *ID* field holds an IPv4 or IPv6 address, identified by the ID type and length fields. The value of the ID field is identical to the value in the source address field of the ANEP source address option.
(2) The *signature* field holds a digital signature, identified by the signature type and length fields. The valid types of digital signature algorithms for smart packets are either DSA over SHA-1 or RSA over MD5.
(3) The *certificate* field holds an X.509 public-key certificate [Housley et al. 1999], identified by the certificate type and length fields. The certificate holds the value of the IPv4 or IPv6 address in its Subject Alternative Name extension field.
(4) The *payload* field holds the length of the data in the ANEP message that is covered by the data in the signature field.

The use of the authentication option is discussed in section 7.

## 4. PROGRAMMING LANGUAGES

We developed two programming languages in support of Smart Packets. The first language, Sprocket, is a high-level language much like C, but with security-threatening constructs such as pointers removed, and network and network management features such as built-in types for MIB access and packets added. The second language, Spanner, is a CISC assembly language. Sprocket programs are compiled into Spanner code which, in turn, is assembled into a compact machine-independent binary encoding that is placed into Program Packets. Sprocket and Spanner are equivalent languages; the difference is that tighter programs can be written in hand-optimized Spanner code.

### 4.1 Language Issues

Spanner is designed to yield very small encoded programs. Since each program stands alone, the strength of Spanner depends on its ability to fit as much code into a packet as possible. The practical limit on program size is 1 Kbyte, or roughly an Ethernet frame less headers and authentication data. Meaningful programs include those able to perform networking functions and MIB information retrieval, operations normal to network management. The language provides these complex concepts as primitives.

The language is designed for safety—a program must not be allowed to do harm to a host or router past the limit of its authorization. (For the rest of this section, we will use the generic term node for a host or router). To this end, the language does not include file system access or general system calls. Yet, a Smart Packets program has to be able to manipulate the node (through MIB variables and other parameters) as well as manipulate its own packet's contents (through filling in the data area in the packet, and the generation of new packets).

After surveying a large set of current programming languages, none were found that could encode more than a trivial program in a compact, platform-independent encoding. To illustrate the comparisons of code sizes, a recursive function for computing Fibonacci numbers was written and compiled into the byte code representation offered by several languages. All implementations used the same algorithm, shown in Figure 4, which is described in Abelson, Sussman, and Sussman [Abelson et al. 1985]. Table 1 shows the size of the resulting compiled byte code representations.

```
(define (fib n) (fib-iter 1 0 n))

(define (fib-iter a b count)
  (if (= count 0)
      b
      (fib-iter (+ a b) a (- count 1))))
```

Fig. 4. Scheme source to compute Fibonacci numbers

| Language | Size (bytes) |
|---|---|
| Emacs Lisp (v19.34) | 617 |
| Java | 370 |
| OCAML (v1.03) | 294 |
| Sprocket | 91 |
| Spanner (hand optimized) | 51 |

Table 1. Relative sizes of byte codes for Fibonacci function

The byte code file from Emacs Lisp includes comments and additional error checking code. Stripping the comments and the added code reduces the byte code to a reasonable 139 bytes. Objective CAML (OCAML v1.03), a functional programming language related to ML, produces a byte code file of 294 bytes. OCAML is strongly typed and features automatic memory management and incremental garbage collection. Nevertheless, the support for functions written in C, both user level and system calls, appeared to be awkward to extend, and it is advantageous for Smart Packet programs to use existing C-based methods to monitor system resources.

The concerns about Java are more subtle. Java is a very rich language, with many primitives such as file access and GUI manipulation that Smart Packets neither needs nor wants. Yet Java lacks features such as MIB access that Smart Packets needs. Today, most of these problems could be solved using Java's Native Interface (JNI) [Liang 1999] but the JNI was still under development when we selected the Smart Packets language. Other concerns about Java are that the Java Virtual Machine Language is based on a RISC architecture, and so trades increased code size for speed (exactly the reverse of the desired Smart Packets tradeoff), and that Java Virtual Machine would have to be modified to support Smart Packet's security requirements (e.g., counting instructions and memory allocation restrictions).

## 4.2 Sprocket

Sprocket was designed to be both as familiar and as simple as possible without leaving out desired functionality. Sprocket is based on C's grammar and keywords, but with the removal of constructs unnecessary for Smart Packets use, including enumerations, typedefs, structures, and unions. Sprocket includes C++-style comments and declarations anywhere within the scope. Parameters to functions can be passed by value or by reference.

C primitive types like int, short, and char have been replaced with types that explicitly indicate the type size and sign, and include sizes up to 128 bits. Floating point values can be either 32 or 64 bits wide. There are built-in types for arrays, strings, and lists. In addition, four new complex types are included as base types: a packet type, an address type, a smart packet identifier type, and a MIB context type. Address and packet are abstract types, since Sprocket can work in several routing environments (IPv4, IPv6, and the Active Networks testbed). The contents of the address and packet types automatically take the appropriate environment-specific form. Since most (if not all) routing environments use the same attributes (e.g., source address, destination address), these abstract types include methods to manipulate these concepts (e.g., "get source address," "set destination address").

Sprocket's complex types have operations that can be performed on instances of that type. Sprocket uses C++ member function syntax to call these operations. For instance, the packet type has an operation send. The syntax to call the send function on an instance pkt is pkt.send().

There are self-referential operations, that is, operations on the Program Packet itself, for retrieving header information. These operations are useful for recording how the smart packet is acted upon as it transits the network. These operations are not associated with a type; the object on which these methods operate is assumed. Additional operations control the loading and unloading of data, called baggage, also carried in this Program Packet.

There are also operations for retrieving node MIB information, through both general queries to the MIB and short-cut primitives. Many of these short-cut primitives deal with the often-retrieved data on the interface devices. The general queries allow a user to navigate a MIB database and to retrieve data using SNMP-style operations (e.g., get, get next, set).

Figure 5 shows a small Sprocket Program. This program gets the number of

```
main() {
   /* variable declarations:
    * addr: for getting addresses later in the code
    * pkt: a packet which will be sent to the originating node
    * num_interfaces: contains the number of interface devices
    * num_addr: contains number of addresses for an interface
    * index: a loop index */
   array of address addr;
   packet pkt;
   unsigned16 num_interfaces = num_ifaces();
   unsigned8 num_addr;
   unsigned16 index;

   /* put the number of interfaces into pkt's data area */
   pkt.data_append(num_interfaces);

   /* find out information about each of the interface devices
    * on the node */
   for (index = 1; index <= num_interfaces; i++) {

      /* determine the number of addresses for the interface; it
       * could be more than one if there are alias addresses */
      num_addr = num_addresses(index);

      /* only need to get addresses if the interface has some */
      if (num_addr > 0) {

         /* set array size to number of addresses and get the
          * addresses */
         addr.set_size_of_dimensions(num_addr);
         get_addresses(addr);
      }

      /* put the following info in pkt's data area:
       *     the interface device number
       *     the addresses (if there are any)
       *     the mtu */
      pkt.data_append(index);
      if (num_addr) pkt.data_append(addr);
      pkt.data_append(get_iface_mtu(index));
   }

   /* send the packet to the originating node */
   pkt.send();
}
```

Fig. 5. Example Sprocket Program

interface devices, and, for each device, gets all of the addresses associated with that device (this will be especially useful with IPv6, where devices can routinely have multiple addresses) and the MTU for the device. This information is put into a packet and sent back to the originating node.

The num_ifaces() call in the declaration section of the program in Figure 5 is an example of a MIB query that is a primitive. We can also get the number of interfaces through the general MIB query primitives. Figure 6 illustrates how to access the number of interfaces through general MIB queries.

## 4.3 Spanner

Spanner is the assembly language for Smart Packets. Spanner reflects a stack-based CISC architecture that differs from traditional assembly languages in two important ways: Spanner has declared variables, and has no access to memory, so storage is

```
/* the following is equivalent to
 * unsigned16 num_interfaces = num_ifaces(); */
mib m;                        // declare a MIB context (initialized to MIB-II)
m.down(2,1);                  // append ".2.1" to the MIB address
                              // (this MIB address is for number of interfaces)
unsigned16 num_interfaces = m.getn_unsigned32();
                              // perform a get_next (which gets num ifaces)
```

Fig. 6. Example use of General MIB Query Primitives

either on the stack or in variables.

Variable declarations are probably the most novel feature of Spanner. Assemblers traditionally do not declare variables, but simply label (and possibly initialize) portions of memory. Since the virtual machine does not have general purpose memory, variables must be created explicitly. Furthermore, the virtual machine needs to know type information to enforce type safety. It is more efficient to explicitly state the variable type once, and expect the virtual machine to keep track of the type, than to have to encode the type information with every use of the variable.

Spanner's types correspond directly to the types in Sprocket, and almost all primitives in Sprocket have direct equivalents to operations in Spanner (some are redundant). There are about 200 such operations. The result of the operation may or may not be pushed on the stack, according to the default for the operation and the programmer's override of the default. Operations to manipulate the stack are also provided, including popping an item off the stack, pushing a value on the stack, and pushing a reference to a variable on the stack. Most Spanner operations affect the virtual machine's condition code as well.

Spanner operations generally map directly to primitives in Sprocket. Spanner does the appropriate automatic type promotion when necessary. Spanner has operations that perform on arrays, lists, strings, addresses, smart packet identifiers (spid), packets, and MIB contexts. For each of these operations, there is a corresponding primitive in Sprocket—using the C++ member function syntax as described above—which operates on an instance of an array, list, a string, an address, a spid, a packet, or a MIB context. There are also operations that provide direct access to the information about the packet the Spanner program arrives within, such as getting source and destination address and appending data to the end of the baggage area.

Spanner provides branch operations—branch if the condition code is set, branch if the condition code is cleared, and branch unconditionally. The Sprocket compiler uses these operations to implement flow control. Spanner also provides support for subroutines: there are jump to and return from operators.

Spanner also provides primitives for sending Data and Message Packets back to the source of the program. Since Spanner programs are designed to be executed on each router along a path, when using the router alert IP option, the program itself can control how, when, and whether its delivery is continued to the next node. It can continue delivery along the default path, or choose which interface or interfaces to continue delivery from. A Spanner program can also choose to terminate delivery while en route.

4.3.1 *Syntax.* Spanner syntax is similar in form to assembly language syntax: a Spanner command is on a single line, and only one command per line. Each line starts with an operation and is followed by arguments. The major difference between Spanner and most assembly languages is that type information is embedded in the operations.

To give a feel for Spanner, the program in Figure 7 is the Spanner equivalent of the Sprocket program example in Figure 5.

```
        decl-addr-arr-np %addresses
                                 ; an array for getting addresses
        decl-pkt %pkt            ; declare a packet and put it on
                                 ; the stack the packet will be
                                 ; sent back to the program source
        niface                   ; put the number of interfaces
                                 ; on the stack
        papp @ &                 ; take the num IFs from the stack
                                 ; and put them in the packet (@ and
                                 ; & indicate arguments on the stack;
                                 ; @ says to remove the argument
                                 ; after the operation, & says to
                                 ; leave it on the stack)
        decl-u16 %index #1       ; declare a counter and initialize
                                 ; to 1
$loop:  lt & &                   ; test if we've looked at all IFs;
                                 ; leave both num IFs and the
                                 ; reference to the variable index
                                 ; on the stack
        brt $done                ; if we've looked at all IFs,
                                 ; branch to $done
        papp %pkt &              ; put the IF device number into
                                 ; the packet's data area
        naddr &                  ; get the number of addresses
                                 ; for the IF
        ne0 &                    ; see if there are any addresses
        brt $do_addresses        ; there are addresses, go to
                                 ; process them
        pop                      ; get rid of the number of
                                 ; addresses from stack
        bru $skip_addresses      ; skip over address processing
$do_addresses:
        sdim %addresses @        ; set array to the number of
                                 ; addresses
        gaddr & &                ; get addresses
        papp %pkt @              ; put addresses into the packet's
                                 ; data area
$skip_addresses:
        mtu &                    ; get the mtu for the interface
        papp %pkt @              ; put the mtu into the packet
        ainc-np &                ; increment %index, the IF counter
        bru $loop                ; jump back to the beginning of
                                 ; the loop
$done:  send %pkt                ; send the packet back to the
                                 ; source
        cont                     ; continue delivery of this
                                 ; program
```

Fig. 7. Example Spanner Program which encodes to 54 bytes.

## 5. VIRTUAL MACHINE

The virtual machine executes the Spanner code in Program Packets. When a Program Packet arrives at a node, the daemon authenticates the identity of the sender, verifies the data origin and data integrity, and checks if the sender is authorized to run the smart packet's program. Then an instance of the virtual machine is instantiated (literally, the daemon forks and the child process runs the virtual machine) and the code within the packet is executed.

### 5.1 Virtual Machine Issues

There are many ways to implement virtual machines [Lindholm and Yellin 1997; Lampson and Redell 1980; Redell et al. 1980]. The critical issues tend to be the richness of the virtual machines's feature set and security. These features tend to dictate the expense and complexity of the virtual machine implementation.

In the Smart Packets system, security was vital but the required feature set was relatively simple. As a result, we were able to implement a small virtual machine.

### 5.2 Virtual Machine Implementation

The Spanner virtual machine has a stack-based CISC architecture with the following components: variable sets to store local variables (and one for global variables), a stack which is used for instruction arguments and results, a condition code for reporting results of operations, and frames for executing subroutines.

For each instruction in the program, the virtual machine extracts and examines the 2-byte opcode and, based on the opcode, extracts the rest of the instruction data (variable identifiers and literals). The arguments are resolved with actual values in either the local variable set, the global variable set, or the stack. After type checking, the instruction is executed, the condition code is modified as dictated by the instruction, and, if indicated by the opcode, the result is pushed on the stack.

A variable declaration instruction is a declaration opcode (which contains the variable type) followed by a variable identifier and, optionally, by an initial value for the variable. The variable's identifier, type and value are added to a variable set. When a declaration is executed in a subroutine, then the variable is added to the subroutine frame's local variable set, otherwise it is added to the global variable set.

The virtual machine condition code is either set or cleared. Branch instructions use the condition code to decide whether or not to branch based on the state of the code. Most instructions modify the condition code. If the result of the instruction is a non-zero value, the condition code is set, otherwise it is cleared. The boolean value of the condition code can be pushed on the stack with a Spanner operation.

The stack contains references to variables and value-type pairs that result from operations. The stack is used when resolving instruction arguments and as a place to put results. Hand-tuned Spanner programs can be made significantly more compact by making use of the stack rather than referencing variables. Figure 7 is an example of a program that uses the stack to this end.

Spanner includes subroutines. When a subroutine is called, a subroutine frame is created and the stack is marked. During execution of the subroutine, stack accesses are not allowed across the stack marker. The frame contains a variable set local to the subroutine, the address to return to, the return type of the subroutine, and an indicator as to whether or not the return value should be pushed on the stack. Upon returning from the subroutine, all stack entries above the stack marker will be removed, and the subroutine frame is destroyed. When a variable is used in an instruction within a subroutine, the virtual machine first looks in the local variable set; if the variable is not found, the virtual machine looks in the global variable set.

The virtual machine evaluates conservatively. If it does not know how to handle a situation, it quits execution and sends an Error Packet back to the source of the program. Examples of error conditions are: not recognizing an opcode, insufficient entries in the stack for an instruction, receiving a return instruction when the virtual machine does not think it is in a subroutine.

As part of the security architecture, the virtual machine is made aware of resource limits such as maximum number of instructions to be executed, and how much memory can be used, and privileges such as whether access to MIB sets is allowed. These resource limits are set within the daemon upon startup, and are passed to the virtual machine when it initializes. The virtual machine counts the number of instructions executed and monitors the amount of memory the program is using, and if one of these values exceed the limits imposed by the daemon, the program will terminate and an Error Packet is sent back to the originating host.

### 5.3 Limiting The Performance Impact of Smart Packets

The Smart Packets virtual machine resides on the router's control processor, not in the router's main forwarding path [Partridge et al. 1998]. As a result, Smart Packets have no impact on the router's forwarding speed. Their only impact is on the router's ability to perform control functions.

Two features limit the impact of Smart Packets on control performance. First, most routers have operating systems that can prioritize tasks. So the Smart Packets' virtual machine can be run at a low priority, to protect more vital functions, such as routing table management. Second, Smart Packets have no persistent state. At no point are resources on the control processor being held in anticipation of receiving another smart packet. So there's no penalty for discarding a smart packet in cases of overload. (One can contrast this freedom to discard with stateful protocols, where discarding a packet may actually increase load, by causing state to linger longer).

## 6. SECURITY

The correct operation of the network requires that individual routers not be diverted from their primary task of forwarding packets. Smart Packets represents a potential threat to network operation, because smart packets contain executable code than can cause routers to malfunction in a number of ways, from consuming processing resources to changing a router's configuration.

Our security architecture uses four mechanisms to limit the potential damage that smart packets can cause. First, we limit who can produce a smart packet. Second, we provide a data origin authentication service for each smart packet, i.e., verifying the identity of the packet sender and then verifying that the sender is authorized to send smart packets. Third, we provide a data integrity service for each smart packet, i.e., verifying each received smart packet has not been changed in an unauthorized or accidental manner while en route. Fourth, we restrict certain particularly risky operations to programs sent by authorized senders.

There are two major challenges in implementing this security architecture. First, the smart packets mutate in flight. Data can be added or deleted from the baggage area of a smart packet at every hop. Second, the amount of information required to check the data authenticity and data integrity of a smart packet is large with respect to the overall packet size. If we wish to preserve the idea that each smart packet stands alone and creates no long term state in routers, we must carry this information with each Smart Packet.

## 6.1 Authentication and Authorization

The Smart Packets security architecture has two components, authentication and authorization. Using an ANEP authentication option, the packet's origin can be authenticated and the integrity of its data verified. The authentication option specifies the entity that signed the datagram, the type of signature contained, the type of certificate contained, and the lengths of the signature, the certificate, and the data in the payload field that is signed.

Authorization of the actions contained in the smart packet message is the second phase of the security architecture. There are two major elements of authorization: (1) the control of Management Information Base (MIB) access, which includes MIB views and read/write authorization, and (2) runtime environment limits in the virtual machine, such as the maximum number of packets sent per invocation, the maximum amount of memory that can be allocated, or the maximum number of instructions that can be executed per invocation.

Access to MIB information is defined in a manner complimentary to SNMPv3, reducing the management overhead of both systems. Smart Packets maps MIB access controls to those defined for SNMPv3's View-based Access Control Model (VACM) [Wihnen et al. 1998], allowing one access mechanism to define access for both smart packets and for SNMPv3 messages. The collection of management information accessible by a SNMP entity is bound to a SNMP context. Smart Packets requires its own context for proper separation of the name spaces within the SNMP agent. Within a particular SNMP context, multiple SNMP group names can be identified. MIB access rights, i.e., views available and read/write authorizations, are bound to each SNMP group name. The SNMP group name to be used for an authenticated sender, along with the values of the runtime environment limits for this sender, are set via keywords in a per-node managed database.

The user identity, obtained by the authentication component, is used to obtain the resource limits for the program in the database. The limits are set in the virtual machine, and the virtual machine enforces limits on host resources and access to privileged instructions. The remainder of this section describes how Smart Packets implements authentication.

An authenticated smart packet carries a X.509 public-key certificate identifying the sender of the smart packet. A digital signature, created with either DSA over SHA1 or RSA over MD5 (as decribed in [Schneier 1996], among others), is used to verify the data integrity of the parts of the smart packet that are not changed en route by intermediate routers, mainly the program, but also other non-changing fields that are necessary for correct interpretation of the packet. Use of a digital signature prevents the malicious tampering of these parts of the packet.

Selecting which data in the smart packet is to be covered by the authentication information is a crucial issue. The authentication information not only verifies the origin of the smart packet, but also verifies the data integrity of the packet. One choice is for the authentication data to cover only the smart packet, but doing so leaves the ANEP header unprotected by the integrity check. Covering the entire ANEP packet is not possible, because some parts of the smart packet, specifically the baggage and the packet length field in the ANEP header, can change from hop to hop, causing the cryptographic data integrity check to fail.

We use a modified authentication option in the ANEP header to carry a digital signature over the entire ANEP header and the entire smart packet, except the zeroed packet length field in the ANEP header and any baggage at the end of the smart packet. This solution trades off exposing the entire packet length against signing the packet on each hop. The smart packet header includes the original length of the packet (before the addition of any baggage) and the authentication option includes the lengths of the both the ANEP header and the original smart packet. Both sets of length fields are covered by the data carried in the authentication option. Since the program and the length fields to find it are covered, tampering with the program will cause the data integrity check to fail.

Nevertheless, this mechanism does not give full end–to–end protection of the entire smart packet. Since the baggage and ANEP packet length fields are not protected, vulnerabilities exist with using the baggage area of a smart packet. The baggage area can be rewritten or deleted, or phony baggage can be added to the packet. However, these vulnerabilities are mitigated by the fact that the smart packet is conservatively interpreted in a virtual machine. An interpretation error stops program execution and signals an error packet.

If IPsec [Kent and Atkinson 1998] tunnels exist between the routers, the smart packets are additionally protected en route on a hop by hop basis. A wire-tapping attack in this environment has a much lower probability of success.

Upon receipt of a smart packet, the following processing is performed:

(1) If an authentication option is not present in the ANEP message, the message cannot be authenticated and so it is authorized with the minimal access rights of the "anyone" entry in the authorization database, if present.

(2) If the authentication option is present, the public-key certificate is validated. The validation process requires the public key of the certificate issuer.

(3) The signature of the entire ANEP message is verified with the following exceptions: the ANEP packet length field is zeroed and the baggage area of the smart packet program is not included.

(4) If the authentication fails, the packet is discarded. If the authentication cannot be completed, e.g., because of certificate validation timeout, the packet is forwarded. If the verification succeeds, the packet proceeds to the authorization phase.

(5) The identity contained in the certificate is used to search the authorization database for access rights. The database entry identifies any restrictions or privileges that the program should have.

Programs whose authenticated sender identity does not match a specific database entry are run in a virtual machine with privileges and resources set to "anyone," the default for messages without an authentication option. We believe that this access is both useful and necessary as a smart packet can replicate unauthenticated services available today, e.g., traceroute. Yet all access may be denied at the discretion of local security policy. In this situation, the packet is still forwarded.

### 6.2 Concerns about Certificates

Implementing the services as described above raises concern on the size of the certificate in relation to the size of the entire smart packet. For example, the length of an X.509 certificate (identified for use in the ANEP specification) containing a 1 Kbit key is approximately 400 bytes. Summing the sizes of the IP, ANEP, and smart packet headers and the certificate and subtracting from the size of a maximum length IP datagram over Ethernet leaves approximately 1024 bytes for the smart packet program, any initializing arguments, and the baggage.

Another concern is the time and resources needed to validate the certificate. Many of the techniques that reduce the time needed to process the certificate require additional certificates in the message. These techniques have their limits for use with Smart Packets. For example, if three certificates are included in the authentication information, only 200 bytes are left for the program. That is enough for some useful programs, like the one in Figure 5, but is still very limiting. On the other hand, there is a tradeoff between the time and computation costs to authenticate a program performing a limited operation, e.g., tracing a route, versus the security cost of just running the program. For this reason, unauthenticated programs are allowed access, albeit in a resource limited environment.

Other possible approaches are to send the certificates ahead of the smart packet or have the certificates requested after the arrival of the smart packet. These approaches work, but have the unfortunate downside of creating cross-packet state, something we have tried to avoid. While there is state retained in the per–node database, this state is static and is maintained outside the execution environment. Furthermore, it does not require rendezvous between two or more smart packets for correct operation.

## 7. EXPERIENCE

As part of the Smart Packets effort, we installed Smart Packets on the CAIRN testbed shown in Figure 8. We then ran several small test programs through the network. A few of those programs are discussed here and compared with the existing alternative methods for getting the same information.

The program shown in Figure 5 finds all the interfaces on each router and, for each interface, extracts the interface IP address and MTU. Currently, the only way to retrieve this information from a router in a standard fashion is via SNMP. SNMP requires that, for each interface, we send two separate SNMP GET messages and get two replies, to learn the the MTU and address for the interface. The Smart Packets program requires just one Program packet and one Data packet in reply for the entire system.

Another interesting toy program is a round-trip traceroute. The classic traceroute program traces the hops in a one way path by sending a series of IP packets with progressively higher TTLs and reading the ICMP TIME_EXCEEDED response messages returned by routers along the path [Jacobson and Deering 1997]. This

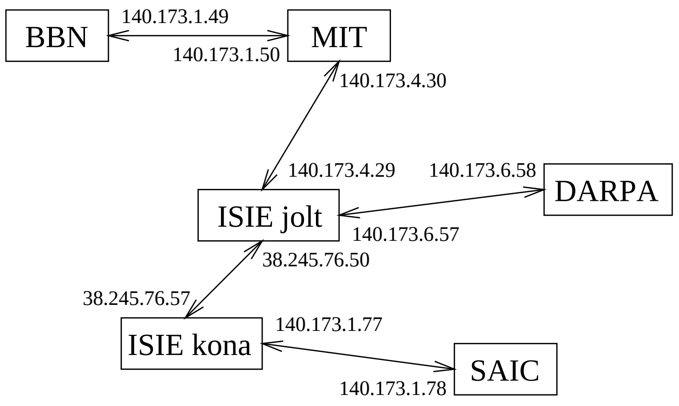

Fig. 8. The CAIRN Testbed

trace method requires two packets to be sent for each hop in the path, and requires traceroute programs at both ends of a connection to get a round-trip path.

A single Smart Packets Program Packet forwarded along the path and back can achieve the same effect. The program simply adds the address of each hop to the storage within the packet and, to protect against failure at the next hop, sends a report of the packet's progress back to the source. So, for instance, the traceroute from BBN to DARPA caused the following six Data Packets to be returned.

```
1. 140.173.1.50
2. 140.173.1.50 140.173.4.29
3. 140.173.1.50 140.173.4.29 140.173.6.58
4. 140.173.1.50 140.173.4.29 140.173.6.58 140.173.6.57
5. 140.173.1.50 140.173.4.29 140.173.6.58 140.173.6.57
    140.173.4.30
6. 140.173.1.50 140.173.4.29 140.173.6.58 140.173.6.57
    140.173.4.30 140.173.1.49
```

Traditional traceroute would have sent roughly twice as many packets.

## 8. LESSONS LEARNED

Over the course of the Smart Packets effort, we learned quite a few lessons. Several came out of our original design decisions, and others came from the nature of an active network. A few of the more general lessons are detailed below.

### 8.1 Statelessness is a two-edged sword

One of our early design decisions was to prohibit persistent execution state across packets. This eliminates the need to design and implement services to manage and check the consistency of the retained state, allowing us to greatly simplify our implementation. Also, the design of Smart Packets to operate over a connectionless transport service and to prohibit fragmentation further reduces the operating system overhead needed support Smart Packets. A potentially more important result is that since the Smart Packets program must fit inside a single datagram, a program either arrives complete or not. In situations where a network is experiencing a large number of packet drops, using connectionless transport can offer benefit over connection-based transport.

Nevertheless, not supporting persistent state affected other design decisions, such as the maximum packet size, how to support the delivery of certificates in the security architecture, and the inability to leave hints to other smart packets.

### 8.2 Compact encoding proved valuable

The design of the compact machine-independent binary encoding of Smart Packets programs helped greatly to relax the constraints imposed by the decisions to use a connectionless transport and prohibit fragmentation. The encoding allows simple programs to be tiny, while complex programs are still small enough to fit inside a single packet.

### 8.3 IP is less extensible than we believed

For active networks technology to co-exist in the current Internet, it must be able to interact with both active and non-active nodes. Using IP as the internetworking protocol under ANEP allows Smart Packets to use the existing infrastructure without needlessly replicating existing services. The IP Router Alert option allows smart packets to pass though non-active nodes without disturbing them. Only routers that support Smart Packets capture and evaluate them. Another benefit of this architecture is that smart packets can discover the Smart Packets-aware nodes along the network path, minimizing the need to design complicated routing overlays or use source routing.

During the course of the effort, we learned that many routers and firewalls are configured to drop any packet with IP options set, prohibiting the passage of smart packets. Since the IP Router Alert option is mandated with the use of other protocols (e.g., RSVP), Smart Packets is not the only service denied by these systems.

### 8.4 Security is challenging when data mutates in flight

The security goals of Smart Packets, combined with the ability of the packet to change en route and not having a priori knowledge of the evaluating nodes, makes for a challenging problem. There are well known digital signature mechanisms to protect the header and data fields of a packet, if the packet's contents do not change en route to the destination. While is it straightforward to compute a digital signature on a packet, this mechanism can not be used to provide end–to–end protection for data appended in the baggage field of a smart packet, because the baggage may be changed from hop to hop.

Our design, to provide data integrity and data origin authentication services to the smart packet, attempts to balance the need to protect the program code while allowing parts of the packet to change en route. As a result, there is not end–to–end protection of the entire smart packet.

Some programs, such as those that characterize paths or trace routes, may need to visit a node more than once for correct operation. The security goals could be interpreted to necessitate the implementation of non-windowing anti-replay methods that prohibit this behavior, thus greatly reducing the potential functionality of active code in networks.

In general, getting the security correct for active networks appears to be more difficult than getting security correct for non-active networks.

## 9. CONCLUSION

In 1987, when work on SNMP was first starting, several people, most notably Dave Mills, suggested doing useful management activities within one programmable packet. There are clear advantages to a powerful, yet constrained, programmable mechanism for network node management and control.

Active networks technology is convincingly well-suited for network management. There is (finally) enough compute power in the control side of routers and other managed nodes that the added load of executing asynchronous programs is no longer prohibitive. Polling, the current technique for network management, is like playing Twenty Questions. Placing intelligence on the node provides more efficient communication and faster discovery of targeted network events.

Implementing most of the pieces of Smart Packets, with security being the major exception, was straightforward. The compact encoding allowed us to express complex ideas in a very small space, and we believe this scheme is an improvement over the approach taken in HEMS. The language provides general purpose MIB navigation and data retrieval methods—which is what SNMP does—but Smart Packets can perform complex operations on these data at the site of the data—which is far more powerful. Finally, having control over our virtual machine architecture has made it far easier to enforce bounds and limits.

Smart Packets is designed as a network management tool. We are experimenting with integrating this technology into real network management centers responsible for operational networks. While we have laid the foundation for security, there is still work remaining in building an infrastructure compact enough to fit in smart packets, strong enough to allow remote control, and flexible enough to allow remote authorization and delegation.

## REFERENCES

Abelson, H., Sussman, G., and Sussman, J. 1985. *Structure and Interpretation of Computer Programs.* McGraw-Hill.

Alexander, D. S., Braden, B., Gunter, C. A., Jackson, A. W., Keromytis, A. D., Minden, G. J., and Wetherall, D. 1997. Active Network Encapsulation Protocol (ANEP). http://www.cis.upenn.edu/~switchware/ANEP/docs/ANEP.txt.

Arnold, K. and Gosling, J. 1997. *The Java Programming Language.* Addison-Wesley.

Bhattacharjee, S., Calvert, K. L., and Zegura, E. W. 1996. On active networking and congestion. Technical Report GIT-CC-96/02, Georgia Institute of Technology, College of Computing.

Case, J. D., Fedor, M., Schoffstall, M. L., and Davin, C. 1990. Simple network management protocol. IETF Network Working Group RFC 1157 (May).

Davin, J., Case, J. D., Fedor, M., and Schoffstall, M. L. 1987. Simple gateway monitoring protocol. IETF Network Working Group RFC 1028 (November).

Gosling, J., Joy, B., and Steele, G. L. 1996. *The Java Language Specification.* Addison-Wesley.

Hicks, M., Kakkar, P., Moore, J. T., Gunter, C. A., and Nettles, S. 1998. PLAN: A Packet Language for Active Networks. In *Proceedings of the Third ACM SIGPLAN International Conference on Functional Programming Languages* (1998), pp. 86–93. ACM.

Housley, R., Ford, W., Polk, W., and Solo, D. 1999. Internet x.509 public key infrastructure certificate and crl profile. IETF Network Working Group RFC 2459 (January).

Jacobson, V. and Deering, S. 1997. Traceroute(8). UNIX Manual Page.

Katz, D. 1997. IP Router Alert Option. RFC 2113 (February), IETF.

Kent, S. and Atkinson, R. 1998. Security architecture for the internet protocol. IETF Network Working Group RFC 2401 (November).

Lampson, B. and Redell, D. 1980. Experience with processes and monitors in mesa. *CACM 23*, 2 (April), 105–117.

Leroy, X. 1996a. A modular module system. Research report 2866 (April), INRIA.

Leroy, X. 1996b. *The Objective Caml System.* http://pauillac.inria.fr/ocaml/: INRIA.

Leroy, X. and Mauny, M. 1993. Dynamics in ML. *Journal of Functional Programming 3*, 4, 431–463.

Liang, S. 1999. *The Java Native Interface: Programmer's Guide andSpecification (Java Series).* Addison-Wesley.

Lindholm, T. and Yellin, F. 1997. *The Java Virtual Machine Specification.* Addison Wesley.

Partridge, C., Carvey, P., Burgess, E., Castineyra, I., Clarke, T., Graham, L., Hathaway, M., Herman, P., King, A., Kohalmi, S., Ma, T., McAllen, J., Mendez, T., Milliken, W., Pettyjohn, R., Rokosz, J., Seeger, J., Sollins, M., Storch, S., Tober, B., Troxel, G., Waitzman, D., and Winterble, S. 1998. A Fifty Gigabit Per Second IP Router. *IEEE/ACM Trans. on Networking.*

Partridge, C. and Jackson, A. W. 1999. IPv6 Router Alert Option. RFC 2711 (October), IETF.

Partridge, C. and Trewitt, G. 1988. The High-Level Entity Management System. *IEEE Network Magazine.*

Paxson, V. 1997a. End-to-end internet packet dynamics. *ACM Computer Communication Review 27*, 4 (October), 139–152.

Paxson, V. 1997b. End-to-end routing behavior in the internet. *IEEE/ACM Trans. on Networking 5*, 5, 601–615.

Redell, D., Dalal, Y., Horsley, T., Lauer, H., Lynch, W., McJones, P., Murray, H., and Purcell, S. C. 1980. Pilot: An operating system for a personal computer. *CACM 23*, 2 (April), 81–92.

Rose, M. T. 1994. *The Simple Book: an introducton to internet management* (2nd ed.). Prentice-Hall.

Schneier, B. 1996. *Applied Cryptography* (2nd ed.). John Wiley & Sons.

Spatscheck, O. 1997. IP signature header. DARPA Active Networks Secutity Mailing List, http://www.ittc.ukans.edu/~ansecure/.

Stallings, W. 1996. *SNMP, SNMPv2, and RMON: practical network management* (2nd ed.).

Tennenhouse, D., Smith, J., Sincoskie, D., Wetherall, D., and Minden, G. 1997. A survey of active network research. *IEEE Communications Magazine 35*, 80–86.

Tennenhouse, D. L. and Wetherall, D. J. 1996. Towards and active network architecture. *ACM Computer Communication Review 26*, 2 (April).

Trewitt, G. and Partridge, C. 1988. HEMS Monitoring and Control Language. RFC 1023 (November), IETF.

Wetherall, D. J. and Tennenhouse, D. L. 1996. The ACTIVE IP option. In *Proc. of the 7th ACM SIGOPS European Workshop* (September 1996). ACM.

Wihnen, B., Presuhn, R., and McCloghrie, K. 1998. View-based Access Control Model (VACM) for the Simple Network Management Protocol (SNMP). IETF Network Working Group RFC 2275 (January).

## ACKNOWLEDGMENTS

The authors are grateful to the following people for ideas and stimulating discussions: Matt Condell, Steve Kent, Charlie Lynn, David Mankins, Linsey O'Brien, Luis Sanchez, Robert Shirey, Greg Troxel, and John Zao. The authors also express our gratitude to the anonymous TOCS reviewers for insightful comments and constructive criticism.