PLAN: A Packet Language for Active Networks

MICHAEL W. HICKS
University of Maryland, College Park
PANKAJ KAKKAR
University of Pennsylvania
JONATHAN T. MOORE
University of Pennsylvania
CARL A. GUNTER
University of Pennsylvania
and
SCOTT M. NETTLES
The University of Texas at Austin

General Terms: Active Networks, Languages

Additional Key Words and Phrases: packet programming language

1. INTRODUCTION

Modern packet-switched networks, like the Internet, transport data in packets that consist of a header, containing control information, and a payload, containing the data itself. One way of looking at the header is as a program in a primitive "programming language" defined by the packet format specification. This program is interpreted by the protocol software in routers and end-hosts and the execution of the program causes the packet to be sent to the next router along the path to the destination. If a desired service cannot be expressed by the current protocol, then the packet format and its semantics must change. Or, using our analogy, the *programming language* and its specification must change.

In the Internet, the dominant packet 'language' is the Internet Protocol (IP) [Postel 1981b], which defines the interoperability layer of the networks that make up the internetwork. Since IP's creation, a variety of services have been proposed that would require changes to IP to support; examples include core-stateless fair queuing [Stoica et al. 1998], anycast [Katabi and Wroclawski 2000], IP traceback [Savage

Author's addresses: M. Hicks, Department of Computer Science, University of Maryland, A.V. Williams Building, College Park, MD 20742. J. Moore, P. Kakkar, C. Gunter, Department of Computer and Information Science, University of Pennsylvania, 200 South 33rd Street, Philadelphia, PA 19104. S. Nettles, The University of Texas at Austin, Electrical and Computer Engineering, 1 University Station C0803, Austin, TX 78712-0240

et al. 2000], and others. Since IP is such a widely-deployed protocol, changes to it must be deliberated and agreed upon by a standards body. Furthermore, for the changes to be of use, the new standard must be deployed in all of the routers that would use it—a daunting task. As a consequence, the introduction of new network services at this level is very slow. For example, there was a span of four or more years from the time RSVP was conceptualized [Clark et al. 1992] to the time it was deployed [Pappalardo 1996], even in a very limited manner. Further, consider that the first IPv6 request-for-comments (RFC) appeared in 1995 [Bradner and Mankin 1995], its latest RFC [Deering and Hinden 1998] is dated 1998, and yet it is *still* not widely deployed [Lawton 2001].

Another problem with IP and similar internetworking approaches involves its strategy for providing interoperability between the many possible higher-level protocols that might use IP and the many possible link technologies that might make up the individual networks connected by IP. The idea is simple, every high-level protocol must use IP and IP supports every link technology of interest. Thus, IP becomes the neck of an hourglass through which all data is funnelled. The problem is that all applications must use whatever services are provided by IP and it is impossible for applications to customize the network according to their own needs.

Active networks propose to deal with both of these problems by making the network infrastructure *programmable*. If the level of abstraction could be raised from that of the bits in IP packet headers to that of a more *general* programming language, the evolution of the network could proceed at the pace of technology, since changes would occur at the level of the program—not the programming language. Moreover, rather than treat the network as a black-box (or 'cloud') that provides only end-to-end service, an application can customize its processing *within* the network using its programming interface. For example, an application could specify its own policies for packet dropping [Bhattacharjee et al. 1996] or aggregation [Sehgal et al. 2002] when resources are overloaded.

If this idea is to be realized, a key question is: *what programming language is needed?* This paper proposes a two-level approach: the packets of the network are programmed in a new programming language *PLAN* (Packet Language for Active Networks), and these packets can make use of general-purpose *services* that reside on the nodes. That is, each packet contains a PLAN program that replaces the IP packet header and payload, and these programs, in turn, tie together service calls to create more complex network functions. Therefore, PLAN can be viewed as a network-level 'glue' language for node-resident services.

Our decision to define a new language was driven by a variety of unique requirements that are described in the next section. For instance, the need for programs that fit within individual packets suggests the use of a compact scripting language. Mobility suggests the need for a 'safe' language, while network availability demands a way to limit the resource utilization of programs. Since mobility is the main aim of the language, it is important to make its computational model fundamentally distributed, rather than provided by a special library extension. In the end, the choice was to design a new language realizing all of these goals well, but relying on programs in other languages to provide more heavyweight functions.

The rest of the paper describes the PLAN language and system, its architecture, specification, and our implementations of it. We base our discussion on PLAN 3.22,

	Packet level	Service level
Language	PLAN	flexible
Code location	in packet	on node
Expressibility	limited	general purpose
Authentication?	no	when needed

Table I. Comparison of the Packet and Service levels

our current design and implementation. We omit detailed discussion of application experience and performance. These are described in other work.

2. REQUIREMENTS AND DESIGN

We will argue that any active networking approach must balance the tensions among the following issues: *flexibility*, *safety and security*, *performance*, and *usability*. To this end, our architecture partitions the problem into two levels: the *packet language* level and the *service language* level, whose roles are summarized in Table I. PLAN, our packet language, is intended for high-level control, while most of the complex functionality resides in services (which, for maximum flexibility, can be dynamically loaded over the network). This approach allows us to draw a clean boundary between lightweight and heavyweight programmability. We now explore in more detail how our two-level architecture helps us to achieve specific design goals.

2.1 Flexibility

A central goal of any active network architecture should be to provide Internet-like service at Internet-like cost, while further providing augmented capabilities for service evolution and customization. The Internet allows any user with a network connection to have some basic services, like basic packet delivery provided by IP, basic information services like DNS, and protocols like HTTP, FTP, SMTP, and so forth. Similarly, a goal of a PLAN-based active network should be to allow any user of the network to have access to basic services; these services should naturally include some 'activeness' to allow application-specific customization.

At the same time, the programmable infrastructure should capture the non-user (control-plane) operations, like routing, network management, and error reporting. Active networks provide the opportunity to unify the variety of protocols of the IP Internet into a single programmable framework. This way, protocols like SNMP, RIP, OSPF, ICMP, and ARP would be programmed using the underlying infrastructure, rather than defined separately.

Given these goals, the key question is: how should this flexibility be provided? Our two-level architecture mirrors the model of the Internet: the packet language is responsible for basic actions concerning how data is transmitted, while the node-resident services implement the complexities of higher-level protocols. That is, PLAN should be able to express 'little' programs for network configuration and diagnostics, and to provide the distributed communication/computing glue that connects router-resident services into larger protocols. To do this, PLAN must do three things: embody a model of distributed computing; have some simple, transmissible datatypes; and perhaps most importantly, be able to cope with and recover from errors in a general way. PLAN moves us away from a world with a fixed set of operations and into one where node-resident services can be easily combined on-the-fly.

2.2 Safety and Security

The shared nature of a network (and especially the Internet) requires that security be taken very seriously. Clearly, this means that increased programmability must be added in a safe and secure manner. By safety we mean reducing the risk of mistakes or unintended behavior, and by security we mean the usual concept of protecting privacy, integrity, and availability in the face of malicious attack. These are really two sides of the same coin, since flaws in the safety of a system are often used in a deliberate fashion to undermine security.

Our security model mirrors that of the Internet: 'normal' user packets require no special privileges for transport, while other special packets could require authorization. For example, routing protocols like RIP and OSPF [Baker and Atkinson 1997; Moy 1998], or management protocols like SNMP [Galvin and McCloghrie 1993] all utilize encryption and/or authorization to prevent user tampering. Our implementation of this model is made manifest in the two levels of our architecture: PLAN packets are designed to be safe by construction, and thus usable by anyone, while individual services may require authorization. For example, a service for updating a node's routing table (used by a routing protocol or a network administrator) would require authorization.

To make PLAN safe for general use, it is functional, stateless, and strongly-typed. This means that PLAN programs are pointer-safe, and concurrently executing programs cannot interfere with one another. In addition, PLAN provides strong resource usage guarantees: all PLAN programs are guaranteed to terminate when evaluating on a single network node (as long as all called service routines terminate), and any program's traversal throughout the network is strictly bounded by a counter set at its creation. Finally, PLAN has a formal specification (presented in Section 5) to help reason about the security behavior of its packet programs.

A final design principle that was motivated by the need for safety and security is that PLAN was designed to be as simple as possible. Features were added only if we could justify them with specific packet programming examples. This contributes to safety and security because it limits the actions a user (or attacker) can take to as small a set as possible. It also makes specification and correct implementations easier, also contributing to safety and security. The tension here is that in making PLAN simple we may limit its flexibility to greatly.

2.3 Performance

To provide Internet-like service at Internet-like cost implies that we streamline common-case processing as much as possible. This was a basic motivation behind our security model: by limiting the expressibility of the packet language so that anyone could use it, we could avoid performing authorization checks on every packet, which entail the need for expensive (relative to basic switching) cryptographic authentication. Furthermore, PLAN's simplicity keeps interpretation lightweight, and thus common tasks can be done quickly. We have avoided adding heavyweight features to PLAN in the belief that such features can be accessed at the service level if they are needed.

2.4 Usability

PLAN programs execute remotely, which makes it difficult to determine the causes of unexpected behavior. Therefore, it is important to provide the PLAN programmer with *a priori* assurances about a program's behavior. We provide some guarantees as part of our design: all PLAN programs are statically typeable and are guaranteed to terminate, as mentioned above.

PLAN is based on the first-order, polymorphic lambda calculus, making it easier to specify a formal semantics, and allowing programs to be expressed succinctly. In addition, it provides error handling facilities to simplify dealing with protocol errors.

2.5 Why a New Language?

Now that the reader has a basic idea of the design goals for PLAN, we can revisit the question of why we need a new language. The need for PLAN to be very simple and lightweight, which serves all of our design goals except flexibility, provides a compelling argument that no general purpose language is really suitable. In fact, the requirement that programs need not be authenticated would seem to make it insecure to use most general-purpose languages. On the other hand, the need to tailor the language to the active networking domain eliminates existing special-purpose languages. Finally, even if we consider special-purpose languages explicitly designed for distributed computing, we find that they are targeted an environment that assumes a working network and are not well suited for use as a low-level network software implementation vehicle.

3. PLAN OVERVIEW

PLAN has four features that facilitate network programming: (1) primitives for *remote evaluation*, (2) a language-level device for encapsulating a computation, called a *chunk*, (3) means to customize a packet's routing and error handling, and (4) means to access general-purpose, node-resident services. These features, along with ones found in more typical programming languages such as data-structure and control flow constructs, facilitate programming a variety of useful networking tasks, including diagnostics like *ping* and *traceroute* for finding paths to remote machines, UDP-style data delivery, distributed routing protocol computations, and address-resolution.

However, because PLAN is designed as a replacement for IP, we must be concerned with the resource usage and security properties of PLAN programs. PLAN has two key properties concerning the resource usage of its programs: all programs are guaranteed to (1) terminate on each node at which they evaluate, and (2) visit only a fixed number of network nodes. These properties aim to ensure that PLAN programs do not make it easier to inflict denial-of-service attacks on network nodes than is already possible with IP. PLAN is also a type-safe language, meaning that its programs cannot illegally interact with the nodes on which they run, say by buffer overflows, unsafe casts from integers to pointers, *etc.*.

This section describes PLAN informally, though with care. Some formal detail can be found in Section 5, where we describe PLAN's mathematical specification, and in an appendix to the paper, where a BNF-style grammar is presented. We begin by introducing the semantics of remote evaluation in PLAN, and present a small example program that performs an active *ping*, to test whether a remote machine is accessible and available. We then explain how remote evaluation is implemented using PLAN packets, and how packets can customize their routing. Next, we describe how PLAN's resource bounding properties are made possible, and discuss how PLAN programs can handle errors. Finally, we conclude our discussion of remote evaluation by fleshing out some remaining details.

3.1 Chunks and Remote Evaluation

At the core of PLAN's design is its use of remote evaluation, implemented by the primitive `OnRemote`. `OnRemote` takes two main arguments *what* to evaluate and *where* to evaluate it. More specifically, its first argument is a *chunk*, and the second argument is a host address.

3.1.1 *Chunks.* A chunk, which stands for *c*ode *hunk*, can be thought of as a suspended function call: all of the arguments have been evaluated, but the function invocation itself has yet to occur. Chunk literals resemble regular function applications except that the function name is surrounded by bars, as in `|f|(1)`. The bars illustrate that the evaluation of the function itself is delayed. When a chunk is transmitted to a remote site, the function named in the chunk expression (*e.g.*, `f`) is invoked in a remote environment where *all* top-level bindings are available; as such it does not obey the usual lexical scoping of functions. This allows a form of recursive function call to be done with chunks, but not permitted with normal function calls; we say more on this in Section 3.3.

When a chunk is sent to a remote machine for evaluation, its code and arguments must be marshalled for transport. PLAN does not provide user-defined mutable state, so all marshalled values can simply be copied. An additional important benefit of this fact is that concurrently running PLAN programs can only share state through service routines, which implies that only the service language must concern itself with concurrency.

Chunks are first-class, meaning they may be manipulated as values. In addition to being provided to `OnRemote` for remote evaluation, a chunk may be evaluated locally by passing it to the `eval` service, which resolves the function name with the current top-level environment, performs the application, and returns the result. Chunks are quite similar to *thunks*, as provided in functional languages, but are not as general due to the changed scoping rules. A detailed discussion of many of the uses of chunks is presented in Section 4.2.

3.1.2 *OnRemote.* Remote evaluation is implemented by bundling the given chunk into a *PLAN packet* and *injecting* that packet into the network. The network routes the packet, perhaps in a customized way, to its *evaluation destination* to be evaluated. Like sending a packet, remote evaluation is asynchronous: once the packet is sent to the remote node, the current PLAN program continues without waiting for a response. Furthermore, like IP, remote evaluation is best-effort. There is no guarantee that a remote invocation will actually occur, as the packet might be dropped by the network. PLAN provides error-handling facilities to deal with these uncertainties, described in Section 3.4. We look at PLAN packets more closely in Section 3.6.

```
fun ping (source, destination) =
  if thisHostIs(destination) then
    OnRemote(|ack|(), source, _, _)
  else
    OnRemote(|ping|(source, destination), destination, _, _)

fun ack() = print("Success")
```

Fig. 1. Programming *ping* in PLAN

Using `OnRemote`, we can program *ping* in PLAN, which is used to test if a remote host in the network is both reachable and available. In the IP Internet, *ping* is provided as a special packet type in the ICMP protocol [Postel 1981a], while it can be programmed using standard facilities in PLAN, as shown in Figure 1. In the figure, the underscores indicate values that we will not consider until later.

To run the program, the `ping` function is invoked with the address of the remote host in the `destination` argument, and the address of the requesting host in the `source` argument. The program is first evaluated on the source. If the source is not also the destination, then the call to `thisHostIs` will return false, and the first `OnRemote` will remotely invoke the `ping` function at the destination, with the same arguments. There the call to `thisHostIs` returns true, so the second `OnRemote` invokes the `ack` function back on the source, which prints the message `Success`.

The *ping* program calls two functions not defined in the program text, `thisHostIs` and `print`. When a PLAN program evaluates at a node, such calls are resolved to node-resident functions called *service routines*. It is expected that all PLAN nodes will provide a set of service routines called the *core services*; these core routines include `thisHostIs`, `print`, and others like `getHostByName` to perform name lookups, or `getNeighbors` to get a list of the hosts immediately neighboring the current host. Other services may be available, but may be subject to authorization checks. We discuss services in greater depth in Section 3.5.

3.2 Routing

The ping program describes what should happen at the source and destination nodes, but not how the PLAN packet makes its way to the destination. This observation raises a larger question: *on which nodes should a packet's program be evaluated?* At one extreme, adopted by some packet programming languages [Wetherall et al. 1998; Schwartz et al. 2000; Nygren et al. 1999; Egawa et al. 2001], we could require that a packet's program evaluate on every node it traverses. Another extreme, embodied by the IP-based Internet, is that customized computation is only allowed at the endpoints. In PLAN, we take the middle ground and allow a packet to evaluate at any number of programmer-specified nodes between its source and ultimate destination. In between its evaluation points, each one indicated by the a field *evalDest* stored in the packet, the packet is *routed* by the network. More justification of this approach is made in Section 3.6.3.

Rather than leave routing entirely up to the whims of the network (as is the case with IP packets), a program can specify how it is to be routed by naming a per-packet *routing function*. This function is a node-resident service that takes as an argument a destination address, and returns back the address of the next hop on the way to that destination. When a packet arrives at a node that is not its evaluation destination, the routing function is called, and the packet is forwarded to the next hop, as returned by the function. In the case that a packet does not require customized routing, it can specify the `defaultRoute` function, available in the core service set. In our experience, `defaultRoute` is used most often, but we have used custom routing as well [Hicks et al. 1999].

Alternatively, the packet can perform its own routing, by setting the evalDest field hop by hop in the network, as described further in Section 3.6.

3.3 Resource Bounds

Because the network is a shared infrastructure, it must ensure that its packets do not consume an unfair amount of resources. In the IP-based Internet, all unicast IP packets have a time-to-live (TTL) field, a fixed maximum size, and have very simple header processing, so they satisfy the following resource usage property:

> *The amounts of bandwidth, memory, and CPU cycles that a single packet can cause to be consumed is linearly related to the initial size of the packet and to some resource bound(s) initially present in the packet.*

While this property does not prevent all forms of *denial-of-service* attack (particularly distributed denial-of-service (DDOS) attacks), it has allowed the Internet to scale effectively to hundreds of millions hosts.

If PLAN packets are to replace IP packets, it stands to reason that they should also satisfy this property, or one like it, or else enable hackers to more easily mount denial-of-service attacks. To this end, all PLAN programs satisfy two resource-usage properties. First, every PLAN program will terminate on each node on which it executes. Second, PLAN packets are limited in the number of nodes on which they may execute by a per-packet counter called the *resource bound* (or RB).

PLAN's termination guarantee arises from its simple flow of control constructs: statement sequencing, conditional execution, iteration over (bounded size) lists using the common functional programming combinator *fold*[1], and exceptions, all in the usual style. Although function calls are supported, notably absent are recursive function calls and constructs that allow unbounded iteration. The lack of recursion and unbounded iteration imply that all PLAN programs terminate on each node on which they run.

PLAN's second guarantee ensuring limited time in the network arises from a per-packet RB counter that indicates the sum total of nodes a packet or any of its progeny (created by `OnRemote`) may traverse. Such a limit becomes clear when we consider the following program:

[1]For those not familiar with functional programming, `fold` takes three arguments: a function `f` to execute for each element of the list, an initial *accumulator* `a`, and the list itself.

$$\mathrm{fold}(f, a, [b_1; b_2; \ldots; b_n])$$

has the meaning

$$f(\ldots f(f(a, b_1), b_2) \ldots, b_n).$$

```
fun ping_pong(pingHost, pongHost) =
  OnRemote (|ping_pong|(pongHost, pingHost), pongHost, _, _)
```

Unchecked, this program would bounce back and forth between `pingHost` and `pongHost` indefinitely. Instead, each node that a packet traverses decrements the packet's RB by one. In addition, when a parent packet creates a child packet, it must donate some of its resource bound to that child (as we describe below). Finally, since `eval(c)` is essentially the same as `OnRemote(c,h)` in which `h` is the local host, calling `eval` subtracts 1 from the resource bound.

Unfortunately, these two properties are not enough to satisfy the resource usage property stated above. In particular, the fact that PLAN programs must terminate does not imply they do so only using a linear amount of resources. For example, the following program runs in time exponential in its size, even though it does no allocation and does not even use iterators:

```
fun f1():unit = ()
fun f2():unit = (f1(); f1())
fun f3():unit = (f2(); f2())
fun f4():unit = (f3(); f3())

fun exponential():unit = (f4(); f4())
```

To avoid pathological programs of this sort, we impose two additional constraints:

(1) Given function f which calls functions $g_1,\ g_2,\ \ldots\ g_n$:

$$\begin{aligned} f\ \in\ & \mathsf{valid}\ \textbf{iff}\ g_1,\ g_2,\ \ldots\ g_n \in \mathsf{valid}\ \textbf{and} \\ & \mathsf{calls}(f) = 0\ \textit{or} \\ & \mathsf{calls}(g_1) + \mathsf{calls}(g_2) + \ldots + \mathsf{calls}(g_n) \le 1 \end{aligned}$$

where $\mathsf{calls}(g)$ is the number of PLAN functions called from function g.

(2) Iteration with `fold` must be limited to a constant amount (*e.g.*, lists of length 5), and use consumption of resource bound beyond that point (*i.e.* subtract 1 for every 5 elements traversed).

Some other active network systems [Nygren et al. 1999; Wetherall et al. 1998; Egawa et al. 2001; Schwartz et al. 2000] attempt to ensure the resource bounding property by imposing fixed CPU and memory counters at each node to limit evaluation resource cost. While straightforward, this method weakens *a priori* guarantees of correctness, since a program could be terminated at any time (of course, this is already somewhat the case, since `OnRemote` and the closely related `OnNeighbor` are unreliable). Furthermore, care must be exercised to perform this termination safely; Hawblitzel *et al.* [Hawblitzel et al. 1998] have shown that termination of threads in the JVM can be unsafe, and Java remains a popular language for active network implementation. Some discussion of the tradeoffs for safety relative to PLAN and other languages can be found in [Gunter 2002].

3.4 Preventing and Handling Errors

PLAN aims to simplify the process of distributed programming by both preventing many errors in its programs, and by providing useful means for handling them when

```
fun exnreport(h:host,e:exn) =
  (print("I raised "); print(e);
   print(" on "); print(h))

fun main(home:host,...) =
  try
    ...
  handle e =>
    abort(|exnreport|(hd(thisHost()),e))
```

Fig. 2. A general error-reporting mechanism

they arise. PLAN uses strong typing to rule out many errors, and its termination guarantee rules out others. Types can be inferred by the compiler using essentially the Hindley-Milner algorithm also employed by Standard ML, with some minor modifications. Although PLAN programs are mostly statically typeable, in our implementation they are dynamically checked. This unorthodox approach arose from the demands of remote programming: static typeability is a benefit to help debugging before injecting a packet into the network, while dynamic checking provides efficient safety (from the nodes' point of view) for mobile scripting code. In addition to a fairly standard set of base types, PLAN provides a homogeneous, variable-length list type and a heterogeneous, fixed-size tuple type, but no support for general recursive types, since their utility is questionable without general recursion. PLAN also supports parametric polymorphism, in the style of Standard ML, and similar to templates in C++.

Anomalous conditions are signalled to PLAN programs through exceptions. For example, `OnRemote` will raise the exception `NotEnoughRB` if the packet does not have enough resource bound to send a new packet. However, because of the asynchrony of `OnRemote`, exceptions alone are not sufficient for handling all errors. Using a synchronous `OnRemote`, a thrown exception could be propagated back to the calling program to be handled; instead, the program that called `OnRemote`, if it even still exists, is in no position to handle the exception.

Therefore, so that the application is notified when something goes wrong, PLAN provides two error handling mechanisms based on *callbacks*. First, an `abort` service is provided which allows a program to execute a chunk on its source node. This is accomplished by extracting the source address from the packet header and sending to it an error packet carrying the chunk. Once there, any remaining resource bound is discarded, and the chunk is evaluated. Coupling the `abort` service with exceptions provides for reasonably flexible error-handling; an example is shown in Figure 2.

However, evaluation on remote nodes may raise exceptions not anticipated by the programmer, and some errors are severe enough that they cannot be handled within PLAN (for example, a transmission error may result in a type-incorrect program that is rejected by the interpreter). For these cases, we provide a mechanism for error handling through a special field in the packet header. The *handler* field names a service to be invoked on the source in case an error or exception not handled in the program is raised. This essentially corresponds to an implicit call to the `abort` service where the chunk to be executed is simply a call to the named handler service.

For both of these cases, error-handling semantics dictates that the source field names the host where the packet's *oldest ancestor* was injected. For example, in the ping program, up to three packets will be created: the first by the *ping* user application, and the second two by `OnRemote` calls in the `ping` function. Because the first packet will create the second two, it is termed the *parent* of those packets. Each of the child packets will share its parent's source field. This allows child packets to report back to their originating application, and for errors to go to the right host.

3.5 Services

PLAN programs essentially 'glue' together service routines, like `thisHostIs` and `print` from the `ping` example. PLAN is lexically-scoped, with the available services occupying the initial bindings in the namespace. Because service invocations are syntactically identical to normal function invocations, a PLAN program may shadow a service routine by defining a local function of the same name. By the same token, if a name fails to resolve at invocation time, the interpreter assumes the program is attempting to invoke an unavailable service routine, and raises a `ServiceNotPresent` exception.

Since we want basic programs like `ping` to be available to all users, a number of 'core' services are made available throughout the network, including `thisHostIs` and `print`. Of course, many computations will require services that should not be available to all users. For example, we could provide a PLAN interface to the router's Management Information Base (MIB) [McCloghrie and Rose 1991] so that PLAN packets could be used for network management. A simple approach to preventing unauthorized access to protected services would be for such services to have arguments that allow whatever security credentials that are needed to be carried in the packet and the passed to the service.

In addition to the simple approach, we also provide a more powerful *namespace-based* approach to security. While the details are presented in another paper [Hicks et al. 2003], we present the basic idea here. In addition to `eval` is a related service called `authEval`, which takes as arguments a chunk, a digital signature, and a public key.[2] `authEval` verifies the signature against the binary representation of the chunk and the provided public key. If successful, the chunk is evaluated; otherwise, an exception is raised. During evaluation, the program's namespace is expanded to include bindings to services commensurate with its level of privilege of the given key. If the program tries to access a protected service for which it does not have appropriate privilege, the service will be not be in the program's namespace, resulting in `ServiceNotPresent` being thrown.

3.6 Remote Evaluation Revisited

Now that we have had a good overview of PLAN, we can refine and complete our discussion of remote evaluation. First, we present our implementation of PLAN packets, summarizing the fields as we have discussed to this point. Second, we

[2]Our implementation actually makes use of shared-key cryptography, rather than public key cryptography, requiring an initial key generation phase. Otherwise, the description here is accurate.

Field		Explanation
chunk	code	top-level functions and values
	entry point	first function to execute
	bindings	arguments for entry function
evalDest		node on which to evaluate
RB		global resource bound
routFun		routing function name
source		source node of initial packet
handler		function for error-handling

Fig. 3. The PLAN packet

present `OnRemote` in its full generality, now incorporating PLAN's notions of resource bounding and routing functions. Finally, we introduce the remote evaluation primitive `OnNeighbor`, and contrast it with `OnRemote`, presenting an alternative ping implementation that makes use of it.

3.6.1 *PLAN packets.* The PLAN packet format is shown in Figure 3. The primary element of each packet is its chunk, which consists of three components: the *code*, the *entry point*, and the *bindings*; the latter two are referred to collectively as the *invocation*. The code consists of a series of definitions that bind variables to either functions, simple values (*i.e.* integers, strings, *etc.*), or exceptions.

The invocation defines the function call (*i.e.*, function name *entry point* and actual parameters $a_1, \ldots, a_n$—the *bindings*) to be evaluated at the *evaluation destination* (or *evalDest*), which is stored in the packet. To resolve variables mentioned in the invocation, the set of all definitions in the code part and the core service functions serve as the legal environment for the call.

The remaining fields in the packet are used to support the features we have thus far described: the *RB* field stores the packet's current resource bound, and is decremented on each hop the packet traverses; the *routFun* field names the packet's routing function (*e.g.*, `defaultRoute`); the *source* field names the origination host of the packet's oldest ancestor, needed for error reporting; and the *handler* field is used when dealing with exceptions that escape the program scope.

3.6.2 `OnRemote` *revisited.* For simplicity, we have so far presented `OnRemote` as taking only two arguments. It takes two additional arguments:

(1) The amount of *resource bound* to give to the packet being sent. The amount specified must be greater than zero and no greater than the parent's current resource bound. This way, a parent packet can create multiple child packets, and give some resource bound to each. Once a packet's resource bound is exhausted, it may not create further packets.

(2) A routing function name. This is stored in the packet's routFun field and used on each intermediate hop as described above.

The program in Figure 1 can be altered to accommodate these changes by using the call `getRB()` to get the current packet's resource bound and provide it to the child, and the `defaultRoute` function for doing the routing.

A more terse (and efficient) ping program is as follows:

```
fun ping () =
  OnRemote(|print|("Success")), getSource(), getRB(), defaultRoute);
```

To use this program, the host application sets the packet's evalDest to the remote host to ping. The local host interpreter forwards the packet to the destination where it evaluates `ping`, and thus the `OnRemote` command. As a result, a new packet is sent back to the source (the source address is extracted using the service routine `getSource`) with all of the remaining resource bound (the packet's resource bound is extracted using the service routine `getRB`), using the service function `defaultRoute` for routing. When the packet arrives at the destination (*i.e.* back at the source), the `print` service will send `Success` to the host application to be printed.

3.6.3 `OnNeighbor`*: neighbor evaluation.* `OnRemote` is useful in the case that packet computation occurs only sporadically along its path. However, the packet may need to perform computation on every hop, or use arbitrary computation to calculate the hop itself. The `OnNeighbor` primitive is provided for these circumstances. `OnNeighbor` is similar to `OnRemote` except that the *evalDest* argument must be a neighbor of (that is, directly connected to) the current node, eliminating the need for routing. Therefore, rather than requiring a routing function as its fourth argument, it requires the link layer device handle on which to send the packet.

PLAN's two remote evaluation primitives, `OnRemote` and `OnNeighbor`, correspond to *network-layer* and *link-layer* packet transmission, respectively. That is, `OnRemote` can send a packet to any node in the network, requiring routing, while `OnNeighbor` can send a packet to hosts that are directly connected to it through the specified network device; no routing is performed. Using only `OnNeighbor` ensures that a packet is in complete control of its processing (*i.e.* it evaluates on every hop between the source and its ultimate destination). In fact, the semantics of `OnRemote` can be implemented using `OnNeighbor` and some additional PLAN code, as shown for `ping` from Figure 1 in Figure 4.

Here, the routing function `defaultRoute` is called directly by the PLAN program to determine the next hop. It returns a pair consisting of the host address of the next hop and the device through which to send to reach that host. Observe that our very simple ping program has expanded quite a lot to deal with hop-by-hop routing. Preventing this code blowup, as well as some performance considerations, motivated our providing `OnRemote` to complement `OnNeighbor`.

4. PROGRAMMING IN PLAN

In this section, we showcase the interesting features of PLAN with programming examples. We will see that PLAN's remote evaluation primitives allow us to make better use of network resources by providing fine control over packets' movements. Furthermore, PLAN's chunk abstractions provide mechanisms for encapsulating code and data, allowing for dynamic protocol composition and signalling.

4.1 Custom Routing

4.1.1 *Traceroute.* Traceroute is a utility for discovering a path between two nodes in the network. In the IP-based implementation of traceroute, the originating

```
fun ping_eval(dst:host) =
  if thisHostIs(dst) then (* got there *)
    let val p:(host * dev) = defaultRoute(getSource()) in
     OnNeighbor(|ack_eval|(getSource()), fst p, getRB (), snd p)
    end
  else (* not there yet *)
    let val p:(host * dev) = defaultRoute(dst) in
     OnNeighbor(|ack_eval|(dst), fst p, getRB (), snd p)
    end

fun ack_eval(src:host) =
  if thisHostIs(src) then (* got there *)
    print ("Success")
  else (* not there yet *)
    let val p:(host * dev) = defaultRoute(src) in
     OnNeighbor(|ack_eval|(src), fst p, getRB (), snd p)
    end
```

Fig. 4. PLAN ping that evaluates on all intervening nodes

host sends out ICMP ECHO packets towards the destination with successively increasing TTL values. Successive ICMP packets, in theory, time out one hop closer to the destination. When a packet times out, another ICMP error message is sent back to the source indicating a TTL expiry and the IP address of the host on which it occurred. The source simply collects these timeout messages to construct the route. If the remote host is unreachable, the path collected will be up to the point where the network has failed.

We can implement traceroute in PLAN using `OnNeighbor`; the PLAN code is shown in Figure 5. Like the standard IP version, the PLAN algorithm ensures that if the destination is unreachable due to a failure in the network, all of the nodes up to that point will be reported. In particular, at each node, `traceroute` sends back the name of the current host with the number of hops traversed so far, and creates a new packet that is sent to the next hop to continue the process.

To start, a host application injects a `traceroute` packet to evaluate at the source. At each host on which it evaluates, the `traceroute` function sends an `ack` packet[3] back to the source. This packet carries with it the host and the number of hops it is away from the source, which is then printed to the host application. In addition, if the `traceroute` packet is not yet at the destination, it determines the next hop towards the destination, and sends another `traceroute` packet there.

The execution of traceroute is depicted visually in Figure 6 between a source node A and a destination node D. Each arrow in Figure 6 represents a single packet, and is labelled with its entry point function name, where the arrowheads indicate the nodes on which the packet evaluates. Thus all `ack` packets are evaluated only at node A, the source, while the `traceroute` packets are evaluated at each node on the way to the destination D.

[3]For brevity, we shall refer to a packet whose chunk has function *foo* as its entry point as a *foo* packet.

```
fun traceroute (dst:host, count:int) =
 let val this:host = hd(thisHost())
 in
  (OnRemote(|ack|(this, count),
           getSource(), count, defaultRoute);
   if (not (thisHostIs(dst)) then
    let val p:(host * dev) = defaultRoute(dst)
    in
     OnNeighbor(|traceroute|(dst, count+1),
               fst p, getRB (), snd p)
    end
   else ())
 end

fun ack(h:host, count:int) =
 (print(h); print(" : "); print(count); print("\n"))
```

Fig. 5. PLAN *traceroute*

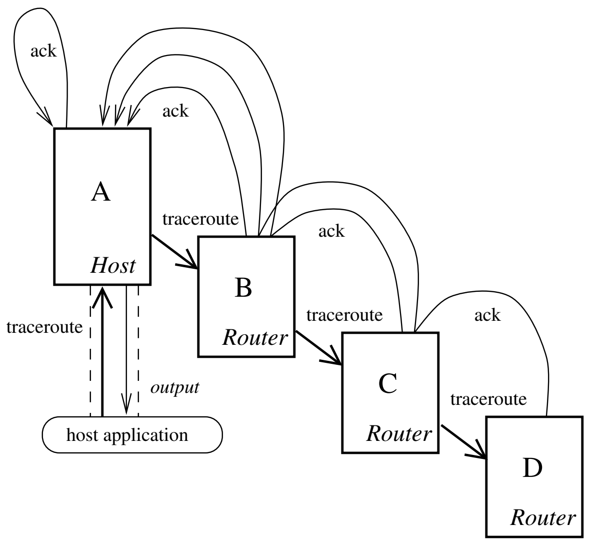


Fig. 6. Evaluation of the `traceroute` program

This example illustrates the use of pairs and lists in PLAN. Pairs are needed because `defaultRoute` returns a pair (of type `host * dev`), and we need to access its fields to provide to `OnNeighbor`.[4] Lists are used by the service function `thisHost`, which returns as a list all of the addresses that may be used to refer to the current host. Like in the IP Internet, when PLAN hosts may be *multi-homed*, meaning they have access to multiple networks, with a different address on each network.

[4]For those unfamiliar with pairs, the first field of a pair is accessed using the function `fst`, while the second field is accessed using the function `snd`.

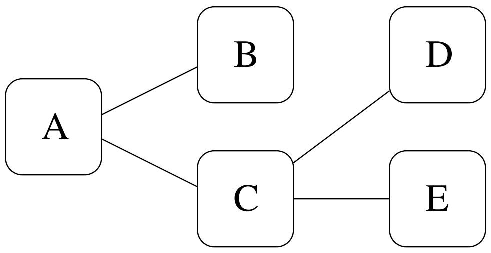


Fig. 7. A sample network topology

Now, contrast the above `traceroute` program with the standard IP traceroute mechanism. Our PLAN `traceroute` uses fewer network transmissions than this scheme, since we have only one "outgoing" traceroute packet, whereas the standard traceroute must re-traverse earlier nodes with each successive outgoing ICMP request. In addition, only having one outgoing packet means a route change in the middle of our traceroute query will not affect the accuracy of our result; the PLAN traceroute will still report hops along a coherent (albeit old) route, whereas the standard mechanism might present a mix of hops from the old and new routes.

Moreover, PLAN admits other implementations. For example, a *collectroute* program could collect the path between the source and destination and only send the result upon reaching the destination. This uses even fewer resources than PLAN's traceroute, but at the cost of failing to report a prefix of the path when their are downed nodes. This flexibility is a central motivation behind active networks: different programs (in this case, diagnostics) can be created on the fly, without standardization.

4.1.2 *Multicast.* While both ping and traceroute are diagnostic functions, PLAN can also express more general network computations, such as multicast. The idea behind multicast is to trade off computation for bandwidth. Consider the topology depicted in Figure 7. If a program at node A were to send packets individually to nodes B, C, D and E, a total of 6 transmissions would occur: $A \rightarrow B$, $A \rightarrow C$, $A \rightarrow C \rightarrow D$, and $A \rightarrow C \rightarrow E$. A multicast packet takes advantage of common prefixes among destination nodes, resulting in only 4 transmissions: $A \rightarrow B$, $A \rightarrow C$, $C \rightarrow D$, and $C \rightarrow E$. However, the reduction in messages is compensated for by additional computation as the multicast tree must be computed by the routers.

Figure 8 illustrates a PLAN program that multicasts a computation to a list of destinations. Each time `multicast` evaluates, the local function `find_hops` is called for each node in the destination list `addrs` by the primitive `foldl`. The result is a pair containing the list of next hops in the multicast tree and a list of the remaining destinations. `multicast` is then invoked remotely on each hop parameterized by the new list of destinations. Note that `find_hops` also evaluates the designated task when the function is evaluating at a destination. This is done via a call to `eval` on the chunk `task`.

Observe that each of the child packets is sent using the following code:

```
foldl(send_packs,(getRB()/length(hops),dests),hops)
```

```
fun multicast(addrs:host list,task:chunk): unit =
  let

    (* This function has two purposes:
       - if this node is a destination, perform the task and remove
         the address from the destination list
       - calculate a list of next hops to take which form the tree *)
    fun find_hops(res:(host * dev) list * host list,dest:host):
      (host * dev) list * host list =
      let val hops: (host * dev) list = fst res
          val dests: host list = snd res in
        if thisHostIs(dest) then
          (eval(task); (hops,remove(dest,dests)))
        else
          let val hop_info:host*dev = defaultRoute(dest) in
            if member(hop_info,hops) then (hops,dests)
            else (hop_info::hops,dests)
          end
      end

    (* This function is called by fold for each hop to be taken.
       It sends the multicast packet to each hop *)
    fun send_packs(params:int*host list,hop:host*dev): int*host =
      (OnNeighbor(|multicast|(snd params,task),
                  fst hop,fst params,snd hop);
       params)

    (* The list of hops and pruned destinations *)
    val hops_dests:(host*dev) list * host list =
      foldl(find_hops,([],addrs),addrs)

    val hops: (host*dev) list = fst hops_dests
    val dests: host list = snd hops_dests
    val num_hops: int = length(hops) in

      (* If we haven't reached the end of the road, send out more
         packets, else quit *)
      if num_hops > 0 then
        foldl(send_packs,(getRB()/length(hops),dests),hops)
      else
        ()
  end
```

Fig. 8. Packet-directed multicast PLAN.

Here, the amount of resource bound to give to each child packet is determined by `getRB()/length(hops)`. In effect, the assumption is that each branch of the multicast tree is roughly balanced, in that each will require equal amounts of resource bound to deliver all of the packets. Of course, this assumption is not true in general, and we may end up giving too much resource bound to one branch and not enough to another. To prevent this problem, some systems, like ANTS [Wetherall et al. 1998; Wetherall 1999], do not enforce a *conservation of resource bound*, as PLAN does, but enforce *strictly decreasing resource bound*: the resource bound of each child must be strictly less than that of the parent, but the sum of resource bound of all of the children can exceed that of the parent. This is the same approach as taken by IP multicast, and effectively solves the RB distribution problem, since each child can be given one less than the parent's bound. The drawback here is that a packet can much more easily wage a denial of service attack on the network than with PLAN.

In related work (Section 6), we discuss a different approach based on Bloom filters [Bloom 1970]. Although one could imagine applying this Bloom filter approach to PLAN, we can solve the problem of resource bound splitting by tracking the shape of the multicast tree. In a traditional multicast implementation, the multicast tree is identified by a key that indexes a hop table at each node. When a multicast packet arrives at a node, it indexes the table with its key, and forwards its payload to each hop listed in the table. In PLAN, we could implement this table using soft state, and additionally store the required RB counts in the table, along with the hops. The added requirement is that this information be kept up-to-date when adding members to the tree. In particular, when a node adds an additional hop to its table, it needs to send a message to its parent in the tree so that it can increase its RB count, which does the same until the root of the tree is reached. This approach is compatible with the standard methods for maintaining multicast trees.

4.2 Chunks

Just because packets are programs does not mean that many of the familiar features of conventional packets do not need to be supported. In particular, the desire to build networks using layering requires that PLAN programs support encapsulation, while the need to support services such as checksumming and fragmentation means that PLAN programs must sometimes be treated as data. In this section, we explore how PLAN chunks can be layered in two ways: to support the implementation of *micro-protocols* [Hutchinson and Peterson 1991] and to provide *adaptive protocols* [Feldmeier et al. 1998; van Renesse et al. 1997].

4.2.1 *Micro-protocols.* Most commonly-used protocols like TCP and IP are complex, with a variety of functionality and many options. Developing and testing such protocols can be difficult and error prone, and the resulting protocols are not particularly flexible. These problems have motivated past research on micro-protocols [O'Malley and Peterson 1992; van Renesse et al. 1997]. Each micro-protocol embodies a single function or option; more complex behavior is achieved by composing many micro-protocols.

We will present here two micro-protocols: one for a packet checksum, and one for fragmentation. These protocols will also serve to show how chunks provide some basic networking implementation techniques within the context of packets as programs.

In PLAN, micro-protocols are built by composing chunks. In general, each micro-protocol takes a chunk plus additional arguments and returns one or more new chunks which add the micro-protocol's functionality. This is analogous to encapsulation in traditional networking, where, as a packet moves down the network stack, each protocol layer encapsulates the higher-level packet while perhaps adding ad-

```
svc verifyChecksum : (blob,int) -> bool
svc evalBlob : blob -> 'a

fun unchecksum(c:blob, sum:int): unit =
  if verifyChecksum(c,sum) then
    (evalBlob(c);())
  else
    () (* drop packet *)
```

Fig. 9. Code for a checksum chunk

```
svc reassemble :
  (blob,int,bool,key) -> 'a

fun defrag(frag:blob, seqnum:int,
           morefrags:bool, session:key)
          : unit =
  (reassemble(frag, seqnum, morefrags,
              session); ())
```

Fig. 10. Code for a fragmentation chunk

ditional header information for itself. The difference is that *code* as well as data is encapsulated.

For example, suppose we had a chunk *c* to which we would like to add checksumming. We can invoke a `checksum` service on *c* which converts it into a stream of bits (i.e., a "blob" type in PLAN) via the standard PLAN marshaling system, computes a checksum *sum*, and then wraps them in a chunk with a code segment like that shown in Figure 9. When this new chunk *d* is evaluated, `unchecksum` is called using *c* and *sum* as arguments. `Unchecksum` then calls the `verifyChecksum` service to ensure that *c* still has the proper checksum, and then either evaluates *c* or aborts, as appropriate.

As another example, consider the task of fragmentation; namely, we have some chunk *c* to transmit and evaluate remotely, but it may be larger than the MTU of the intervening path. We can have a `fragment` service that takes *c* (and the MTU size, as determined in advance, and returned by a `pathMTU` service), represents it as a "blob," and divides it into MTU-sized[5] pieces. Each fragment is then wrapped in a chunk with a code segment as shown in Figure 10. The bindings for these new chunks would each have a piece of the original chunk, a sequence number, an indication whether it was the last fragment or not, and a unique identifier. The new chunks, when evaluated, simply register themselves with the `reassemble` service on the destination. This service collects all the incoming fragments, puts them in the proper order, reconstitutes the original chunk, and then evaluates it.

[5]Less the overhead for the reassembly chunk.

```
svc defaultRoute : host -> host * dev

fun send_frag (x:int*host,c:chunk)
              : int * host =
  (OnRemote(c,snd x,fst x,defaultRoute);
   x)

svc checksum : chunk -> chunk
svc fragment : (chunk,int) -> chunk list
svc pathMTU : host -> int
svc length : 'a list -> int
svc getRB : void -> int

fun udp_deliver (b:blob, p:port,
                 dest:host) : unit =
  let val c:chunk = |deliver|(p,b)
      val d:chunk = checksum(c)
      val ds:chunk list = fragment(d,pathMTU(dest))
      val l:int = length(ds) in
   (foldl(send_frag, (getRB()/l,dest), ds); ())
  end
```

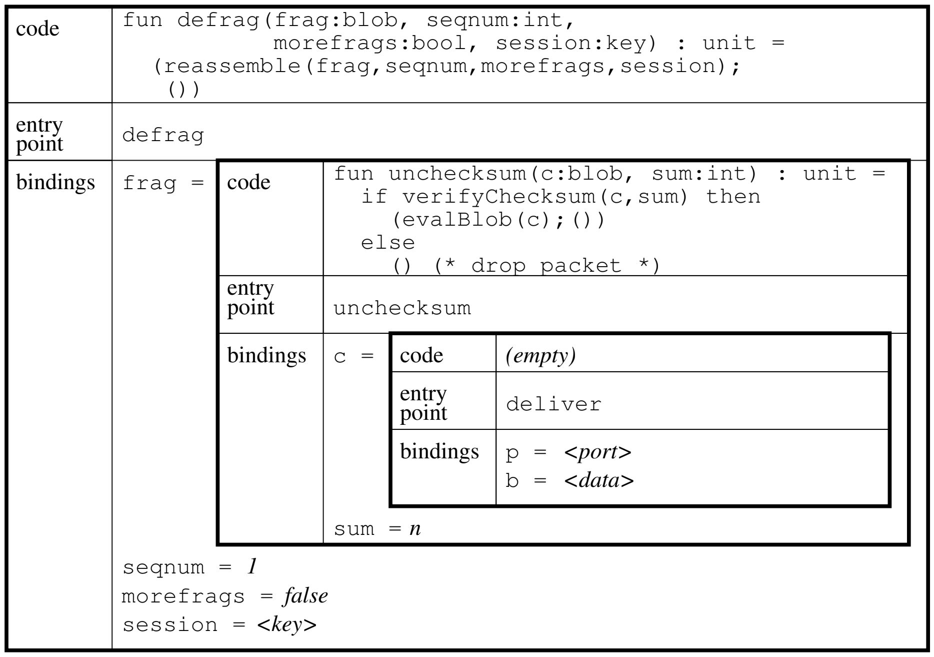
Fig. 11. UDP-style delivery

Figure 11 shows the composition of our two protocols to form a UDP-like delivery service. At the highest level, we have some data (represented as a `blob`) which we want to deliver to a specific port on a specific host. First we create the chunk c, which encapsulates this behavior. Then we create a new chunk d which adds checksumming. After querying the appropriate path MTU, we then invoke the `fragment` service to get a list of fragmentation chunks. If the original chunk was small enough to fit within one link-layer frame, these chunks take the form shown in Figure 12[6]. Finally, we use `foldl` to apply `send_frag` to send each fragment to *dest*.

As the fragments arrive they will be evaluated causing them to be placed in the reassembly table. Once the table is complete, they will be reassembled into chunk d, which when evaluated will verify the checksum, resulting in chunk c. When c is evaluated it will call `deliver` with port argument p causing data argument d to be delivered to the correct port.

Both of the above micro-protocols share the same basic structure: a service on the source is invoked with a chunk plus some configuration parameters. This results in the creation of a new chunk which carries the code to perform the destination side of the micro-protocol; note that this code may refer to services that reside on the destination (and need not necessarily reside on the source). The new chunk could then potentially be wrapped in yet another micro-protocol or simply sent across the network. At the destination, the chunks simply "unwrap" themselves.

[6]If fragmentation was actually required, we would have only a part of the innermost two chunks; however, for ease of illustration, we have not shown this case.



code

```
fun defrag(frag:blob, seqnum:int,
           morefrags:bool, session:key) : unit =
  (reassemble(frag,seqnum,morefrags,session);
   ())
```

entry point

```
defrag
```

bindings

frag =

code

```
fun unchecksum(c:blob, sum:int) : unit =
  if verifyChecksum(c,sum) then
    (evalBlob(c);())
  else
    () (* drop packet *)
```

entry point

```
unchecksum
```

bindings

c =

code	*(empty)*
entry point	`deliver`
bindings	`p` = *<port>*
	`b` = *<data>*

`sum` = *n*

`seqnum` = *1*
`morefrags` = *false*
`session` = *<key>*

Fig. 12. Chunk encapsulation

This common structure makes many things easy. For one, we can have dynamic, per-application policy drive the composition of micro-protocols, rather than having dependencies built into complex protocols. Indeed, in our above example, rather than have fragments of a checksummed delivery packet, we could have invoked *fragment* first, and then done a `checksum` on each resulting chunk, thus ending up with checksummed fragments of the original chunk. Each micro-protocol takes a chunk and returns a chunk or list of chunks, so they may be arbitrarily ordered in a type-correct way. Of course, the order *does* matter from a semantic point of view.

Second, micro-protocols can be coded to avoid redundant functionality. For example, if a path only has Ethernet interfaces, the `checksum` service might simply return the original packet, as the checksum would be redundant with the underlying CRC check. Similarly, the `fragment` service can (and does) just return the original chunk if it was already small enough. Either of these optimizations remove the need to execute certain receiving code at the destination.

In fact, the destination will not even have to do a test to determine that it need not execute the receiving code. Since the demultiplexing path is encoded in the way the chunks are encapsulated, the unnecessary code will simply not be called as an arriving chunk "unwraps" itself. This mechanism is in fact quite powerful and allows us to do straightforward asynchronous protocol adaptation, as we see in the next section.

4.2.2 *Asynchronous Adaptation. Adaptive* protocols are ones that can be dynamically reconfigured. In particular, they can react to changing network conditions to improve performance. For example, if a data stream is bottlenecked due to a low bandwidth link, it might be desirable to compress the stream. Similarly, many checksum errors arising from a noisy link might suggest using an error correction scheme to introduce redundancy.

In most approaches to adaptive protocols, a primary problem is synchronization. Namely, a source and a destination must agree on the structure of the protocol stack they are using: a protocol where the sender encrypts data but the receiver fails to decrypt it would hardly be useful. As described in [van Renesse et al. 1997], such signaling protocols can often be complex (and sometimes expensive).

With PLAN chunks, there is no need for negotiation between the endpoints for correct functionality. A sender need only start using a new sequence of encapsulated chunks, and they will be correctly handled at the receiver because the structure of the "protocol stack" is *encoded in the packets themselves*. There need be no delay for the protocol switch to happen safely. Indeed, this adaptation is independent of the underlying routing infrastructure; there is no need for nodes to maintain peering protocol agreements with each other, or to handle "handoffs" when a route changes. Of course, it may be still be important for a sender and receiver to exchange information to maintain an accurate network view so that a policy regarding the insertion and removal of micro-protocols may be reasonably applied, but this need not be synchronized with the actual protocol switchover.

Furthermore, adaptation with PLAN chunks is not limited to endpoints. It is straightforward to add micro-protocols just over some portion of the network infrastructure, as in the style of Protocol Boosters [Feldmeier et al. 1998]. In this case, a router might intercept incoming PLAN packets, wrap their top-level chunks

```
fun vpnEnter(c:chunk,dest:host) =
  let val afterTun:chunk = |fwd|(c,dest)
  let val peer:host = vpnRoute(dest)
  let val ec:chunk = encryptForPeer(afterTun,peer)
  let val sig:blob = signForPeer(ec,peer)
  let val id:blob = idForPeer(peer)
  in
    OnRemote(|authEval|(ec,sig,id),peer,
             getRB(),defaultRoute)
  end

fun fwd(c:chunk,dest:host) =
  OnRemote(c,dest,getRB(),defaultRoute)
```

Fig. 13. PLAN VPN code for tunnel entrance

in a new "boosting" micro-protocol, and send them on to a "de-boosting" location. Once there, the wrapper chunk will perform the receive-side of the micro-protocol and then send the original top-level chunk on to its final destination. Conventional, non-active packets can be treated the same way, essentially letting them tunnel in an active packet to allow dynamic protocol composition.

We make these ideas more concrete by presenting two examples. Both involve micro-protocols which are applied at points within the network rather than just at the endpoints of a communication.

4.2.2.1 *Virtual private networks.* Our first scenario considers a virtual private network, in which we have several networks of trusted nodes that we wish to connect by traversing untrusted links. We would like to give all the end hosts the illusion of being within a single trusted network. This can be accomplished in a straightforward manner by encrypting and encapsulating packets between trusted networks. In the Internet, IPSec [Atkinson 1995] may be used in exactly this way.

We can achieve a similarly elegant implementation using PLAN chunks, as shown in Figure 13. An end host transmits an unencrypted packet which is intercepted by a firewall machine when it is about to leave its trusted network. The firewall extracts the top level chunk `c` and final destination `dest` from the packet, and then runs the algorithm shown in `vpnEnter`, building up a series of encapsulated chunks. This process is implemented (and best explained) by working backwards.

After the packet exits the VPN tunnel, we want to restore the original packet's intention to evaluate `c` on `dest`; thus we create the `afterTun` chunk to do exactly that. Now, the next step is to encrypt our new chunk for the tunnel; first we figure out where the tunnel exit is by calling the `vpnRoute` service to tell us the corresponding gateway for our ultimate destination `dest`. We assume that the VPN gateways will maintain pairwise shared secret keys (obtained via a mutual-authentication protocol such as described in [Hicks et al. 2003]) for efficiency, so we use the `encryptForPeer` service to encrypt our `afterTun` chunk for transmission.

The resulting chunk, `ec`, is set to call the `vpnExit` function (shown in Figure 14)

```
fun vpnExit(enc:blob,peer:host) =
  let val c:chunk = decryptFromPeer(enc,peer)
  in
    eval(c)
  end
```

Fig. 14. Chunk executed after authEval on VPN tunnel exit

to arrange for decryption on the tunnel exit. This function, given an encrypted chunk (represented as a generic *blob* bytearray) and an originating endpoint, asks the current node to decode and evaluate the encrypted chunk. Of course, the tunnel exit will want to verify that the said chunk actually came from a participant in the VPN; *i.e.* only VPN participants should be able to invoke the `decryptFromPeer` service. To ensure this, we use the `signForPeer` and `idForPeer` services to digitally sign `ec`. We finally evaluate an `authEval` chunk on the tunnel endpoint with `OnRemote`.

When the chunk arrives at the endpoint to be evaluated, this stack of chunks "unwinds." First, `authEval` authenticates the signed chunk against the provided signature and evaluates the chunk contents. Assuming this succeeds, the chunk will be allowed to access privileged services, like `decryptFromPeer`, that would not otherwise be available. Next, the chunk calls `vpnExit`, which decrypts the encrypted chunk (producing `afterTun` again) and evaluates it. `AfterTun` then calls `fwd` which sends the original toplevel chunk `c` along on its way to `dest`.

4.2.2.2 *Mobile computing.* Our second scenario considers mobile computing over wireless links, which are more noisy than wire-based LANs. Wireless links often have poor TCP throughput, as negative acknowledgments due to checksum failures are interpreted (incorrectly) as network congestion. To compensate for packet errors, the networking software on the laptop could engage a forward error correction (FEC) micro-protocol when operating in mobile mode. While FEC often operates at the physical or link layer, it can be profitable to use FEC at the network layer [Rizzo ; Hadzic et al. 1998], as we propose to do here.

With PLAN chunks, we can easily limit the FEC just to the wireless link, thus conserving overall bandwidth in the rest of the (less lossy) network. On the source, we wrap our original chunk and its intended destination in a wrapper chunk which registers itself with the FEC service on the other side of the link. This service would check the encapsulated chunk for errors; if none are present, it can be unwrapped and forwarded onwards. We would periodically generate an additional parity packet which also registers itself with the FEC service. In turn, when the FEC service finds errors, it would attempt to apply received parity packets to correct the errors. Any original chunks that can be corrected are unwrapped and forwarded onwards. If an error corrupts the encapsulating packet (i.e. the one that registers with the FEC service), then that packet will fail to decode and must be dropped; this would be no different if a network-layer protocol like IP were used to encapsulate user data.

One issue is that the laptop might cross a cell boundary, thus switching gateways.

Normally, this would require some amount of synchronization and communication, but since the FEC chunks are carried with the packets, the new gateway immediately knows that forward error correction is being used by the laptop. At worst, the laptop may have to retransmit the batch of packets which were being transmitted when the switch occurred. Here, actively specifying the required processing within the PLAN packets themselves saves us additional communication over the lossy link.

5. SPECIFICATION AND IMPLEMENTATION

PLAN has a formal specification and several implementations. This section discusses some of the challenges in the specification and some characteristics of the implementations in OCaml and Java.

5.1 Specification

Two features of the PLAN language have an interesting impact on its specification. First, a language that is almost exclusively intended for writing mobile programs places special emphasis on issues of trust and distributed evaluation. These impact the determinism of the specification and the role of types. Second, the fact that PLAN is a 'glue' language over a general purpose layer of services makes reasoning about guarantees about PLAN more complex. This raises new challenges for reasoning about PLAN relative to functions in its service layer. Let us consider each of these issues in more detail. The specification of PLAN per se can be found at `http://www.cis.upenn.edu/~switchware/` and a discussion of the semantics can be found in [Kakkar et al. 1999].

5.1.1 *Types.* Consider a basic programming task in which data is read from an input file, analyzed, and the outcome is written to an output file. If this program fails in the middle of its evaluation, then the output file may be left in an incomplete state. To help reduce the likelihood of this problem, we can provide a static or dynamic type system for the language that may catch the cause of the failure at the point the program is compiled or perhaps before it has begun an incorrect modification of the output file. In either case, it is helpful to have a semantic specification of the language that can indicate the range of possible actions that could occur in a program with a type error. If the language provides static analysis, then this answer is easy since the error will be detected before running the program. If it uses dynamic type checking, then the answer is harder, but typically can be seen in tracing the program to the point at which it had the type error. Now, suppose we are given a program that makes a remote call. In a statically typed language we can check the remote code and prevent type errors in advance of the call.

However, reasoning about the effect of a dynamic error is significantly more complicated if the remote call can evaluate concurrently with the calling program, and the two processes communicate with one another. This overall problem becomes more challenging in a mobile language like PLAN where programs principally evaluate by splitting into collections of remote evaluations on nodes that cannot be predicted statically from the spawning programs. All of this suggests that the simplicity of static type checking would greatly aid reasoning about PLAN programs.

However, a semantics that relies on static typechecking is problematic for PLAN. First, a node executing a PLAN program needs to develop its own trust in the PLAN program and will therefore carry out checks necessary to provide adequate security for its execution environment. Second, static typechecking of chunks is not always possible. Third, static checking of whole PLAN programs at intermediary nodes may be undesirable for performance reasons. For example, a value may be placed in a chunk or conditional branch may be intended for evaluation only on a destination node and not on intermediary nodes. Thus static evaluation would cause many unnecessary checks on intermediate nodes.

These design considerations lead to two characteristics of the PLAN specification. First, it should be possible for a conformant implementation to exploit any of a gradation of typechecking options. Second, it should be possible to take a program and determine its entire range of potential behaviors in a PLAN network. Concerning the first point, active nodes have an entire spectrum of possibilities vis-a-vis type checking available to them, with static and dynamic type checking being the two extremes of this range. Routers could, for example, start the execution of the program with dynamic checking and statically type-check it at the same time, terminating the execution if a type error is found either dynamically or in the static check. Alternatively, one could statically check selected *fragments* of the program that looked like they would benefit from static analysis and check other parts dynamically. This is similar to optimizations for dynamically typed languages that omit runtime checks for parts of the program known to be type correct after an initial analysis phase. Concerning the second point, this flexibility raises concerns for the predictability of the behavior of PLAN programs. Consider, for instance, the following program:

```
fun foo () : unit =
    (OnRemote (|foo| (), host1, ...);
     OnRemote (|foo| (), host1, ...);
     OnRemote (|foo| (), host1, ...);
     1 + true)
```

This program first sends three PLAN packets to `host1`, and each will try to execute `foo ()` after getting there. Then it tries to add the integer `1` to the boolean value `true`, generating a type error. Static type checking will catch this error before execution while dynamic type checking will catch this error only after the three packets have been sent. Allowing for a range of alternative modes of type-checking means that any of 0, 1, 2 or 3 packets could have been sent to `host1`. We are not aware of another langauge and typing system that allows this kind of flexibility. For instance, a statically typed language would reject the program without evaluating it, therefore sending 0 packets, whereas a dynamically typed language would send three packets and then fail with a type error.

The PLAN semantics is therefore designed to enable reasoning about all possible type error points. This is done by the use of a 'small step' semantics. Programs with type errors may validly terminate at any point when the type error is recognized, even if it is not the current evaluation point. However, an evaluation point that contains a type error will cause termination of the program with an error.

5.1.2 *Reasoning with Services.* All widely-used general-purpose languages offer some capability for library extensions. When these libraries are written in the langauge itself using only its basic standardized constructs then properties can be proved by study of the language specification alone. However, if the library contains functions whose semantics is outside of the language, then reasoning about these requires supplementary semantic information. For example, most languages have I/O primitives whose semantics is at least partially dependent on the system for which the language is compiled. Many other operations may exist, such as functions to control new hardware, whose meaning is not feasible to include in the language definition itself. Scripting languages take this to a new level, however, since it is not unusual for *most* of their operations to lie in this class of external calls. For example, a shell language that executes each of three programs in some sequence with various parameters will have only a small part of its meaning explained by the semantic specification of the shell language. PLAN is like a shell language because of its fundamental reliance on a service layer. This has a profound impact on reasoning about PLAN programs, basically saying that properties of 'pure' PLAN will be comparatively easy to obtain, but somewhat limited to use. One can therefore expect reasoning about PLAN to be a *relative* kind of reasoning.

As an example of this phenomena, consider the challenge of reasoning about whether a network of PLAN nodes would preserve basic information flow properties on which autonomous systems connected to the Internet often rely. Suppose, for instance, that users of a network indulge in the use of cleartext password protocols on a broadcast wireline LAN. These passwords are quite vulnerable to sniffing by insiders attached to the LAN, but they are not especially vulnerable to the wider network. Let us call this 'security Against Outsiders' or *AO security* for short. AO security broadly means that someone who can sniff a password is someone 'you can fire', that is, someone who can be made accountable. This kind of security is common in protected domains, such as companies operating behind firewalls. Of course, cleartext passwords are discouraged and security sensitive organizations and users avoid them, but more sophisticated attacks like traffic analysis display the same AO security characteristics. In general, routers do not offer ordinary users any capabilities that would allow them to steal passwords or do traffic analysis.

But suppose the routers at an institution are running a PLAN network—do the AO security properties change? A simple answer can be given in a simple case: if there are no PLAN services other than the standard ones, then there is no risk of monitoring that is added by the routers. Pure PLAN programs even from the same user do not 'communicate' with one another except at endpoints. However, many PLAN programs rely on services for which interference is less obvious. For example, a program that leaves a routing label for subsequent packets implicitly communicates with these subsequent packets. In particular, an outsider could use the labels to direct legitimate packets off of the network. The problem is non-trivial to reason about. If the attacker cannot find the labels (for instance, they may be large random numbers), then they do not aid him, but if the labels are sent by packets that are themselves subject to diversion then the labels are vulnerable. A detailed discussion of AO security guarantees and how to reason about them relative to various PLAN service layer functions can be found in [Kakkar et al. 2000]. This work relies on an abstraction of the PLAN specification called uPLAN, a theoretical calculus based on PLAN. Transferring results from uPLAN to PLAN is still a challenge, but it is similar to that faced by other studies that rely on language abstractions to control the complexity of real languages.

5.2 Implementation

When choosing an implementation language for PLAN, we had several specific requirements. First, to make the claim that the network is programmable, services must be dynamically loadable. This means that our implementation language must allow some form of dynamic code loading. Second, the heterogeneous nature of an internetwork means that the implementation language should be easily portable. Third, our implementation language needed to provide strong typing for safety. We have completed implementations of PLAN in two languages that meet these requirements: OCaml [Leroy et al. 1999; Leroy 2001] and the Pizza [Odersky and Wadler 1997] extension to Java [Gosling et al. 1996]. Our most current implementation is in OCaml, due to better performance and ease-of-use. A partial/experimental version is part of the Mobile Active Network Environment (MANE) at The University of Texas at Austin [Song et al. 2002]. It is based on a experimental language, Popcorn, which compiles to Typed Assembly Language [Morrisett et al. 1999; Morrisett et al. 1999] and supports dynamic updating [Hicks et al. 2001b].

We currently transmit abstract syntax trees in our packets, and use an RPC-style marshalling scheme for the arguments to the invocation function. This same marshalling scheme could be extended to allow nodes to offer services from different languages. However, our services are currently implemented in the same language as the PLAN interpreter, so service calls are simply function calls within the interpreter.

New services may be dynamically installed over the network by having PLAN programs pass bytecodes as arguments to special service installation routines. In principle, though, services could be transmitted in various forms (such as source code) and installed via compilation, perhaps taking advantage of run-time code generation.

PLAN has been taught in both a graduate-level network primer course and an Active Networking seminar at the University of Pennsylvania, where students were asked to use PLAN to implement useful network services on a small testbed network of five nodes. Feedback from the students on the PLAN system was encouraging. One common comment was on the ease of dynamically installing services written in Java (Pizza was the main implementation language at the time), thus validating our initial design decision of following a two-level approach.

More details about the OCaml implementation of PLAN and Query Certificate Manager [Gunter and Jim 2000] (a trust management system used with PLAN in [Hicks et al. 2003]) can be found in [Alexander et al. 1998].

6. RELATED WORK

Postscript [Systems 1985] and Java [Gosling et al. 1996] are the most well known examples of using programmability and mobile code to increase the flexibility of a system. The first application of programmable network routing may be the Softnet [Zander and Forchheimer 1983] system, which provided for the execution of packets of multi-threaded M-FORTH code. Numerous other motivations for the advent of active networks are described in [Tennenhouse et al. 1997]. A more recent survey of active networking technology can be found in [Psounis 1999].

The fundamental idea of Active Packets first appeared in Wall [Wall 1982]. In this paper, Wall outlined a new approach to networking. Quoting from the paper's abstract:

> "*Network algorithms are usually stated from the viewpoint of the network nodes, but they can often be stated more clearly from the viewpoint of an active message, a process that intentionally moves from node to node.*"

Although PLAN differs in many important particulars from Wall's vision, perhaps the most significant advance for packet programming over Wall's work is the introduction of chunks.

Since the initiation of DARPA's Active Networking program there have been many Active Packet systems proposed, including the Active Network Transfer System (ANTS) [Wetherall et al. 1998; Wetherall 1999], Smart Packets [Schwartz et al. 1999], ALIEN [Alexander and Smith 1999; Alexander 1998], and SNAP [Moore et al. 2001; Moore 2002]. Below we briefly discuss some of these systems principally in comparison to PLAN. We also discuss an additional approach to resource use bounding and some work related to dynamic protocol stack modification.

6.1 Other Active Packet Systems

6.1.1 *ANTS.* The Active Network Transport System (ANTS) [Wetherall et al. 1998] was one of the first active packet systems developed. In an ANTS capsule (ANTS terminology for an AP), programs are written in a restricted subset of Java and are transported as Java bytecodes. Although ANTS differs from PLAN in key ways, what is striking is way that they are under the surface solving the same key problems. For a detailed comparison see [Hicks et al. 2002].

A novel aspect of ANTS is that capsule programs are transported by reference. Capsule programs are cached at nodes and if a referenced program is not node resident, it is fetched from the node that sent the capsule. This is advantageous when many capsules use the same program and is important because the overhead of using Java bytecodes both in terms of size and processing power for linking, verification, and perhaps JIT compilation is significant. PLAN could support such an approach without changes in the language it would have less of an impact because PLAN programs are both more succinct and we have control over intermediate representations and runtime costs

PAN [Nygren et al. 1999; Nygren 1998] is a follow-on project to ANTS, also developed at MIT. The main question Nygren *et al.* address is: are the computational overheads of providing active processing too high to ever achieve practical performance? Fortunately for us, the answer was "no." PAN achieves its high performance through the use of in-kernel packet execution, code caching, and standard network performance tuning such as minimizing data copies. A key component of PAN's performance lies in its use of native x86 code as the intermediate representation for APs. Although this results in high performance, it also sacrifices safety, especially since the x86 code is dynamically linked into the kernel.

6.1.2 *Smart Packets.* The Smart Packets project [Schwartz et al. 1999] from BBN targets network management tasks. Because Smart Packets are meant to be deployed in potentially misconfigured or failed networks, they must be extremely robust. In particular, they have been designed to be self-contained, so that no new router state is required. Furthermore, useful programs should be encodable within a single link layer frame so that fragmentation may be avoided. Smart Packets are coded using two equivalent languages: Sprocket is a safe subset of C extended with primitives for MIB access; Sprocket, in turn, may be compiled to Spanner, a compactly-representable CISC-like assembly language. Sprocket, like PLAN, provides for resource control, although it uses both hop and instruction counts.

6.1.3 *ALIEN and SANE.* ALIEN [Alexander and Smith 1999; Alexander 1998] is an active networking architecture also developed at the University of Pennsylvania. SANE (Secure Active Network Environment) [Alexander et al. 1998] is a specific instance of the ALIEN architecture. Like PLAN, ALIEN supports both APs and Active Extensions. In fact, the PLANet implementation of PLAN [Hicks et al. 1999] uses the same Active Extension system as SANE.

A crucial difference between ALIEN and PLAN is that ALIEN uses packets written in a general purpose language, specifically for SANE, OCaml [Leroy et al. 1999], a dialect of ML supporting dynamic linking of bytecodes. Although OCaml bytecodes are typesafe, the use of an unrestricted general purpose language for APs means that security must be enforced outside the language. SANE uses public key cryptography to establish security associations between neighboring nodes. Using these associations, it is possible to guarantee the origin and integrity of the OCaml bytecodes. Unfortunately the need to perform cryptographic security for each packet results in unacceptably low performance [Alexander and Smith 1999; Alexander 1998]. This result indicates that the PLAN approach, which avoids such security measures unless needed is the preferred approach.

6.1.4 *PLAN-P.* PLAN-P [Thibault et al. 1998] is not an AP system, but rather is a modification of PLAN to support programming services rather than packets. The PLAN-P work focuses on studying the use of optimization techniques based on partial evaluation to provide fast implementations of these service routines.

6.1.5 *SNAP.* Safe and Nimble Active Packets (SNAP) [Moore et al. 2001; Moore 2002] is the first second-generation AP system, drawing from most of the systems mentioned above. However, not surprisingly since its principle designer (Jon Moore) is one of PLAN's designers, SNAP is most directly an evolution of PLAN. Along a number of dimensions, SNAP takes the same approach as PLAN. For example, it is a safe, restricted domain-specific language that shares PLAN's division between AP programs and services.

However, SNAP improves on PLAN (and other AP systems) in some significant ways. First, it is designed to be high-performance without sacrificing safety. It does this in part by carefully avoiding aspects of PLAN that proved costly, such as marshalling and garbage collection. It also uses a much lower-level representation than PLAN, being a byte-code language. The result is that SNAP's LINUX implementation has performance quite close to that of IP packets. Like PLAN, SNAP provides a guarantee of resource safety. SNAP programs run in time, space, and bandwdith that is linear in the length of the packet. However, it achieves this bound by a simple low-level restriction: all branching instructions must have forward targets. While this is a significant restriction, we have found that the many useful PLAN programs can be compiled to SNAP programs [Hicks et al. 2001a], with corresponding performance benefits. An active network with PLAN as its high-level packet language, but SNAP as its low-level wire-format, would have improve the performance of PLAN, and the usability of SNAP.

6.2 Dynamic Protocol Stack Modification

Chunks are one of PLAN's most novel feature and they provide an elegant and lightweight method to modify protocols stacks on-the-fly. Ensemble [van Renesse et al. 1997] is a toolkit for distributed application development in which applications may adapt and dynamically reconfigure their protocol stacks. However, Ensemble uses a Protocol Switch Protocol that halts communication, synchronizes through a central coordinating participant, and then resumes communication. PLAN chunks do not require this pause for synchronization and do not need centralized coordination. Furthermore, PLAN chunks are not limited to an end-host only regime of operation.

Protocol Boosters [Feldmeier et al. 1998; Mallet et al. 1997] interpose additional functionality within the network infrastructure. These boosters enhance performance in a way that is transparent to the applications communicating across the "boosted" subnets. However, for multi-component boosters, signaling is required to support the addition or removal of a booster. Finally, because they reside in the network infrastructure itself, some boosters are subject to failures due to routing changes sending boosted packets around their intended de-boosting element. PLAN chunks are not subject to these failures because the chunk encapsulation essentially records which micro-protocols have been applied and must be undone at the destination.

6.3 Resource Use Bounding

Although PLAN, like most AP systems uses a combination of bounding resource use on each node and hop-count limits, another approach, used by the Icarus system [Whitaker and Wetherall 2002], relies on carrying a Bloom filter [Bloom 1970] in a packet. Each network link has its own bitmask that it bitwise ORs into the bloom filter; if the filter is unchanged, the network assumes it has already seen this packet and drops it. This approach combines the programming convenience of strictly decreasing resource bounds while preventing classes of denial of service attacks by ensuring loop freedom. However, the false positive rate of this approach depends heavily upon the relative sizes of the Bloom filter, the link bitmasks, and the number of network links, and it is as yet unclear how to appropriately set these constants.

7. CONCLUSIONS

PLAN's design began with a few basic ideas all motivated by the need to tradeoff flexibility, safety and security, performance, and usability:

(1) That a special purpose domain specific language could achieve safety and security by limiting its expressibility and yet remain flexible enough to allow a wide variety of useful packet programs to be written.

(2) That a two-level architecture in which PLAN served primarily to glue together functionality residing at the service layer would prove to both provide a security model that was lightweight and yet secure when needed and a model of composition that made node-resident services significantly more flexible.

(3) That a low-level distributed systems programming language based around unreliable remote evaluation would prove a good programmable packet analog to the usual functionality of static packets.

We believe that these initial elements of PLAN were fundamental to finding a sweet spot in the AP design space that added significant flexibility with little compromise in the other important factors.

In particular, we believe that PLAN shows that many of the key challenges of building AP systems are surmountable, in particular that AP systems can have good safety and security. PLAN also has shown that even with its simple and limited programming model, quite sophisticated network algorithms can easily be expressed and deployed. Active packets can take the place of passive headers and to good effect.

We invite readers to browse the PLAN home page,

`http://www.cis.upenn.edu/~switchware/PLAN`

which makes available detailed documentation and downloadable software.

Acknowledgments

We would like to thank Alex Garthwaite and Suresh Jagannathan for their valuable feedback on previous drafts of this paper. We would also like to thank Jonathan Smith for using PLAN in his TCOM500 class at the University of Pennsylvania.

REFERENCES

Alexander, D. S. 1998. ALIEN: A Generalized Computing Model of Active Networks. Ph.D. thesis, University of Pennsylvania.

Alexander, D. S., Arbaugh, W. A., Keromytis, A. D., and Smith, J. M. 1998. A secure active network architecture: Realization in SwitchWare. *IEEE Network Special Issue on Active and Controllable Networks 12,* 3, 37–45.

Alexander, D. S., Hicks, M. W., Kakkar, P., Keromytis, A. D., Shaw, M., Moore, J. T., Gunter, C. A., Jim, T., Nettles, S. M., and Smith, J. M. 1998. The SwitchWare active network implementation. In *ML Workshop*, G. Morrisett, Ed. Baltimore, Maryland. `http://www.cis.upenn.edu/~gunter/dist/AlexanderHKKSMGJNS98.ps`.

Alexander, D. S. and Smith, J. M. 1999. The Architecture of ALIEN. In *Proceedings, First International Working Con- ference on Active Networks*, Covaci, Ed. Springer-Verlag, Berlin, 1–12.

Atkinson, R. 1995. Security Architecture for the Internet Protocol. Tech. Rep. RFC 1825, IETF. August.

Baker, F. and Atkinson, R. 1997. RIP-2 MD5 authentication. RFC 2082, IETF. January.

Bhattacharjee, S., Calvert, K. L., and Zegura, E. W. 1996. On active networking and congestion. Technical Report GIT-CC-96-02, Georgia Tech.

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

Bradner, S. and Mankin, A. 1995. The Recommendation for the IP Next Generation Protocol. RFC 1752, IETF. January.

Clark, D., Shenker, S., and Zhang, L. 1992. Supporting real-time applications in an integrated service packet network: Architecture and mechanism. In *Proceedings, 1992 SIGCOMM Conference*. 14–26.

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

Egawa, T., Hino, K., and Hasegawa, Y. 2001. Fast and secure packet processing environment for per-packet QoS customization. In *Proceedings of the IFIP-TC6 Third International Working Conference (IWAN 2001)*.

Feldmeier, D. C., McAuley, A. J., Smith, J. M., Bakin, D., Marcus, W. S., and Raleigh, T. 1998. Protocol boosters. *IEEE JSAC, Special Issue on Protocol Architectures for the 21st Century 16,* 3 (April), 437–444.

Galvin, J. M. and McCloghrie, K. 1993. Security protocols for version 2 of the simple network management protocol (SNMPv2). RFC 1446, IETF. April.

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

Gunter, C. A. 2002. Micro mobile programs. In *Foundations of Information Technology in the Era of Network and Mobile Computing*, R. Baeza-Yates, U. Montanari, and N. Santoro, Eds. IFIP 17th World Computer Congress — TC1 Stream / International Conference on Theoretical Computer Science (TCS 2002), Kluwer, Montreal, Canada, 356–369.

Gunter, C. A. and Jim, T. 2000. Policy-directed certificate retrieval. *Software - Practice and Experience 30,* 15, 1609–1640.

Hadzic, I., Marcus, W. S., and Smith, J. M. 1998. On-the-fly programmable hardware for networks. In *Proceedings of the IEEE GLOBECOM Conference*.

Hawblitzel, C., Chang, C.-C., Czajkowski, G., Hu, D., and von Eicken, T. 1998. Implementing Multiple Protection Domains in Java. In *Proceedings of the 1998 USENIX Annual Technical Conference*.

Hicks, M., Keromytis, A. D., and Smith, J. M. 2003. A Secure PLAN (Extended Version). *IEEE Transactions on Systems, Man, and Cybernetics, Special Issue on Programmable Networks*. To appear.

Hicks, M., Moore, J. T., Alexander, D. S., Gunter, C. A., and Nettles, S. 1999. PLANet: An active internetwork. In *Proceedings of the Eighteenth IEEE Computer and Communication Society INFOCOM Conference*. IEEE, 1124–1133.

Hicks, M., Moore, J. T., and Nettles, S. 2001a. Compiling PLAN to SNAP. In *Proceedings of the Third International Working Conference on Active Networks*, I. W. Marshall, S. Nettles, and N. Wakamiya, Eds. Lecture Notes in Computer Science, vol. 2207. Springer-Verlag, 134–151.

Hicks, M., Moore, J. T., and Nettles, S. 2001b. Dynamic Software Updating. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, 13–23.

Hicks, M., Moore, J. T., Wetherall, D., and Nettles, S. 2002. Experiences with Capsule-based Active Networking. In *Proceedings of the DARPA Active Networks Conference and Exposition (DANCE)*. IEEE, San Francisco, CA.

Hutchinson, N. C. and Peterson, L. L. 1991. The *x*-Kernel: An Architecture for Implementing Network Protocols. *IEEE Transactions on Software Engineering 17,* 1 (January), 64–76.

Kakkar, P., Gunter, C. A., and Abadi, M. 2000. Reasoning About Secrecy for Active Networks. In *Proceedings of the Computer Security Foundations Workshop*.

Kakkar, P., Hicks, M., Moore, J. T., and Gunter, C. A. 1999. Specifying the PLAN networking programming language. In *Higher Order Operational Techniques in Semantics*. Electronic Notes in Theoretical Computer Science, vol. 26. Elsevier. `http://www.elsevier.nl/locate/entcs/volume26.html`.

Katabi, D. and Wroclawski, J. 2000. A framework for scalable global IP-anycast (GIA). In *SIGCOMM*. 3–15.

Lawton, G. 2001. Is IPv6 Finally Gaining Ground? *IEEE Computer 34,* 8 (August), 11–15.

Leroy, X. 2001. *The Objective Caml System, Release 3.02*. Institut National de Recherche en Informatique et Automatique (INRIA). Available at `http://caml.inria.fr`.

LEROY, X., REMY, D., AND WEIS, P. 1999. Objective Caml—a General Purpose High-level Programming Language. *ERCIM News* 36 (January).

MALLET, A., CHUNG, J. D., AND SMITH, J. M. 1997. Operating Systems Support for Protocol Boosters. In *HIPPARCH Workshop*.

MCCLOGHRIE, K. AND ROSE, M. 1991. Management information base for network management of TCP/IP-based internets: MIB-II. RFC 1213, IETF. March.

MOORE, J. T. 2002. Practical active packets. Ph.D. thesis, University of Pennsylvania.

MOORE, J. T., HICKS, M., AND NETTLES, S. 2001. Practical Programmable Packets. In *Proceedings of the Twentieth IEEE Computer and Communication Society INFOCOM Conference*. IEEE, 41–50.

MORRISETT, G., CRARY, K., GLEW, N., GROSSMAN, D., SAMUELS, R., SMITH, F., WALKER, D., WEIRICH, S., AND ZDANCEWIC, S. 1999. TALx86: A Realistic Typed Assembly Language. In *Second Workshop on Compiler Support for System Software*. Atlanta.

MORRISETT, G., WALKER, D., CRARY, K., AND GLEW, N. 1999. From System F to Typed Assembly Language. *ACM Trans. Program. Lang. Syst. 21,* 3 (May), 527–568.

MOY, J. 1998. OSPF version 2. RFC 2328, IETF. April.

NYGREN, E. L. 1998. The Design and Implementation of a High Performance Active Network Node. M.S. thesis, Massachusetts Institute of Technology.

NYGREN, E. L., GARLAND, S. J., AND KAASHOEK, M. F. 1999. PAN: A High-Performance Active Network Node Supporting Multiple Mobile Code Systems. In *Proceedings of the 2nd Workshop on Open Architectures and Network Programming (OPENARCH'99)*. 78–89.

ODERSKY, M. AND WADLER, P. 1997. Pizza into Java: Translating theory into practice. In *Proceedings of the 24th ACM Symposium on Principles of Programming Languages (POPL'97), Paris, France*. ACM, 146–159.

O'MALLEY, S. W. AND PETERSON, L. L. 1992. A dynamic network architecture. *ACM Transactions on Computer Systems 10,* 2 (May), 110–143.

PAPPALARDO, D. 1996. BBN to test RSVP. *Network World 13,* 50 (December), 1,14.

POSTEL, J. 1981a. Internet control message protocol. Tech. Rep. RFC 792, IETF. September.

POSTEL, J. 1981b. Internet protocol. Tech. Rep. RFC 791, IETF. September.

PSOUNIS, K. 1999. Active networks: Applications, security, safety, and architectures. *IEEE Communications Surveys 2,* 1.

RIZZO, L. On the feasibility of software FEC. Tech. Rep. LR-970131, DEIT. Available as `http://www.iet.unipi.it/~luigi/softfec.ps`.

SAVAGE, S., WETHERALL, D., KARLIN, A., AND ANDERSON, T. 2000. Practical network support for ip traceback. In *SIGCOMM'00*.

SCHWARTZ, B., JACKSON, A. W., STRAYER, W. T., ZHOU, W., ROCKWELL, R. D., AND PARTRIDGE, C. 2000. Smart packets: Applying active networks to network management. *ACM Transactions on Computer Systems 18,* 1 (February), 67–88.

SCHWARTZ, B., ZHOU, W., JACKSON, A. W., STRAYER, W. T., ROCKWELL, D., , AND PARTRIDGE, C. 1999. Smart packets for active networks. In *Proceedings of the Second IEEE Conference on Open Architectures and Network Programming (OPENARCH)*. 90–97.

SEHGAL, A., CALVERT, K. L., AND GRIFFIOEN, J. 2002. A flexible concast-based grouping service. In *IWAN'02*.

SONG, S.-K., SHANNON, S., HICKS, M., AND NETTLES, S. 2002. Evolution in Action: Using Active Networking to Evolve Network Support for Mobility. In *Fourth International Working Conference on Active Networks (IWAN'2002)*.

STOICA, I., SHENKER, S., AND ZHANG, H. 1998. Core-stateless fair queueing: A scalable architecture to approximate fair bandwidth allocations in high speed networks. In *SIGCOMM'98*.

SYSTEMS, A. 1985. *PostScript Language Reference Manual*. Addison-Wesley.

TENNENHOUSE, D. L., SMITH, J. M., SINCOSKIE, W. D., WETHERALL, D. J., AND MINDEN, G. J. 1997. A survey of active network research. *IEEE Communications Magazine 35,* 1 (January), 80–86.

THIBAULT, S., CONSEL, C., AND MULLER, G. 1998. Safe and efficient active network programming. In *Proceedings of the Seventeeth IEEE Symposium on Reliable Distributed Systems*. 135–143.

VAN RENESSE, R., BIRMAN, K., HAYDEN, M., VAYSBURD, A., AND KARR, D. 1997. Building Adaptive Systems Using Ensemble. Technical Report TR97-1638, Cornell University. July.

WALL, D. W. 1982. Messages as Active Agents. In *Proceedings, 9th Annual POPL* (Albuquerque). 34–39.

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

WETHERALL, D. J., GUTTAG, J., AND TENNENHOUSE, D. L. 1998. ANTS: A toolkit for building and dynamically deploying network protocols. In *Proceedings of the First IEEE Conference on Open Architectures for Signalling (OPENARCH)*. 117–129.

WHITAKER, A. AND WETHERALL, D. 2002. Forwarding Without Loops in Icarus. In *Proceedings of the 5th Workshop on Open Architectures and Network Programming (OPENARCH'02)*. 63–75.

ZANDER, J. AND FORCHHEIMER, R. 1983. Softnet—An approach to higher level packet radio. In *Proceedings, AMRAD Conference*. San Francisco.

Appendix: PLAN grammar

Note: This is intended to be a human-readable form of the grammar; it is not intended to indicate precedence or associativity of operators.

program	::=	*def-list*
def-list	::=	*def* | *def def-list*
def	::=	*fundef* | *exndef* | *valdef*
fundef	::=	**fun** *var* **(** *paramlist* **)** **:** *type-expr* **=** *expr*
	|	**fun** *var* **(** **)** **:** *type-expr* **=** *expr*
param	::=	*var* **:** *type-expr*
paramlist	::=	*param* | *param paramlist*
exndef	::=	**exception** *var*
valdef	::=	**val** *var* **:** *type-expr* **=** *expr*
type-expr	::=	*tuple-type-list*
tuple-type-list	::=	*nontuple-type-list* ***** *tuple-type-list*
nontuple-type-list	::=	*nonlist-type-exp* | *nonlist-type-exp* **list**
nonlist-type-exp	::=	*base-type* | **(** *type-expr* **)**
base-type	::=	**unit** | **int** | **char** | **string** | **bool** | **host** | **port**
	|	**key** | **blob** | **exn** | **dev** | **chunk**
expr	::=	*value*
	|	*op-expr*
	|	**if** *expr* **then** *expr* **else** *expr*
	|	**raise** *var*
	|	**try** *expr* **handle** *id* => *expr*
	|	**let** *def-list* **in** *expr* **end**
	|	**(** *expr-list* **)**
arg-list	::=	*expr* | *expr* **,** *arg-list*
expr-list	::=	*expr* | *expr* **;** *expr-list*
value	::=	*var* | **true** | **false** | **()** | **[]** | **[** *expr-list* **]**

	|	*int-literal* | *char-literal* | *string-literal*
	|	**(** *arg-list* **)**
	|	|*var*|(*arg-list* **)**
	|	|*var*|( )
op-expr	::=	*id* **(** **)** | *id* **(** *arg-list* **)**
	|	*unary-op expr*
	|	*expr binary-op expr*
	|	*nary-op-expr*
unary-op	::=	~ | **not** | **hd** | **tl** | **fst** | **snd** | # *int-literal* | **noti**
	|	**explode** | **implode** | **ord** | **chr**
binary-op	::=	/ | **%** | ∗ | **+** | − | **and** | **or** | < | <= | > | >= | **=** | <> | **::** | ^
	|	<< | >> | **xori** | **andi** | **ori**
nary-op-expr	::=	**OnRemote** **(** *expr* **,** *expr* **,** *expr* **,** *expr* **)**
	|	**OnNeighbor** **(** *expr* **,** *expr* **,** *expr* **)**
	|	**foldr** **(** *expr* **,** *expr* **,** *expr* **)**
	|	**foldl** **(** *expr* **,** *expr* **,** *expr* **)**
int-literal	::=	*digit* | *nonzero-digit digit-string*
digit-string	::=	*digit* | *digit digit-string*
nonzero-digit	::=	[**1 - 9**]
digit	::=	[**0 - 9**]
char-literal	::=	**'** *character* **'**
character	::=	~[', \\] | \\ | **\\n** | **\\t** | **\\b** | **\\r** | **\\'** | **\\"**
string-literal	::=	**""** | **"** *strchar-list* **"**
strchar-list	::=	*strchar* | *strchar strchar-list*
strchar	::=	~[", \\] | \\ | **\\n** | **\\t** | **\\b** | **\\r** | **\\'** | **\\"**
id	::=	*var* | *var* **.** *id*
var	::=	*varstartchar* | *varstartchar varchar-list*
varstartchar	::=	[**a - z**, **A - Z**]
varchar	::=	[**a - z**, **A - Z**, **0 - 9**, _]
varchar-list	::=	*varchar* | *varchar varchar-list*