PLAN: A Packet Language for Active Networks

Michael Hicks, Pankaj Kakkar, Jonathan T. Moore,
Carl A. Gunter, and Scott Nettles *

Department of Computer and Information Science
University of Pennsylvania

Abstract

PLAN (Packet Language for Active Networks) is a new language for programs that form the packets of a programmable network. These programs replace the packet headers (which can be viewed as very rudimentary programs) used in current networks. As such, PLAN programs are lightweight and of restricted functionality. These limitations are mitigated by allowing PLAN code to call node-resident *service routines* written in other, more powerful languages. This two-level architecture, in which PLAN serves as a scripting or 'glue' language for more general services, is the primary contribution of this paper. We have successfully applied the PLAN programming environment to implement an IP-free internetwork.

PLAN is based on the simply typed lambda calculus and provides a restricted set of primitives and datatypes. PLAN defines a special construct called a *chunk* used to describe the remote execution of PLAN programs on other nodes. Primitive operations on chunks are used to provide basic data transport in the network and to support layering of protocols. Remote execution can make debugging difficult, so PLAN provides strong static guarantees to the programmer, such as type safety. A more novel property aimed at protecting network availability is a guarantee that PLAN programs use a bounded amount of network resources.

*This work was supported by DARPA under Contracts #DABT63-95-C-0073, #N66001-96-C-852, and #MDA972-95-1-0013, and by the National Science Foundation CAREER Grant #CCR-9702107, with additional support from the Hewlett-Packard and Intel Corporations and the University of Pennsylvania Research Foundation. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the U.S. Government.

1 Introduction

Active networking is all about getting programmability into the network.
—Jonathan Smith

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. The header may be viewed as a primitive program in the programming language defined by the packet format specification. This program is interpreted by the protocol software in the routers, and the execution of the program causes the packet to be sent to the next router along the path to the destination. If new functionality needs to be added to the protocol then the packet format and its semantics must change. Or, using our analogy, the *programming language* and its specification must change.

For a widely deployed protocol, such as IP [23], changes to the packet format must be deliberated and agreed upon by a standards body. As a consequence, the introduction of new network services at this level is very slow. For example, there was a span of five or more years from the time RSVP was conceptualized [5] to the time it was deployed [17], even in a very limited manner.

Active networks are an approach to getting more flexibility at the IP (or similarly standardized) level 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.

If this idea is to be realized, a key question is: *what programming interface is needed?* This paper offers an approach based on a two-level distinction between an interoperability layer based on a new programming language *PLAN* (Packet Language for Active Networks) and a level of node-resident *services*, which may be written in general-purpose languages. Each packet contains a PLAN program that replaces the IP packet header and payload. These programs, in turn, tie together service calls to create more complex network functions (*e.g.*, customized routing). Therefore, PLAN can be viewed as a network-level 'glue' language.

More concretely, consider *ping*, one of the most basic network diagnostics. In the current network, it must be provided as a special packet type in the ICMP [22] protocol. In an active network, it can just be a program provided by a user. Here is how it is coded in PLAN (language details will be explained later):

```
fun ping (src:host, dest:host) : unit =
  if (not thisHostIs(dest)) then
    OnRemote(|ping|(src,dest), dest, getRB(),
             defaultRoute)
  else
    OnRemote(|ack|(), src, getRB(),
             defaultRoute)

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

Ping works as follows. This program is placed in an 'active' packet that executes `ping` at the source. The packet is not at the destination and so the first `OnRemote` call causes a packet to be sent that will invoke the `ping` function at the destination. Once there, the second `OnRemote` sends a packet that will invoke the `ack` function on the original source. Although simple, this example shows that protocols can be built with PLAN *without requiring encapsulation or the definition of new packet types.* Later in the paper, we will also discuss how the PLAN programming environment was used to implement all of the protocols necessary to support an internetwork that does not depend on IP. This is a testament to the high level of flexibility in our system, and flexibility is precisely what is needed to allow faster network evolution.

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 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 other kinds of functions.

The rest of the paper describes the PLAN language and system, its architecture, and our implementation of it. We base our discussion on PLAN 3.0, our current design and implementation, and at the end we consider some of the changes that might appear in future versions.

2 Requirements and Design

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 1. 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

Increased flexibility is the primary motivation for active networking. The key question is: how far should we go? Because of our two-level approach, PLAN does not have to be completely general, because general purpose expressibility is provided by the service level language. On the other hand, PLAN must 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: 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.

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

Table 1: Comparison of the Packet and Service levels

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. To address some of these concerns, we have made PLAN a functional, stateless, strongly-typed language. This means that PLAN programs are pointer-safe, and concurrently executing programs cannot interfere with one another.

The other aspects of security are more problematic. For one, network administrators would like to prevent arbitrary users from altering their nodes in arbitrary ways. One possible approach would be to force the authentication of every packet. However, many common operations, such as data delivery, do not require authentication in the current network; this suggests that authentication should be possible but not required. Therefore, we have chosen to restrict PLAN's expressibility so that it may be authentication-free. Functionality requiring authentication can be made available by using services, and we expect to leverage active networking security research like the SANE project [1].

2.3 Performance

PLAN should offer new functionality without compromising the performance of functionality offered by the current network, particularly payload delivery. This would be impossible if all PLAN packets had to be authenticated. A major benefit of keeping PLAN simple is that its interpretation can be lightweight, and thus common tasks can be done quickly. We have avoided adding heavyweight features to PLAN in the belief that if such features are needed, they can be accessed at the service level.

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 long as they only call service routines that terminate (more details on this termination guarantee will be given shortly).

PLAN is based on the simply typed lambda calculus, making it easier to specify a formal semantics. The *a priori* guarantees mentioned above are made possible by the deep understanding of the lambda calculus in the programming languages community. As a supplement to these static guarantees, however, PLAN should also provide some basic error handling facilities.

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 lightweight, which serves all of its design goals (except greater flexibility), is 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.

Our design is based on remote evaluation, rather than remote procedure call. Specifically, child active packets may be spawned and asynchronously executed on remote nodes. This is supplemented with the ordinary form of function call in a language that resembles the simply typed lambda calculus. Our goal was to use the simplest and most basic set of assumptions that satisfied the requirements we have just outlined.

3 PLAN Description

We describe the PLAN language primarily by example, with a highlight of its key constructs[1]. Although PLAN is based upon the simply typed lambda calculus, it is missing some features available in common functional programming languages. We discuss these differences before moving on to a discussion of how PLAN's evaluation model addresses resource bounding.

3.1 PLAN by Example

Figure 1 shows a PLAN version of the utility *traceroute*, a diagnostic program that reports the path taken from a source node to a destination. This version of `traceroute` visits each router between two hosts, collecting a list of nodes visited thus far. At each node it sends this list back to its source to be printed, and creates a new packet that is sent to the next hop to continue the process.

```
fun print_host(h:host, count:int) : int =
 (print(h); print(" : ");
  print(count); print(" ");count+1)

fun ack(l:host list) : unit =
 (foldr(print_host,l,1); print("--\n"))

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

Figure 1: The PLAN *traceroute* program

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

Table 2: The PLAN packet

`PLAN Packets and Chunks.` `Traceroute` creates a number of *PLAN packets* that traverse the network. As shown in Table 2[2], the primary component of each packet is its *chunk* (short for *c*ode *hunk*), which consists of code, a function name to serve as an entry point, and values to serve as bindings for the arguments of the entry function. The PLAN syntax `|f|(a,b,c)` designates a chunk containing the code required to execute the (top-level) entry function `f`, and bindings having the values of the expressions `a`, `b`, and `c`. The other fields in the packet will be explained shortly.

The traceroute program is depicted visually in Figure 2. Each arrow in Figure 2 is labeled with its entry function name, and the arrowheads indicate the nodes on which the corresponding chunks will be evaluated. 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.

Injection. A *host application* constructs a PLAN packet and *injects* it into the active network via a *port* connected to the local PLAN interpreter. This *injection port* is used by PLAN to provide output to the application, and allows the application to submit new packets; it is shown as the pair of dashed lines. Here, the application creates a PLAN packet that is injected at host A with an *evalDest* of A, an initial

[1] A complete grammar is found in Appendix A. More details can be found at `http://www.cis.upenn.edu/~switchware/PLAN`.

[2] These are only those packet fields required by the PLAN language definition; for example, our active internetwork implementation PLANet defines additional fields to assist with special routing protocols.

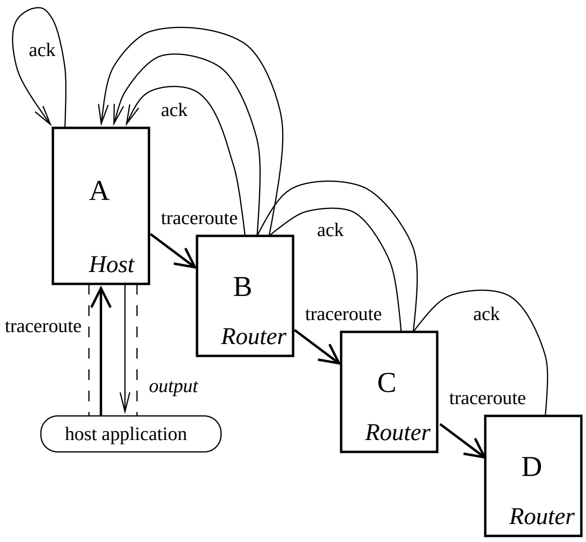

Figure 2: Evaluation of the `traceroute` program

chunk of `|traceroute|(A, D, [], 1)`, an RB of n, and a *routFun* of `defaultRoute`. Output from `print` is passed to the application through the injection port.

Remote Execution. PLAN programs create new packets through calls to the network primitives `OnRemote` and `OnNeighbor`. A prototype `OnRemote` call looks like:

`OnRemote`(*C, evalDest, Rb, routFun*)

This call essentially means 'create a new packet that will evaluate the chunk C on node *evalDest*.' The bindings in the chunk are PLAN values and are evaluated locally, although the function application will be delayed until the new packet arrives on the remote host. *routFun* specifies the routing scheme for the new packet, and *Rb* indicates how much of the current packet's resource bound is transferred to the new packet. It is important to note that the implementation of this primitive enforces that *Rb* be positive but less than the resource bound of the current packet, thus ensuring that the overall resource bound cannot be increased.

`OnNeighbor` is similar to `OnRemote`, except that the *evalDest* must be a neighbor of the current node, eliminating the need for routing. In the example, `traceroute` creates two new packets: the 'backward' packet, created by `OnRemote`, and the 'forward' packet, created by `OnNeighbor`. Consider evaluation of the `traceroute` function on Host A: the backward packet has fields: chunk `|ack|([A])`, evalDest A, routFun `defaultRoute`, and an RB of 1, while the forward packet has fields: chunk `|traceroute|(A, D, [A], 2)`, evalDest B, and an RB of $n-1$.

Routing. Once a packet is created, it is sent to its evalDest for evaluation. The evalDest may be many hops away, so intermediate nodes need a way to determine the 'next hop.' This is done using the packet's routFun field, which names a service function that takes the destination as an argument and returns the next hop towards that destination. At each hop, the RB field is decremented by one; if the resource bound is exhausted, the packet is terminated. In the example, the routing function is `defaultRoute`, which is also used explicitly to determine the evalDest of the 'forward' packet. Since PLAN evaluation need not occur on the intermediate nodes, this sort of routing can be implemented quite efficiently using the same techniques used by the current IP network.

3.2 Language Characterization

Since PLAN's semantic basis is the typed lambda calculus, our examples should have a familiar feel to functional programmers. However, PLAN is missing several common functional programming constructs. We followed a policy of not adding a language feature unless: it was necessary for important applications, did not compromise security, preserved all *a priori* guarantees, and enhanced the usability of PLAN. In that light, let us look at specific components of PLAN.

Flow of Control. In keeping with our goal of simplicity, PLAN has simple flow of control constructs: statement sequencing, conditional execution, iteration over lists with *fold*, and exceptions, all in the usual style. Although function calls are supported, notably absent are recursive function calls. The lack of recursion and unbounded iteration (as well as the monotonically decreasing resource bound in the packet) imply that all PLAN programs terminate. PLAN does not currently support higher order functions, but we have applications in mind that might be simplified by this addition. While in general PLAN does not support pattern-matching, we do provide a binding form of exception handler, which we shall discuss shortly.

The Type System. PLAN is strongly typed, and although it is mostly statically typeable, it is dynamically checked. This arises from the demands of remote programming: static typeability is a benefit to help debugging before injecting a program into the network, yet dynamic checking provides efficient safety (from the nodes' point of view) for mobile scripting code. Although PLAN is currently monomorphic, we see no reason not to add polymorphism; this may be the subject of future work.

In addition to a fairly standard set of base types, PLAN provides a homogeneous list type and a heterogeneous tuple type, but no support for general recursive types, since their utility is questionable without general recursion. Instead, we are considering providing a set of built-in recursive data types, with accompanying tools.

Scoping. 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. The function named in a chunk expression like `|f|(a)` 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 and `OnRemote`, but such calls must decrement the resource bound, so such recursion must terminate.

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

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

Figure 3: A general error-reporting mechanism

Mutable state. PLAN does not provide user-defined mutable state, although some aspects of PLAN, such as the resource bound, are stateful. Not providing state simplifies PLAN in a number of important ways. Firstly, it simplifies transmitting PLAN values to remote nodes, since values can be copied without changing their meaning. Secondly, concurrently running PLAN programs can only share state through service routines, which implies that only the service language must concern itself with concurrency.

3.3 Error handling

Although PLAN provides a basic exception handling mechanism, this is not sufficient for handling all errors in the PLAN system. To make sure that the programmer is notified when something goes wrong, the PLAN system provides two main error handling mechanisms. Firstly, an `abort` service is provided which allows a program to execute a chunk on its source node. This is accomplished by extracting the source from the packet header, sending an error packet carrying the chunk back to the source, discarding any remaining resource bound, and then `eval`ing (see Section 3.4 below) the chunk. The `abort` service coupled with exception handling provides a reasonably flexible error-handling mechanism; an example is shown in Figure 3.

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). 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.

3.4 Encapsulation

Chunks are first-class values in PLAN, and as such, they can be included in the bindings lists of other chunks. In addition, PLAN provides an `eval` primitive to invoke a chunk. Together, these features allow chunks to be manipulated, 'encapsulated,' dispatched, extracted, and finally executed—essentially providing for protocol layering within PLAN. For an example of chunk encapsulation, let us consider how to do UDP-like [21] delivery in the PLAN system. Our program appears in Figure 4.

The heart of the communication is a chunk `c` that delivers the payload to the desired port on the remote host.

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

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

Figure 4: UDP-like delivery in PLAN

This chunk is 'encapsulated' within another chunk `d` which contains code (not shown) to compute a checksum for `c`. `d` is then fragmented using a service `fragment`, which uses the result of another service `pathMTU` to determine an appropriate size for the fragments. The result is a list of chunks containing code for reassembly as well as one fragment each of the chunk `d`. These new chunks are then dispatched using `OnRemote`. As described earlier, these packets are not evaluated until they reach the final destination. Once there, their reassembly code is invoked, producing a reconstituted chunk `d`. Its checksum code will be invoked and confirm the checksum of `c`. If the test succeeds then `c` will be evaluated and deliver its payload to the appropriate port.

3.5 Resource Bounds

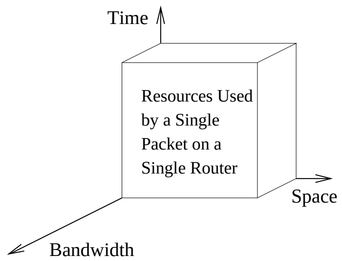

Figure 5: Resource Cube

If active network programs were allowed to use unbounded resources, it would be trivial to use them to implement denial of service attacks. Since unicast IP packets have a time-to-live (TTL) field, a fixed maximum size, and have very simple header processing, they satisfy the following safety property:

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

If PLAN programs are to serve as header replacements, we claim that they should also satisfy this property. Consider the maximum amount of resources consumable by a single packet at a single router as a 'resource cube,' depicted in Figure 5. The RB field of a packet therefore bounds the number of resource cubes that can be produced, since the RB is decreased each time the packet hops to a different node and each time it creates a new packet (by donating some RB to the child).

Bounding the number of cubes, however, is not as difficult as bounding their size. At one extreme, this is possible 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 `OnNeighbor` are unreliable). We have implemented the fixed counter approach, and in our current implementation found that it adds an additional overhead of about 8% to packet processing times.

One might expect that the restrictions placed on PLAN programs, in particular that they terminate, would allow these conditions to be satisfied without timers and space counters. Unfortunately, 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())
```

We are currently exploring further restricting the PLAN language to make programs such as this one illegal. In particular, we could obtain linearly bound execution time by imposing the following constraint:

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

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

where $\mathsf{calls}(g)$ is the number of PLAN functions called from function g. This is still not enough for programs using `fold`, so further restriction is needed (perhaps by consumption of resource bound). Future work in formalizing the resource control policies of PLAN may allow us to improve them and perhaps permit some real-time guarantees for PLAN programs.

4 PLAN Applied

Recently, we have used the PLAN programming environment to build an active internetwork, *PLANet*. PLANet's basic protocols are based on ones used in IP, but with a key difference: all packets are PLAN programs. PLANet currently provides a number of application services, such as reliable and unreliable datagram delivery mechanisms, as well standard network services, such as RIP-style routing [9] and ARP-like address resolution [20]. The basic performance of our user-space implementation using the OCaml bytecode interpreter is quite respectable: a PLANet router running on a dual 300 MHz Pentium II can switch packets at 48 Mbps over a 100 Mbps Ethernet. More details about PLANet and its performance may be found in [11], but the use of the PLAN programming environment to implement it deserves mention.

In PLANet, distributed protocols used to maintain the network, such as routing and address resolution protocols, are implemented as a combination of PLAN programs and services. In particular, protocol state, timing threads, etc. are implemented on each node as services; these services communicate with their counterparts on other nodes via PLAN programs. This has the convenient property that a protocol designer does not need to define new packet formats: all exchanged packets are PLAN programs, and so the packet formats are simply the standard wire representation of those programs. Generally, we found that even with its language restrictions, PLAN was more than adequately expressive for such networking tasks.

5 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 [4] and the Pizza [19] extension to Java [7]. Our most current implementation is in OCaml due to the need for access to the source code to provide Ethernet access for our internetwork PLANet.

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.

6 Related Work

Postscript [27] and Java [7] 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 [30] system, which provided for the execution of packets of multithreaded M-FORTH code. The potential of active networks has been demonstrated by Advanced Intelligent Networking (AIN) [3], which was successful in reducing the deployment time of some telecommunication services from years to weeks. A motivating technology called Protocol Boosters [6] provides customizable protocols, but these customizations would be difficult to deploy without some of the capabilities provided by active packets. Numerous other motivations for the advent of active networks are described in [28].

Several other active networking projects address parts of the same design space as PLAN. The Active Bridge [2] is part of the SwitchWare Project [26] at the University of Pennsylvania. It uses OCaml [4] as a service language for constructing an extensible bridge. The dynamic loading infrastructure provided by the Active Bridge forms a basis for the PLANet internetwork. ANTS [29] (Active Network Transfer System), is a toolkit for deploying Java protocols on active nodes. It provides implicit demand loading of protocols, essentially using Java for both its packet and service languages. The key disadvantages of this approach are that Java is not as secure, simple, or lightweight as PLAN. Hence ANTS seems more attractive as a service-level system than as a packet language; using ANTS to transport service extensions within a PLAN system is something we hope to investigate in the near future. Sprocket is a language from the Smart Packets project at BBN [18]. It uses a special-purpose byte-code language and like PLAN has as a design goal of providing flexible network diagnostics, although it makes no provision for extending its service level dynamically. Sprocket, like PLAN, provides for resource control, although it uses both hop and instruction counts. The Quantum [14] language model provides resource control for distributed computing, including the ability to grant and revoke resources to processes. It is not clear, however, how applicable this more complex model of resource usage would be in the realm of ephemeral active packets.

There are a variety of projects related to networks, distributed computing, and operating systems that are related to PLAN's philosophy of active networks. For example, the Tacoma Project [12] is a programming-language-based system for communications between mobile agents. Interpacket communication, which is forbidden in PLAN, is the core of their approach. As such, they have done more extensive security work than many other active networking projects. The reader is referred to the SwitchWare white paper [25] for more information about systems-related issues in active networking.

7 Future Work

We are currently focusing much of our effort in the area of security. In particular we are looking at mechanisms for trust management and resource bounding. Trust management is important for authorizing the use of sensitive services, such as the ability to modify a routing table. We are looking into using QCM [24], which allows us to easily define a distributed key and authorization infrastructure which should scale nicely in a large network. We are also exploring ways to modify PLAN itself to obtain better security at the expense of expressiveness (one such approach was described in Section 3.5). Preliminary results may be found in [10].

Two topics related to security are that of namespaces and formal semantics. We currently have a very basic method for managing service namespaces; a much more sophisticated mechanism will eventually be needed for scalability. Another related topic is the formal specification of PLAN and its guarantees. Although we have worked hard to keep the language simple and close to areas in which programming language theory is advanced, there are still major challenges in the formulation of service safety and security requirements. It is possible that approaches like proof carrying code [16, 15, 8] might provide some guidance.

A topic of particular interest is how to improve the performance of PLAN processing by active routers. The mobile programming environment provides some unique opportunities for optimization. For example, we have already found that transmitting a program as an AST has space and time benefits over transmitting source, since its representation is more compact and allows tasks like lexing and parsing to be done once at an originating host rather than at each evaluating node. We might alternatively consider a byte-code representation which could presumably further improve PLAN execution times. Applying runtime code generation techniques [13] to service extensions seems very likely to provide substantial service time improvements. Implementing a code cache for commonly used PLAN programs (such as the UDP-delivery example in Section 3.4) might also allow such techniques to work on PLAN programs themselves.

8 Conclusions

We have developed a design philosophy based on a two-level architecture and built a language to support that architecture. Our work so far leads us to believe that this is a very promising approach to active network design. The fact that the PLAN system has been used to implement an internetwork 'from scratch' lends strong credence to this claim. 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, Suresh Jagannathan, and the anonymous referees 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

[1] D. Scott Alexander, William A. Arbaugh, Angelos D. Keromytis, and Jonathan M. Smith. A secure active network architecture. *IEEE Network Special Issue on Active and Controllable Networks*, 1998. To appear.

[2] D. Scott Alexander, Marianne Shaw, Scott M. Nettles, and Jonathan M. Smith. Active bridging. In *Proceedings, 1997 SIGCOMM Conference*. ACM, 1997.

[3] Bell Communications Research Inc. *AIN Release 1 Service Logic Program Framework Generic Requirements*. FA-NWT-001132.

[4] Caml home page. `http://pauillac.inria.fr/caml/index-eng.html`.

[5] D. Clark, Scott Shenker, and L. Zhang. Supporting real-time applications in an integrated service packet

network: Architecture and mechanism. In *Proceedings, 1992 SIGCOMM Conference*, pages 14–26, August 1992.

[6] David C. Feldmeier, A. McAuley, and Jonathan M. Smith. Protocol boosters. *IEEE Journal on Special Aspects of Communication*, 1998. To appear in the issue on Protocol Architectures for the 21st Century.

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

[8] Carl A. Gunter, Scott Nettles, and Peter Homeier. Infrastructure for proof-referencing code. In *International Conference on Theorem Proving in Higher Order Logics*, 1997.

[9] C. Hendrick. *Routing Information Protocol*. RFC 1058, Rutgers University, June 1988.

[10] Michael Hicks. PLAN system security. Technical Report MS-CIS-98-25, Department of Computer and Information Science, University of Pennsylvania, April 1998.

[11] Michael Hicks, Jonathan T. Moore, D. Scott Alexander, Carl A. Gunter, and Scott Nettles. PLANet: An active internetwork. `http://www.cis.upenn.edu/~switchware/papers/planet.ps`.

[12] Dag Johansen, Robbert van Renesse, and Fred B. Schneider. Operating system support for mobile agents. In *Proceedings of the 5th Workshop on Hot Topics in Operating Systems*, May 1995.

[13] Peter Lee and Mark Leone. Optimizing ML with run-time code generation. In *Proccedings of the ACM SIGPLAN '96 Conference on Programming Language Design and Implementation*, pages 137–148, May 1996.

[14] Luc Moreau and Christian Queinnec. Design and semantics of quantum: a language to control resource consumption in distributed computing. In *USENIX Conference on Domain Specific Languages (DSL'97)*, pages 183–197, October 1997.

[15] George C. Necula. Proof-carrying code. In *Proceedings of the 24th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '97)*. ACM Press, 1997.

[16] George C. Necula and Peter Lee. Safe kernel extensions without run-time checking. In *Second Symposium on Operating System Design and Implementation (OSDI '96)*, 1996.

[17] D. Pappalardo. BBN to test RSVP. *Network World*, 13(50):1,14, December 1996.

[18] C. Partridge and A. Jackson. Smart packets. Technical report, BBN, 1996. `http://www.net-tech.bbn.com/smtpkts/smtpkts-index.html`.

[19] Pizza home page. `http://www.math.luc.edu/pizza`.

[20] David C. Plummer. *An Ethernet Address Resolution Protocol*. RFC 826, November 1982.

[21] J. Postel. *User Datagram Protocol*. RFC 768, ISI, August 1980.

[22] J. Postel. *Internet Control Message Protocol*. RFC 792, ISI, September 1981.

[23] J. Postel. *Internet Protocol*. RFC 791, ISI, September 1981.

[24] Query Certificate Manager project home page. `http://www.cis.upenn.edu/~qcm`.

[25] Jonathan M. Smith, Dave J. Farber, David C. Feldmeier, Carl A. Gunter, Scott M. Nettles, William D. Sincoskie, and Scott Alexander. Switchware: Accelerating network evolution. `http://www.cis.upenn.edu/~switchware/papers/sware.ps`, 1996.

[26] SwitchWare project home page. `http://www.cis.upenn.edu/~switchware`.

[27] Adobe Systems. *PostScript Language Reference Manual*. Addison-Wesley, 1985.

[28] David L. Tennenhouse, Jonathan M. Smith, W. David Sincoskie, David J. Wetherall, and Gary J. Minden. A survey of active network research. *IEEE Communications Magazine*, 35(1):80–86, January 1997.

[29] David J. Wetherall, John Guttag, and David L. Tennenhouse. ANTS: A toolkit for building and dynamically deploying network protocols. In *IEEE OPENARCH*, April 1998.

[30] J. Zander and R. Forchheimer. Softnet—An approach to higher level packet radio. In *Proceedings, AMRAD Conference*, San Francisco, 1983.

Appendix A: 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*)
| **RetransOnRemote** (*expr* , *expr* , *expr* , *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*