PLANet: An Active Internetwork

Michael Hicks, Jonathan T. Moore, D. Scott Alexander
Carl A. Gunter, and Scott M. Nettles*
Department of Computer and Information Science
University of Pennsylvania

August 2, 1998

Abstract

We present *PLANet*: an active network architecture and implementation. In addition to a standard suite of Internet-like services, PLANet has two key programmability features:

1. all packets contain programs
2. router functionality may be extended dynamically

Packet programs are written in our special purpose programming language PLAN, the Packet Language for Active Networks, while dynamic router extensions are written in Caml, a byte-code-interpreted dialect of ML.

Currently, PLANet routers run as Linux user-space applications, and support Ethernet and IP as link layers. On 300 MHz Pentium-II's attached to 100 Mbps Ethernet, PLANet can route 48 Mbps and switch over 5000 packets per second. We demonstrate PLANet's utility by showing experimentally how PLANet can non-trivially improve application and aggregate network performance in congested conditions.

1 Introduction

The applications that the Internet must support and the underlying network technologies that it uses to support them are evolving rapidly. Regrettably, the Internet itself is hard to change. Coupled with the desire of application and network programmers to customize the network to match their special needs, this evolutionary rigidity motivates research into *active* networks. An active network is one that is programmable and extensible; exploring and justifying this change in network architecture requires understanding the advantages, disadvantages, and tradeoffs of the new approach. A number of questions arise. First, how does one build such a network and what programming abstractions should it provide? Secondly, will the added flexibility inherent in the network unduly compromise the performance and safety of the network (in particular, relative to the current Internet)? Finally, how can applications and service providers benefit from the improved capabilities of an active network? In this paper we address these questions experimentally using a new internetwork, PLANet.

PLANet is an active network that is programmable in two ways. First, packets contain programs written in a special-purpose packet language called *PLAN* (Packet Language for Active Networks) [14]; these programs serve a role similar to the header of a traditional packet in providing control of how packets operate inside the network. For simplicity and added performance, a PLAN packet may choose to rely on 'passive' transport and thus need not be evaluated at every intermediate node. PLAN programs are typically small and serve as glue for router-resident *services* that provide general-purpose functionality not expressible in PLAN alone. Such services may provide simple information such as the address of the current host, or more substantial functionality, such as segmentation and reassembly.

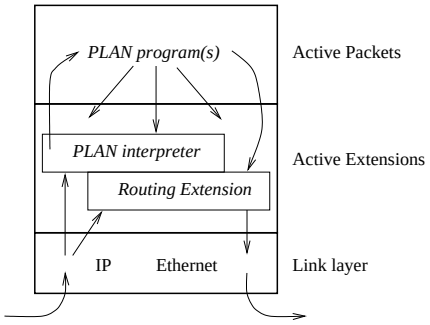

Figure 1: PLANet node architecture

In addition to evaluating PLAN programs, nodes may be programmed by dynamically loading *active extensions* written in OCaml [7], a dialect of the ML programming language. These extensions implement new functionality or enhance existing functionality, and may provide a service interface to PLAN programs. Active extensions are used to program essential services needed to operate the network, like address resolution, routing, DNS, etc. while PLAN programs are used as a means of 'smart' communication between nodes. This node architecture is visualized in Figure 1. First, a packet arrives through the link layer interface; PLANet currently supports Ethernet as a true link-layer and IP as a virtual link-layer. If the program has reached its evaluation destination, it is passed to the PLAN interpreter to be evaluated; otherwise, it is simply routed onwards. During evaluation, PLAN programs may make service calls, including the service that sends PLAN programs to other nodes to be evaluated.

In measuring the performance of PLANet, we wanted to examine the costs of common, non-active network operations. Therefore, we measured both latency and bandwidth for 'dumb' packets (whose only program is to deliver a payload) and compared the results to those of a kernel-based

*This work was supported by DARPA under Contract #N66001-96-C-852.

IP implementation. These measurements place an upper-bound on the overhead of using our active network architecture/implementation as compared to a non-active one.

Finally, we wanted to determine the possible gains from using our active network. While it is impossible to measure the benefit of an active network in a strictly quantitative manner, we are able to demonstrate the benefits of some optimizations that can be readily programmed using our system. Given the need to carry out communication between two nodes, we consider (1) the ability to modify a given path between the nodes by changing features of intermediate routers, and (2) the ability to select a good path between the nodes. Our aim has been to show how programmability can yield potential performance gains for problems that are not likely to have a unique solution fitting all needs, thus recommending the advantage of a flexible approach.

The paper is conceptually divided into three parts, each answering one of the questions posed earlier. The first part describes the design and implementation of PLANet and the motivation behind the abstractions we provide. The second part presents the basic performance of PLANet in comparison to a kernel-based IP implementation. The final part describes some useful applications of PLANet and presents some measurements of expected benefits. We wrap up with related work and conclusions.

2 Active Network Design

In designing PLAN and PLANet, we identified two design-space axes that are important for active networking. The first axis addresses possible mechanisms for network programmability. At one extreme, each packet carries a program that may be evaluated at intermediate hops to effect its routing, compute some useful result, or in some other way affect the network. Here, networking changes occur by changes at the programmable packet level. The other extreme is that packets are passive, and that extensibility is provided by downloading code into the routers. There is a mixture of these approaches in which packets carry programs that may refer to and invoke more general (and loadable) router-resident functionality.

The second axis determines at which hops active evaluation should occur. The Internet currently lies at one extreme: interesting 'active' processing can occur only at the endpoints. Another view is that active processing should occur at every intermediate hop. Again, there is an intermediate position that allows evaluation at *some* of the intermediate hops, thus allowing more flexibility than end-to-end approaches while avoiding unnecessary processing overhead for simple tasks which do not require evaluation at every hop.

PLANet implements the more flexible intermediate positions of these two design-space axes, allowing us to use it to explore a larger portion of the active networking design space. Parts of this space have been explored by other projects. The Active Network Transport System (ANTS) [25] and Sprocket [22] are instructive examples to contrast with each other and with PLANet: ANTS relies entirely on dynamically extensible services while, at the other extreme, Sprocket relies entirely on programs held in packets. ANTS and PLANet also differ on the other design axis since ANTS 'evaluates' its packets on all of the active nodes through which they pass, an approach we have gone to some lengths to avoid. Sprocket also evaluates on each active node, but instead of carrying a key referencing a program on the active node or triggering an installation of a program with that key onto the node, it carries the actual program, which is written in a very dense CISC-like code. Sprocket programs are designed to be no more than 1000 bytes long so they can fit in a single packet. Other projects have explored additional parts of the AN design space; we discuss more related work in the concluding section.

2.1 PLAN

Much of the design of PLANet is based upon the language of its packets, the Packet Language for Active Networks, or PLAN. We briefly discuss PLAN and its programming environment here.

PLAN is a small language that has elements in common with Lisp, Scheme, and ML, but differs from these in being based on remote evaluation in addition to local evaluation. That is, much of PLAN's computation is carried out by calls to a primitive called `OnRemote` that causes an expression to be evaluated at a remote node. PLAN is purely functional, a design that aids the implementation of remote evaluation, but a program is able to make calls to state-altering service functions on routers. Another special characteristic of PLAN is a resource-limited semantics which ensures that PLAN programs always terminate and visit only a fixed number of nodes. We refer the reader to [14] for more details about the PLAN language.

Each node in the PLAN network is augmented by service routines which may be invoked during the evaluation of PLAN programs. PLANet allows service routines to be dynamically loaded as active extensions. The combination of transient PLAN programs with router-resident, loadable service routines provides a two-level architecture with which we can measure and evaluate interesting active networking alternatives. We examine such alternatives in Section 4.

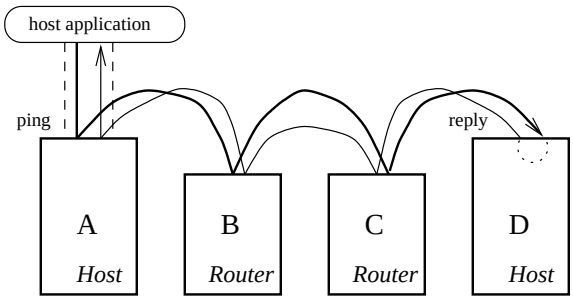

Figure 2: PLAN ping

As an example, consider the following PLAN program, which implements the functionality of the well-understood *ping* program:

```
fun reply(payload:blob):unit =
  print("Success")

fun ping(payload:blob):unit =
  OnRemote(|reply|(payload),
           getSource(), getRB(), defaultRoute)
```

To ping some destination, the source node remotely evaluates the `ping` function on the destination. In turn, the evaluation of `ping` causes `reply` to be evaluated back on the source which prints `Success`. Note that both functions have a single argument that names some arbitrary payload of the program (allowing us to create different sizes of packets). `Print`, `getSource` and `getRB` are service routines; calling a service routine is syntactically identical to calling a PLAN function. Service routine names are resolved during PLAN program evaluation; if a service routine is not present an error occurs which may be handled on the source.

The evaluation of this program is depicted in Figure 2. Here, a host application on node A wishes to *ping* host D. The host application first marshals the program into an active packet, and sends it towards the evaluation destination. Here, since the destination is not adjacent to the source, the packet must be routed at each hop. No evaluation occurs on intermediate routers; routing is handled by `defaultRoute`, a service function present on each node that provides 'next-hop' routing information. Once the packet reaches D (the bold line in the figure) it is unmarshaled and evaluated, resulting in another packet being created and sent back to A to evaluate `reply`.

PLAN was designed to be flexible enough to write useful programs, but limited enough that its programs will not pose a security risk. This is in contrast with service routines, which are general-purpose and may need to be protected by cryptographic means. (Here our coverage of security issues is limited due to space considerations. For more details, we refer the reader to our other papers on the topic [1, 12].) Other functions we have implemented as PLAN programs include 'route scouting' which seeks out low-congestion routing paths (as described in Section 4), source-directed multicast, traceroute, and network-DFS, to name a few. More detail about programming with PLAN may be found in [13].

2.2 PLANet

PLANet is our active internetwork implementation based on the PLAN environment described above. Whenever possible, we have drawn from the experience of IP [21, 6] in making implementation decisions, so as to leverage the experience represented by that design. The remainder of this section details the design and implementation of PLANet.

The fundamental programming model provided by PLAN is remote evaluation. The means by which a PLAN program is transmitted to its evaluation destination is a matter of design, but practical experience with the Internet indicate that the following are required for an internetwork:

- uniform, network-layer addressing and packet formats
- address resolution from link-layer to network-layer addresses
- routing between physical networks
- error reporting

These functions form a core part of PLAN itself: with the exception of address resolution, each is visible as an abstraction in the language. We have also adopted from the Internet the idea that remote evaluation is *best-effort*; reliable delivery may be achieved with the addition of appropriate services.

Beyond these basic functions, we have implemented other services that have counterparts in the Internet. These services do not *require* standardization and so are not part of PLAN but are part of the loadable service routines available to PLAN programs. These include a domain-name service, a fragmentation and/or segmentation service, reliable stream transport, network management, among others. Indeed, it is the flexibility of being able to augment active nodes with such 'non-standard' services that makes the prospect of active networks appealing. For instance, as we mentioned before, a standard routing scheme is not strictly needed since different packets may choose different schemes; we experiment with this idea in Section 4. Of course, *some* routing functions are needed, and routing functions of interest must be widely deployed if they are to be widely useful. Our approach has been to supply a collection of basic network functions as PLAN programs or router-resident service routines; packets do not need to use the ones that we provide because new ones can be installed, but some basic ones will be available.

2.2.1 Packet Formats

To allow interoperation between diverse physical networks, an internetwork must define a set of standard packet formats. In PLANet *all packets consist of PLAN programs*, so this standardization reduces to defining a standard marshaling scheme for PLAN programs. This greatly simplifies the task of the implementor of a new network service, as they need only concern themselves with *what* information needs to be communicated, not with *how* that information is encoded.

An extreme view of active networks would allow packets to have only as much formatting as required to unmarshal a program for evaluation. However, this would entail unmarshalling the program at every hop, potentially an expensive operation. The experience with the Internet has been that most packets only require basic transport. We expect this will be true for PLANet as well, and so we make the assumption that most PLAN packets will require routing but not evaluation. Therefore, we have placed the information necessary for routing in a standard position to improve routing efficiency. The PLAN packet format is illustrated in Figure 3.

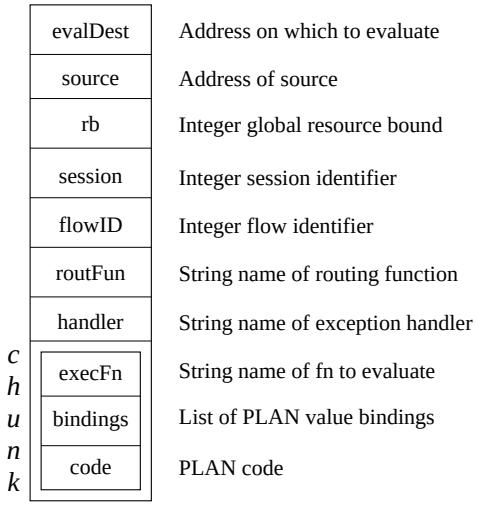

Figure 3: PLAN Packet Format

Chunks The program portion of the packet is contained in the final three fields which are collectively called a *chunk (short for code hunk)* , which is derived from the 'function call' provided to `OnRemote`. In the ping example above, the `ping` function made the call

```
OnRemote(|reply|(payload),
         getSource(), getRB(), defaultRoute)
```

Here, `reply` is the *execFn*, `payload` is the only *binding*, and the *code* is at least the part the current program that is needed to evaluate the *execFn*; in this case, it is just the code for the `reply` function.

Addressing The first two fields of the packet indicate the *evaluation destination* (*evalDest* for short), which is where the packet's chunk should be evaluated, and the *source*, which names the packet's oldest ancestor (used for error reporting). In the case of the ping example, the source of the `reply` packet would be A (not D).

Currently, PLANet uses 48 bit addresses, with one address for each network interface on a node. In our implementation we typically use the assigned IP address for the interface together with a 16 bit port number. This choice makes it easy for us to use UDP/IP as a 'link layer' to create tunnels between physically separated PLANets.

As in any internetwork, we must resolve network layer addresses into link layer ones for physical transport. PLANet adopts the same basic technique as ARP [19]. The key difference is that in PLANet ARP requests and responses are not special kinds of packets, but rather PLAN programs. Most simply, we did this to maintain the invariant that all packets in PLANet contain PLAN programs. This obviates the need for special-purpose ARP processing; it can be a service like any other.

Remaining Fields Let us skim through the remaining packet fields; the crucial ones are more fully explained later in the paper. The third field of the packet *rb* is a resource bound similar to the IPv4 Time-To-Live field or the IPv6 hop count field. The *session* field provides an identifier for the end application on a host (similar to the protocol field in IP), while the *flowId* field provides an identifier for a packet flow through a router. The roles of these fields will be illustrated in a QoS routing example later. In order to evaluate its chunk at *evalDest* the packet must potentially be routed, so the *routFun* field supplies the name of the routing function to do this. The chunk is not evaluated on any of the nodes between its current location and its *evalDest* and may therefore be treated as a 'passive' packet for efficient transport. The *handler* field provides the name of a service routine on the *source* that will handle certain error communications the router may choose to offer; examples are given in the next subsection.

Packets as Data While requiring all packets in the network to be marshaled PLAN programs simplifies the question of packet formatting, it complicates the implementation of remote evaluation. The issue is simple: if a packet is a program to be evaluated remotely, then what should be done if the size of the marshaled program exceeds the path MTU? The active packet must be cut into pieces and reassembled before it can be evaluated. If these pieces are also to be active packets, then each must contain a program that contains a fragment of another program. Moreover, each must coordinate with the other fragments to carry out reassembly and then evaluate the reassembled program. We achieve this by making chunks data objects that can be manipulated, allowing them to be fragmented, checksummed and so on. We also add features to evaluate the reassembled chunk. The ability to make chunks *first-class* also provides an elegant mechanism for enabling encapsulation.

Currently the link layers we support (IP and Ethernet) provide an MTU of at least 1500 bytes, and so we segment all oversized packets to this size. Note that it is easy to write a PLAN packet that does path MTU discovery (perhaps as part of connection setup) and we could easily use this information to set segment sizes. A more interesting challenge arises when routing changes such as those in the adaptive routing example we consider later (Section 4) could result in varying path MTUs. In this case we will need to dynamically adjust segment sizes.

2.2.2 Routing

The use of a per-packet routing function in PLANet arose as a means to explore the space between two extremes. At first, we considered an approach which required that all legal evaluation destinations be located on the same network as the sender; to reach distant networks packets would evaluate on each hop towards that network to determine what the next hop should be. While this approach affords the maximum flexibility, it is also the most costly, both in terms of performance and programmer convenience. The other extreme is to implement a single routing protocol for all nodes in the network (or at least autonomous portions of it) and force all packets to be routed based on its determinations, as in the Internet[1]. This approach suffers from the same problem as the first: while 'dumb' packets are efficiently routed, packets wishing to follow non-standard routes must pay the penalty of per-hop evaluation. The per-packet *routFun* serves as an alternative that does not favor one particular routing strategy over another but is still 'lightweight': it looks up the next hop without requiring the packet to be unmarshaled

[1] Of course, packets can in principle be source-routed, but this option is so infrequently used that many protocol stacks are compiled without support for it.

and/or evaluated. The only additional cost is the lookup of the routing function itself.

Most packets will specify the `defaultRoute` routing function, which in PLANet is implemented with a simple distance vector routing service based closely on RIP [11]. On the other hand, more savvy applications may make use of customized routing functions, as illustrated in Section 4.

2.2.3 Diagnostics and Error Handling

Network errors typically fall into two categories, *packet-level* errors and *network-level* errors. Packet-level errors occur when a particular packet fails to properly reach its destination, perhaps because that destination is unreachable or the TTL of the packet has expired. Packet-level errors are often symptomatic of network-level anomalies: a circularity in the routing table causes packets' TTL fields to expire or a failed node causes a destination to become unreachable. In a well-designed network, applications should be able to *handle* packet-level errors, and network administrators should be able to *diagnose* network-level errors.

In the Internet, low-level diagnostics and errors are reported with ICMP [20]. The fundamental difficulty is that the sorts of errors or diagnostics reported are fixed until a re-standardization defines new ones. This limits the vocabulary of an application/operating system in dealing with errors or a network administrator in making use of diagnostics.

In PLANet, diagnostics such as *ping* or *traceroute* are simply PLAN programs, so new diagnostics can be crafted as needed. Typically, diagnostic programs will derive information about a node or query it directly (perhaps to find the current packet queue length). If a PLAN interface to some required information is not available, then active extensions can be downloaded to provide it, subject, of course, to security considerations. This provides significant new flexibility in diagnosing network problems, since diagnostics can be created on-the-fly. This ability is increasingly important as the Internet becomes more wide spread and harder to change.

Furthermore, the means by which packet-level errors may be handled is greatly enriched in PLAN, since the level of abstraction has been raised to the remote evaluation-level as opposed to packet-level. PLAN enables three classes of error handling:

1. *handling that can be done within the current evaluation with a normal continuation.* Example: a PLAN program attempts to invoke a particular service which is not currently loaded, so an exception is raised. The program catches the exception and instead invokes an alternative version of the service.

2. *handling that must abort the current evaluation and send a given message back to the source.* Example: in trying to create a new packet, `OnRemote` raises an exception indicating that not enough resource bound is available. The program catches the exception and aborts the computation, resulting in a remote evaluation on the source to recover from the error.

3. *handling that must be done by the router* **before** *evaluation may begin but where the source is given some information about the problem.* Example: on the way to its evaluation destination, the packet reaches a link with an MTU smaller than the packet size. The router notifies the source by remotely evaluating the function indicated in the packet's *handler* field with the exception raised (`MTUexceeded`) and information about the packet.

While each of these mechanisms is present in the current version of PLANet, we have yet to thoroughly study their possible uses. For example, one could implement 'reactive path-MTU discovery' of the flavor of IPv6. In the third example presented, the handler invoked on the source for exceeding the MTU could resend the original packet in properly-sized fragments.

2.3 Implementation Details

PLANet is implemented in OCaml v1.07 [7] in user space under the Linux kernel version 2.0.30. Our implementation is in part based on the Active Loader, described in [2]. The choice of implementing in user-space was largely motivated by expedience, and we have efforts ongoing to integrate PLANet into the kernel of various operating systems.

Our choice of OCaml is motivated by several concerns. OCaml is a type-safe, garbage-collected language. These features are exploited to provide for safe mobile code, an issue that will become even more important if we attempt to move PLANet into the kernel to improve performance. The implementation of OCaml also provides for machine-independent and downloadable code, which we need for dynamically loading active extensions. OCaml has been used like Java to provide for web-based mobile code [16]. Our earlier IP-based implementation used Java (or, more precisely, Pizza [18], a Java extension). However, when we attempted a parallel development in OCaml we found that the code was easier to write and more efficient, so OCaml was adopted as the preferred language for PLANet. The free availability of OCaml's source code, which we needed to modify for direct access to our networking hardware, was also a major advantage.

3 PLANet Performance

This section addresses the second question posed in the introduction: can an active network attain reasonable performance, especially for common operations? We begin by describing our basic experimental setup and the statistical issues of our measurements. We then present measurements of both the latencies to send PLANet packets and the bandwidth that PLANet achieves. We finish by looking at some of the aspects of the PLANet implementation that influence its performance.

Our analysis leads us to believe that given the prototype nature of our implementation, PLANet's performance is quite respectable—within a factor of two of the optimal link bandwidth. Future work will involve attempting to reduce our overheads, but even our current numbers suggest that an active network need not be a slow one.

3.1 Benchmarks and Systems Measured

For both latency and bandwidth we compared IP running over Ethernet (*IP*) to PLANet running over Ethernet (*PLAN/Ether*). In order to assess certain overheads (such as forwarding costs) we additionally measured PLANet running over IP (*PLAN/IP*). When referring to either or both of the two PLAN implementations, we shall simply use the term *PLANet*.

We made measurements of machines that were directly connected (one hop), as well as communicating through one,

two, or three switching elements (two, three, or four hops). For IP, switching is performed by an in-kernel router, while PLAN/Ether and PLAN/IP are switched by our PLANet router running in user-space.

For each experiment we used four packet sizes: the minimum size feasible given any overheads (i.e. a 0 byte payload), and ones resulting in 342, 750, and 1500 byte Ethernet payloads. The 342 byte size is the smallest Ethernet payload that can support all of our experiments. Space prevents us from presenting all of our measurements, but we will make final versions available over the World Wide Web.

After our initial set of measurements, we were interested in determining the overheads due to our implementation. In particular, we wanted to determine upper bounds on the possible routing performance in user-space and when executing OCaml byte-codes in user space. To measure these two bounds, respectively, we wrote two user-space bridge programs, one in C and one in OCaml. The C bridge uses the `recv` system call to read packets from the raw Ethernet interface. It then uses a fixed (two entry) table to find the outgoing interface and does a `send` out that interface. The C bridge does no copying, except that necessary for the two kernel crossings. The OCaml version of the bridge works in the same way as the C bridge, but has the additional overhead of executing OCaml bytecodes.

3.2 Experimental Conditions

For the experiments in this section, all the machines used were 300 MHz Pentium-II's with split first level caches for instruction and data, each of which is 16 KB, 4-way set associative, write-back, and with pseudo LRU replacement. The second level cache is a unified 512 KB and operates at 150 MHz (we were unable to find any additional details about the second level cache). These machines receive a rating of 11.7 on SPECint95 and have 256 MBs of EDO memory. The machines used as the source and destination for the experiments in Section 4 are the same, but only have 64 MBs of EDO memory. In all cases the machines have enough memory that they do not page fault. For Section 4, we also use two 200 MHz Pentium-Pro's which have enough computing power to saturate the network when running as load generators.

All machines are equipped with enough 100 Mbps Ethernet interfaces to construct the required topologies. For the bandwidth and latency measurements presented in this section, we simply connect the machines linearly. Section 4 uses a more complicated topology which we will describe there.

Our latency results are based on repeated measurements of individual "pings", while our bandwidth measurements are based on measuring the transmission time for 5000 packets. In all cases we collect 21 trials and compute the mean, median, standard deviation, and quartiles. For most of our measurements, the standard deviation is less than 5% of the mean, but we do observe somewhat skewed distributions. For this reason, as recommended by Jain [15], we report medians, since they are less sensitive to influence by skewed distributions. All times reported are elapsed times, and are measured with a clock having a 4 μs resolution, which is more than adequate for the measurements presented here.

3.3 Latency Measurements

We measured the latency of both PLANet and IP as a function of the number of network hops and the payload size. For IP, we used the standard ICMP-based ping program (*ping/IP*). For PLANet, we used two PLAN versions of ping: the one presented in the earlier example which evaluates once on the destination and once back on the source (*ping/noeval*), and another version that evaluates on every hop, determining its route as it goes (*ping/eval*). With these tests we can determine the basic latencies of PLANet, compare those to IP, and evaluate the cost of packet evaluation per-hop.

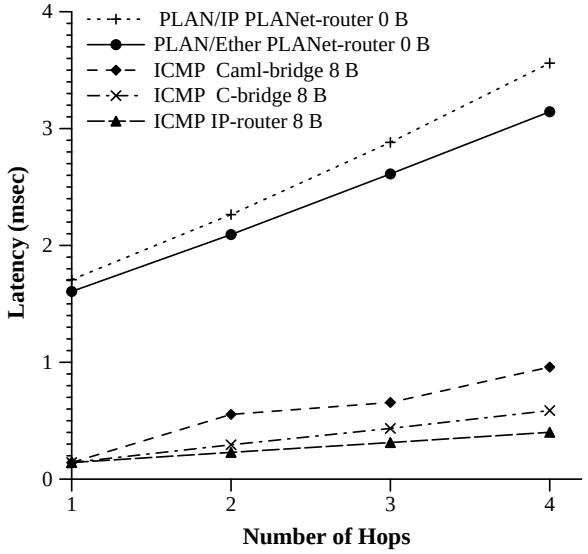

Figure 4: Latencies for Switching Alternatives

Figure 4 shows the switching latency for ping/noeval. The X-axis shows the number of hops while the Y-axis shows the round-trip latency in milliseconds. Only minimum packet sizes are shown here. In addition to showing measurements for IP, PLAN/Ether, and PLAN/IP, we show IP being switched by both the C-bridge and the OCaml-bridge.

Kernel crossing overheads Comparing minimally-sized packets over one hop, IP ping has a latency of 0.16 ms, while PLAN/Ether ping has a latency of 1.6 ms. The majority of this cost comes from the PLAN evaluation on the endpoints; this is analyzed in more detail later. The next sizeable portion of this difference can be attributed to kernel crossings. IP ping only requires two system calls, both on the source: once to send the packet, and once to receive the response. PLANet ping is implemented as a host application which communicates with the PLAN interpreter via PLAN ports. Since both the PLAN ping program and the PLAN interpreter are in user space, two crossings are needed to hand the packet to the interpreter (one `send` by the application, and one `recv` by the interpreter) followed by a third when the interpreter puts the packet on the wire. The packet then must cross the kernel boundary twice on the destination (there and back), and finally do three more crossings to go through the source interpreter and back into the ping application. This results in a total of 8 crossings, 6 of which could be eliminated with an in-kernel implementation. Each additional hop imposes 4 more crossings (2 per switch per direction) for both the PLAN router and the bridges, while IP is switched in-kernel. We estimate the cost of these crossings later in the discussion.

	IP	C	OCaml	PLAN
Per Packet (μs)	36	71	131	259
Per Byte (μs)	0.13	0.15	0.15	0.16

Table 1: Switching overheads

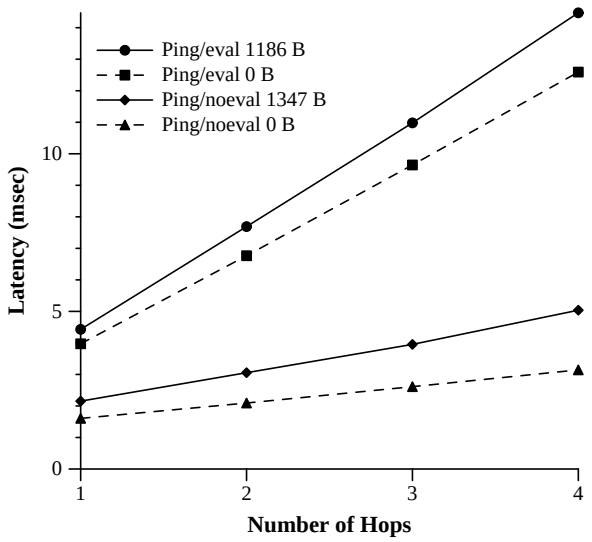

Figure 5: Evaluation Overheads

Switching costs We were interested in the detailed per-packet and per-byte switching overheads. To calculate these costs, we first made a linear least squares fit to each of our constant Ethernet packet size measurements using hops as the independent variable. The slope of these lines is the cost per hop (and remember each hop is traversed twice) for a given switching mechanism and size. Next, to find the fixed cost per packet and the cost per byte of each switching mechanism, we used the packet size as the independent variable and the cost per hop as the dependent variable. This gave us three different estimates of these values for the bridges (IP, PLAN/Ether, and PLAN/IP), and two estimates each for the IP (IP and PLAN/IP) and PLAN (PLAN/Ether, and PLAN/IP) routers.

Table 1 shows the (averaged) results of our calculations, for the IP router (IP), the C bridge (C), the OCaml bridge (OCaml), and the PLAN router (PLAN). We have divided the per-hop calculations by 2, so these values are for one pass through the switch. The lack of kernel crossings for the IP router makes it unsurprising that this system has lower per-byte costs, and since the two bridges and the PLAN router do no copying internally, and make the same system calls, it is to be expected that they will have (essentially) the same per-byte cost. The most striking differences were in the per-packet overheads. It would appear that the C bridge added about 35 μs to the latency over IP – an indicator of the cost of a single crossing of the user/kernel boundary. There is an additional 60 μs penalty for going to the OCaml-bridge and a further 130 μs penalty (a doubling) when going through the PLANet implementation. Thus it is probably worthwhile to further optimize PLANet before attempting to move OCaml into the kernel.

Ping with intermediate evaluation We also wanted to find out the overhead of PLAN program evaluation. Figure 5 compares the latencies of ping/eval and ping/noeval. It shows only the maximum and minimum packet size measurements for PLAN/Ether.

From the figure it is obvious that evaluation in this case is expensive, with the added cost per hop for the smallest packets being almost 2.5 ms. Also note that ping/eval has an additional space overhead: 314 bytes per packet as opposed to 153 bytes for ping/noeval. For a fixed Ethernet packet size (note the graph shows usable payload sizes, 1186 bytes for ping/eval and 1347 bytes for ping/noeval results in the same size ethernet frame), many of the overheads, such as kernel crossings, are the same for both approaches. However, at each hop, the evaluating version must unmarshal its code, evaluate it, and then remarshal the version to be sent on to the next hop. A more sophisticated implementation might be able to avoid the unmarshalling and remarshalling by executing in place.

Using the same statistical techniques as before, we found that ping/eval has a fixed cost of about 1400 μs and a per-byte cost of about 0.19 μs for each traversal of the router. Ping/noeval has fixed cost of 260 μs and per-byte cost of 0.16 μs. The difference suggests that the fixed cost of (this particular) evaluation is about 1200 μs and the per-byte cost is .04 μs (using unrounded original values). This is clear evidence that avoiding evaluation on intermediate routers can have important performance advantages.

3.4 Throughput Measurements

We also measured bandwidth as a function of hops and payload size. We used UDP packets to provide transport for IP, while for PLANet we used simple PLAN programs providing UDP-like functionality; these programs only evaluate at the endpoint to deliver their data.

For our IP measurements, we use `ttcp`, while for PLANet we wrote our own measurement functions. One way our PLANet results differ from IP is that for PLANet we adjusted the rate of the sender until the packet loss at the receiver was consistently less than 1%. This results in lowering our peak transmission number by a few megabits per second, but we believe it better represents the load that PLANet can reasonably support. Unfortunately, `ttcp` did not allow us to make the same adjustment. Our reported bandwidths are in terms of useful payload received.

Figure 6 shows the results for a subset of our bandwidth measurements for maximally sized packets. For UDP, we present measurements for both IP routing and switching with the OCaml bridge; the C bridge has the same performance as the IP router. For PLANet, we show both PLAN/Ether and PLAN/IP.

For one hop with 1500 byte Ethernet payloads, PLAN/Ether achieves a rate of 56.0 Mbps which is 60% of the maximum possible 93.4 Mbps bandwidth possible for the 1437 bytes of useful payload. In this case, the receiver is the bottleneck; we measured the maximum send bandwidth to be the peak bandwidth. Adding PLAN routers for two through four hops reduces our throughput to 48.3 Mbps or 52% of the peak possible bandwidth. Here, the router is the bottleneck. We did a linear regression on the bandwidths from 2 to 4 hops and found that each additional router reduces our bandwidth by 0.8 Mbps. For PLAN/Ether with PLAN routers, for minimum sized packets, (which carry no payload and thus get no bandwidth by our previous metric), we can transmit 7200 packets per second over one hop, and 5100

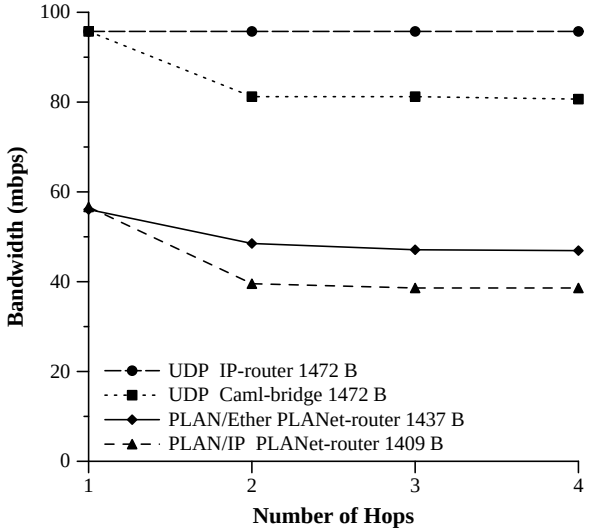

Figure 6: Bandwidth versus Number of Hops

packets per second over two. For maximum size packets, one hop can process 4900 packets per second, while the two hop case nets 4200 packets per second.

UDP routing over in-kernel IP routers achieves 95.6 Mbps regardless of the number of hops. This figure is 99.9% of the 95.7 Mbps possible for the 1472 bytes of useful data in our 1500 byte Ethernet payloads, indicating that the network is the bottleneck in this case. The C bridge also achieves this maximum throughput, but the OCaml bridge is a bottleneck and limits performance to a little more than 80 Mbps.

3.5 Performance Discussion

During the development of PLANet, we measured a number of aspects of our implementation to gain a better understanding of its performance. These measurements, and the enhancements that they motivated, resulted in increased peak bandwidth by about a factor of four. In this subsection, we present some of those measurements and the resulting improvements, as well as possible future improvements.

3.5.1 Bottleneck Overheads

Typically we studied the overheads in the PLAN router, which was the main bottleneck. Based on the measured throughput, we know that for large packets the router spends about 240 μs processing each packet.

One source of overhead is the need to cross the kernel boundary. The C bridge has almost no additional overhead beyond this one and can forward large packets at maximum rate. We added a delay loop to the C bridge and increased the delay until it started to act as a bottleneck. This happened when our added delay was 45 μs, resulting in an overall service time of 122 μs (based on observed throughput). Subtracting the added delay from the service time gives us an estimate of 77 μs for the two kernel crossings; Note this is very close to the estimate of 35 μs per crossing found for ping. Since the PLAN router makes exactly the same kernel calls to transport the same data (and in this test we actually bridged PLAN packets) it seems likely that this is a good estimate of the kernel crossing overhead for PLAN.

The next obvious measurement is the time the packet takes between the system calls for receiving and sending a packet. We measured this directly and found it to be in the range 135-150 μs. We are studying this receive-to-send path to see how it can be optimized. It appears that most of that path involves executing OCaml bytecodes, so compiling to native code might result in a substantial benefit. OCaml provides a native code compiler, which produces code claimed to typically be six times as fast as the interpreted byte-codes [8], but a bug in its thread system has prevented us from using it thus far.

3.5.2 Threads, Copying, and GC

During our efforts to improve PLANet's performance, tuning three areas had significant impact. First, we noted that calling the scheduler in OCaml's user-level threads system is expensive—as much as 100 μs per scheduling event. We rewrote some of the router to avoid thread hand-offs and eliminate almost all calls to the scheduler. Secondly, we observed that our initial implementation had a number of extra copies of the packet. We eliminated these and noticed not only the decreased copying costs, but a more significant decrease in the cost of GC (which would occur far less frequently). Finally, this improvement in the cost of GC caused us to investigate how setting key GC parameters might influence our results. Such adjustments allowed us to eliminate almost all GC overhead. The first two improvements are ones that would probably be needed in any prototype router implementation, while tuning the collector mostly required understanding the application rather than the collector.

3.5.3 Further Improvements

A number of possible improvements to the PLAN implementation are evident:

- **Lower the cost of PLAN evaluation.**
  This requires a more detailed analysis of the costs of the PLAN primitives and lowering these where possible. As mentioned, one potential win is to avoid unmarshalling and remarshalling programs to be evaluated and simply evaluate them in place. A feasible wire format for this effort might be bytecodes rather than abstract syntax trees.

- **Lower the cost of thread scheduling.**
  This might involve a more savvy reimplementation of the OCaml user-level scheduler, or instead a movement to a kernel-level scheduler. We are currently exploring both approaches.

- **Use native code for the PLANet 'core' and bytecode for loaded extensions.**
  This is not possible in the current OCaml implementation, but we are looking into the necessary modifications.

A variety other improvements are possible, and we expect to significantly enhance our performance as we refine our prototype.

4 Two Experiments with Active Internetworking

The previous section explored performance for simple end-to-end latency and bandwidth—tasks that the current Internetwork can do efficiently. However, we have not yet taken

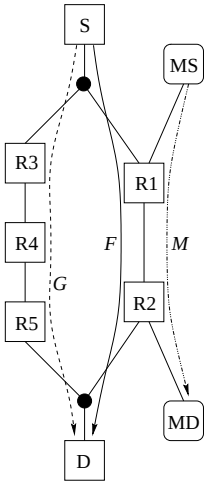

Figure 7: Experimental Topology

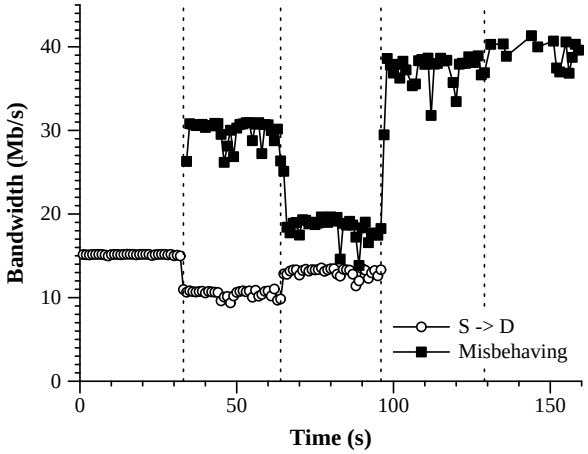

Figure 8: Dynamic Queue Modification

any real advantage of the fact that PLANet is an *active* network. This section shows how programmability can be used to create more flexible and better performing network functionality. We present one demonstration each for active extensions and active packets, both sharing the same basic scenario. The network topology is shown in Figure 7. Our protagonists are a source S and a destination D trying to maintain a certain transmission bandwidth. Using the default shortest hop-count routing, this flow F passes through routers R_1 and R_2. However, during the transmission, a misbehaving source MS begins sending to another destination MD along path M at a much higher bandwidth. This saturates the router R_1, degrading the throughput from S to D.

We will demonstrate two techniques which S might use to recover some of the lost bandwidth: using router extensibility at the active loader level, and using programmability at the packet level. It is important to note that it is not the *particular* algorithms or techniques presented here that we are attempting to demonstrate; rather, it is the *way* in which the algorithms are deployed that is significant. It seems impossible to anticipate all future demands on networking functionality and create a single global standard—the continued evolution of the Internet attests to this fact. Instead, it seems likely that having a programmable networking infrastructure will allow a variety of approaches to solving future problems. This section serves to illustrate how the programmability features we support can be used.

4.1 Modification of a Queuing Strategy

Active extensions allow us to employ a strategy in which the router is enhanced dynamically to improve the performance of the network. One router alteration would be to install a fairer queue. In the default PLANet implementation, there is a single, first-come, first-served (FCFS) packet queue shared by all devices. Thus as the combined traffic generated by S and MS exceeds the switching capacity of R_1, the switching capacity of the router will not be divided fairly between the two incoming packet flows. Rather, the generator which fills the queue more quickly will receive the majority of the bandwidth, while the more-well-behaved sender will receive proportionally less.

We constructed our demonstration as follows. S is attempting to send packets to R at a steady 15 Mbps. At some point, MS starts sending at the switching capacity, 40 Mbps[2]. Rather than allow S to continue at 15 Mbps and limit MS to 25 Mbps, the FCFS queuing system causes S to attain only 10 Mbps while MS achieves roughly 30 Mbps. This first two intervals of Figure 8 illustrate this situation.

To more fairly share bandwidth, we dynamically alter the queuing system used by R_1, resulting in the bandwidths depicted in the third interval of the figure. This is done by sending a PLAN program to R_1 from S that carries an active extension (in the form of OCaml bytecodes) to implement a fair queue. This queue consists of three queues, one for each of the devices on R_1, each one third the size of the original FCFS queue. Each queue is serviced in round-robin fashion and so each interface is guaranteed at least 1/3 the switching capacity. In fact, this is exactly what S sees after the switchover (about 13 Mbps), while MS falls to about 20 Mbps. The fair queue is not without drawback: the maximum attainable capacity through any given interface is also limited, due to the additional overhead of servicing multiple queues, and the potentially wasted buffer space. While both S and MS operate with the fair queue, the total link bandwidth is reduced to $20 + 13 = 33$ Mbps. In the fourth interval, when S stops sending, we see that MS increases its bandwidth to 38 Mbps, but it is not until the fifth interval, when the FCFS queue is dynamically reinstalled, that MS attains the maximum capacity of 40 Mbps. Note that the time to actually dynamically link in the queuing code is negligible—about 30 ms in our current implementation.

4.2 Selection of a Better Route

Let us re-examine our scenario. If we look at the topology shown in Figure 7, we see that there is an additional path G from S to D that passes through routers R_3, R_4, and R_5. This path is 4 hops long and so would not be used by a simple shortest-hop-count routing protocol like RIP. However, while MS is saturating R_1, it might well be worthwhile to take a longer route to avoid congestion.

The approach we take in this demonstration is to allow the sender S to "shop around" for the route with the best

[2]This number is lower than the 48 Mbps earlier reported due to the additional overhead of using the active loader.

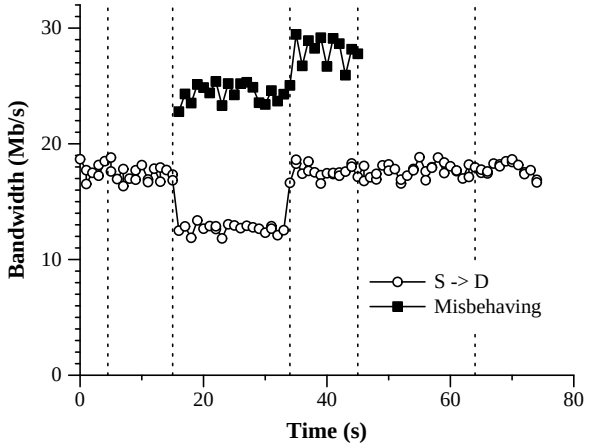

Figure 9: Flow-Based Adaptive Routing

available bandwidth. The bulk of the data will be carried in the same largely non-active *transport packets* that we used to measure bandwidth in Section 3. However, we will periodically intersperse *scout packets* that will explore the network searching for a better route and directing the flow of the transport packets.

Each scout packet fits within a 1500 byte Ethernet frame, yet carries out some non-trivial computations. In particular, at each hop, the scout packet will send a copy of itself on each of the router's outgoing interfaces, thus fanning out over the network. Unlimited fanout is prevented by the properties of the PLAN language presented earlier in Section 2; the global network resource usage is bounded by the resource bound found in the initial scout packet. All of the copies of the scout packet then take part in a distributed graph algorithm to discover the best available route. For our experiment here, we compute a rough path congestion metric based primarily on the queue length at each hop and secondarily on the total hop count.

The packets communicate with each other by leaving a small bit of state at each router which times out after half of the scouting interval. This state records the metric computed thus far, so that arriving packets which cannot better that mark can terminate their search. If a packet reaches the destination with the best metric so far, it generates a destination-unique flow ID and then steps backwards along its route. At each router on the return trip, the packet installs an entry in a flow-based routing table that maps the flow ID to the next hop. This results in routing quite similar to the VC-switching used in ATM networks [9]. Finally, the packet reports the flow ID to the controlling application, which can then start sending the transport packets along the route just set up by the returning scout packet.

The results of this demonstration are shown in Figure 9. S begins transmitting to D using the default RIP routing service, simultaneously sending out the first scout packet. For the particular trial shown here, the scout packet returns in 36 ms with the same route through R_1 and R_2. After a 5-second hysteresis to collect any other returning scout packets, the sender switches to the flow-based routing—note that the bandwidth is not noticeably interrupted. At time $t = 15$, we start a load generator on MS which begins sending on the link between R_1 and R_2 at 25 Mbps, and the perceived bandwidth at D drops to 12 Mbps. At time $t = 30$, the next scout packet goes out from S, and one of its descendants returns after 42 ms with a route through R_3, R_4, and R_5 which goes around the overloaded link. After the hysteresis interval (at $t = 35$), the sender begins using this new flow, bringing its bandwidth back up to 17 Mbps (note the small increase in perceived bandwidth at MD up to 28 Mbps). Finally, at time $t = 45$, the misbehaving host stops sending, and at the next reporting interval (time $t = 60$), a scout packet discovers that the original route is now usable again, and the flow reverts.

This technique, which we call Flow-Based Adaptive Routing (FBAR) illustrates a key aspect of the design space of the evaluation of active packets. As we saw in Section 3, evaluation can be costly. However, the majority of packets only require "passive" transport, and this simpler service can be performed much more efficiently than evaluation. Our view is that evaluated active packets are like the spice that makes the meal taste better: too much or too little yields less appetizing fare. An overall benchmark objective for active networks would be to provide switching for most packets with performance comparable to IP routers while achieving better overall performance by selective use of evaluated packets.

5 Related Work

PLANet is the first purely active internetwork. The reader is referred to [23] and the Active Network Program home page, `http://www.ito.darpa.mil/research/anets` for a listing of other active network projects.

Some non-active networking projects share implementation ideas with PLANet. The Fox project [4, 5] implements the TCP/IP protocol suite for Ethernet and ATM link layers using the ML programming language, in the approach of the x-kernel [17]. This work provides a number of insights into the challenge of implementing network software using a high-level language like ML. Their implementation uses SML/NJ rather than OCaml so there are some differences; in particular, our programs are byte-code interpreted and we make use of OCaml's dynamic loading. These aspects of OCaml have been exploited to provide mobile programs for the MMM browser [16]. OCaml has been used for other serious distributed systems. The Ensemble Project [24] provides a generalization to group communication of TCP/IP, and its (native code) OCaml implementation has a performance comparable to similar systems written in C.

Some work has been done concerning security in Active Networks. Most notable is the SANE [1] project (Secure Active Network Environment) which defines a general set of guidelines for trust relationships in an Active Network. An adaptation of this approach has been applied to PLANet [12].

There is a scarcity of information available about active networking performance, and so it is difficult to compare our results to others. Wetherall *et al.* [25] report a maximum packet forwarding rate for ANTS on a 167 MHz Ultrasparc over 100 Mbps Ethernet of 1680 packets per second for minimum size packets. This measurement uses a Java JIT, and represents about a 60% improvement over byte-code interpretation. For larger packets, over a slower link, Banchs *et al.* [3], measure rates of 3.8 Mbps for M0, and about half that rate for ANTS. Finally, Hartmann *et al.* [10] show that by using aggressive compilation and special purpose operating systems, they can reduce the latency of ANTS by about a factor of about 3.5.

Conclusion

PLANet is a concrete demonstration that building highly extensible and non-trivial active networks is feasible. Performance of the prototype is good enough to believe that active networking techniques will perform acceptably. We have also demonstrated how active extensions and active packets can be used to provide interesting active services. Papers about PLAN and PLANet and our freely available code distribution may be found at `http://www.cis.upenn.edu/~switchware/PLAN`.

References

[1] D. Scott Alexander, William A. Arbaugh, Angelos D. Kerom yts, and Jonathan M. Smith. A secure active network environment architecture: Realization in SwitchWare. *IEEE Network Magazine*, 1998. To appear in the special issue on Active and Controllable Networks.

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

[3] Albert Banchs, Wolfgang Effelsberg, Christian Tschudin, and Volker Turau. Multicasting multimedia streams with active networks. Technical Report 97-050, International Computer Science Institute, 1997.

[4] Edo Biagioni. A structured TCP in Standard ML. In *Proceedings, 1994 SIGCOMM Conference*. ACM, 1994.

[5] Edoardo Biagioni, Robert Harper, Peter Lee, and Brian G. Milnes. Signatures for a network protocol stack: A systems application of Standard ML. In *Proceedings of the 1994 ACM Conference on Lisp and Functional Programming*, pages 55–64. ACM, 1994.

[6] Scott O. Bradner and Allison Mankin, editors. *Ipng, Internet Protocol Next Generation*. Ipng Series. Addison-Wesley, 1996.

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

[8] Objective caml — questions and answers. `http://pauillac.inria.fr/ocaml/speed.html`.

[9] Martin de Prycker. *Asynchronous Transfer Mode: Solution for Broadband ISDN*. Ellis Horwood, West Sussex, England, 1991.

[10] John H. Hartman, Larry L. Peterson, Andy Bavier, Peter A. Bigot, Patrick Bridges, Brady Montz, Rob Piltz, Todd A. Proebsting, and Oliver Spatscheck. Joust: A platform for communication-oriented liquid software. Technical report, University of Arizona, 1997.

[11] C. Hedrick. Routing information protocol. Technical report, RFC 1058, June 1988.

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

[13] Michael Hicks, Pankaj Kakkar, Jonathan T. Moore, Carl A. Gunter, and Scott Nettles. Network programming with PLAN. In *Workshop on Internet Programming Languages*. Springer, 1998. To appear.

[14] Michael Hicks, Pankaj Kakkar, Jonathan T. Moore, Carl A. Gunter, and Scott Nettles. PLAN: A programming language for active networks. In *International Conference on Functional Programming (ICFP'98)*, 1998. To appear.

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

[16] Franc̀ois Louaix. A web navigator with applets in Caml. In *Fifth WWW Conference*, 1996.

[17] Sean W. O'Malley and Larry L. Peterson. A dynamic network architecture. *ACM Transactions on Computer Systems*, 10(2), May 1992.

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

[19] David C. Plummer. An Ethernet address resolution protocol. Technical report, IETF RFC 826, 1982.

[20] J. Postel. Internet control message protocol. Technical report, IETF RFC 792, September 1981.

[21] J. Postel. Internet protocol. Technical report, IETF RFC 791, September 1981.

[22] Smart packets home page. `http://www.net-tech.bbn.com/smtpkts/smtpkts-index.html`.

[23] 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.

[24] Robbert van Renesse, Ken Birman, Mark Hayden, Alexey Vaysburd, and David Karr. Building adaptive systems using Ensemble. Technical Report TR97-1638, Cornell University, 1997.

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

Appendix

ARP in PLAN

```
fun ask(l:blob,n:host,n':host): unit =
    let val ifc:dev = getSrcDev() in
      if (thisHost(ifc) = n') then
       let val l':blob = retrieveBinding(n',ifc) in
         OnNeighbor(|bind|(l',n'), n, getRB(), ifc)
       end
      else ()
    end
```

When broadcast by the requester, this function is invoked on each host on the local network using the arguments L, N, N'. Here, L and N represent respectively the link layer and network addresses of the requester. N' in turn, is the network address for which the requester would like to find the corresponding link layer address L'. On a given host, the program first determines the device over which the request was received, and then checks to see if it is evaluating on the host to whom the request is directed. If not, then the program terminates without action. Otherwise, the program should report a binding back to the requester. This is done by retrieving the link layer address L' and making a call to the function `OnNeighbor`, which differs from `OnRemote` by assuming that routing will not be necessary. This in turn calls the `bind` function *on the requester* with the arguments L', N' and this call places the desired binding into the table of the requester. This illustrative protocol differs from ARP in not attempting to modify the bindings of the responder if it is missing binding information about the requester. Our actual implementation more closely adheres to the ARP protocol [19].

Scout Packets

Below is the code that we used for our scout packets in Section 4. A detailed description of the service routines used herein may be found in the PLAN Programmer's Guide (XXX cite). Each packet begins evaluation on the initiating host with the PLAN function `startDFS` (the last function presented below) with the argument set to the node to which a route is desired. The initial program will spread out multiple packets which search the network, each computing a metric as it goes, finally collaborating to find the best route to take to a destination.

```
(* setupRoute: installs the flow route on the way back. arguments are:
     session  : controlling app's session--so scout packets from different
                applications do not interfere
     flowKey  : unique on destination--we will use this as the key under
                which to install the routes
     path     : the (host,metric) pairs corresponding to the path remaining
                to install. the first element of the list should correspond
                to the current host
     last     : the last host we visited; we will make the route we install
                point to him
     lastMet  : the metric value on that last host
     finalMet : what the overall route metric was *)

fun sR(session:key, flowKey:key, path:(host * int) list,
               last:host, lastMet:int, finalMet:int) : unit =
    if (not(thisHostIs(fst(hd(path))))) then
      () (* ended up on the wrong host somehow *)
    else
      try
          if (snd(hd(path)) <> get("metric",session)) then
            () (* a better route was discovered, so stop *)
          else
            (* install the route *)
            (flowSet(flowKey,fst(defaultRoute(last)),
                     lastMet,snd(defaultRoute(last)));
             if (tl(path) <> []) then
               (* more hosts to go *)
               OnRemote(|sR(session,flowKey,tl(path),
                                  fst(hd(path)),snd(hd(path)),finalMet)|,
                fst(hd(tl(path))), getRB(), defaultRoute)
             else
               (* report back to the controlling application *)
               (deliver(getImplicitPort(),(flowKey,finalMet))))
      handle NotFound => (* we've been gone so long our metric timed out! *)
        ()

(* cM : compute the metric on the current host, using the route metric so
        far as part of the computation *)
```

```
fun cM(lHM : int):int =
  lHM + getToll()

(* pN (process neighbors):
   stuff = (source,dest,path-so-far,rb-per-packet,session-key)
   neighbor = neighbor to send to *)
fun pN(stuff:(host * host * (host * int) list * int * key),
                neighbor: host * dev) : unit =

    (OnNeighbor(|dfs(#1 stuff,#2 stuff,#3 stuff,#5 stuff)|,
                #1 neighbor,#4 stuff,#2 neighbor);
     stuff)

(* does the brunt of the network search
   source = where the search originated
   dest = final destination to which we are trying to establish a route
   path = path-so-far (hops and metrics)
   session = per-application identifier *)
fun dfs(source:host, dest:host, path:(host * int) list,
        session:key) : unit =
  (* what's the metric so far? *)
  let val lastHM:int = try
                         snd(hd(path))
                       handle Hd => ~1
      val nM:int = cM(lastHM) (* metric at this hop *)
  in
    (* require that a smaller metric is better *)
    if (setLT("metric",session,nM,15)) then
      (* we're the best so far *)
      let val hostname:host = hd(thisHost()) in
        if (thisHostIs(dest)) then
  (* in fact, we have found the best route to the dest! *)
          let val flowKey:key = genKeyHost(dest) in
             (* get a destination-unique key *)
             try
               (* work backwards and set up the new route *)
       OnRemote(|sR(session,flowKey,path,
                                  dest,nM,nM)|,
                fst(hd(path)),getRB(),defaultRoute)
             handle Hd => (deliver(getImplicitPort(),(flowKey,nM)))
               (* was already at the source *)
          end
        else
  (* not at destination yet, keep searching *)
          foldl(pN,
                ((source,dest,(hostname,nM)::path,
          getRB() / length(getNeighbors()),session)),
                getNeighbors())
      end
    else
      (* someone better has already been through here, so we die *)
      ()
  end

(* initial call for base case *)
fun startDFS(dest:host) : unit =
  dfs(hd thisHost(),dest,[],getSessionKey())
```