Network Programming Using PLAN

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

Department of Computer and Information Science
University of Pennsylvania
200 South 33rd Street
Philadelphia, PA 19104-6389
{mwh,pankaj,jonm,gunter,nettles}@dsl.cis.upenn.edu

Abstract. We present here a methodology for programming active networks in the environment defined by our new language PLAN (Packet Language for Active Networks). This environment presumes a two-level architecture consisting of:

1. *active packets* carrying PLAN code; and
2. downloadable, node-resident *services* written in more general-purpose languages.

We present several examples which illustrate how these two features can be combined to implement various network functions.

1 Introduction

The Internet consists of separate networks of host computers that are interconnected by routers to form a homogeneous internetwork. General-purpose computation is done on hosts, possibly involving communication with other hosts in the internetwork, while routers are specialized to the task of moving packets between the networks. To do this, routers 'store and forward' packets to their 'next hop,' guided by information in the packet header, such as the destination address. An *active* network is one in which the routers go beyond this basic model and allow themselves to be programmed either by the packets passing through them or via some 'off-line' method. Such a network could have significant advantages if basic issues of security and performance are solved and a convenient programming model enables the creation of new or improved network services. Such services might be stored on the routers themselves, in the packets, or partly in the packet and partly on the router, and invoked when the packet is processed by the router. This enables entirely new ways of thinking about routing (for instance, a packet may not require an explicit destination address because its *evaluation* will determine its routing) and network protocols, which

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

can be supported by programming routers within the network on a per-user, per-connection, or per-packet basis.

We have developed a new Packet Language for Active Networks (PLAN) [12] whose programs are intended to be carried in packets and executed on routers (or hosts) through which the packets pass. PLAN is a simple scripting language that seeks to perform a number of functions in the active network:

1. Network management and configuration: customization of the use of network services by system administrators and applications.
2. Distributed communications: information exchange among applications and network elements.
3. Diagnostics: analysis of network status or behavior.

The natural question here is: why a new language? In short, PLAN was designed to fit within an architecture that makes solving the problems of security and performance more tractable. For instance, to protect the network while still making it available to all users, all PLAN programs must, by nature, terminate, and may not access information private to the router or other programs. PLAN was also designed so that its programs would be small enough to fit within network packets. No existing general purpose language meets these criteria while presenting a fundamentally distributed computational model. A more detailed analysis of this issue may be found in [12].

Because PLAN programs are limited in nature, they form only one part of our architecture. Additional functionality is available via calls to node-resident *services*, which may be written in general-purpose languages, such as Caml [6] or Java [8]. Some services may be available only to certain users and therefore may require authentication and authorization of the packets requesting their use. To provide full extensibility, service routines can be dynamically loaded into routers.

Together, PLAN packets and general-purpose services form a two-level active network architecture. Using this architecture, we have recently built an active internetwork called *PLANet* [14] in which all packets in the network contain PLAN programs. This paper illustrates how to program the network by giving examples of packet level and service level programs. We also illustrate an interesting active application which takes advantage of network programmability. We conclude by discussing future challenges and some related work.

2 Packet-Level Programming

PLAN programs are limited first in the amount of computation they can do on a router, and second in the number of routers on which the program can cause computation to occur. The programs are carried in packets and their evaluation may produce other packets, but this proliferation is controlled by the resource bound of the original packet, which must share its resource counter with its progeny. A PLAN program which only calls ‘safe’ services will evaluate on a given router in a predictable number of steps and then terminate without affecting the state of the router. PLAN packets do not communicate with one another directly at the PLAN level, although they can do so indirectly through application programs at endpoints or through service calls on routers, if such calls are permitted.

Ping

PLAN 3.1, our current version and implementation, is a small script-like functional programming language with a syntax similar to Standard ML whose basic programming model is a form of remote evaluation. How this is used is perhaps best illustrated with a small sample program that performs an active ping.

```
fun ping (source:host, destination:host) : unit =
  if thisHostIs(destination) then
    OnRemote(|ack|(), source, getRB(), defaultRoute)
  else
    OnRemote(|ping|(source, destination), destination,
              getRB(), defaultRoute)
```

where `ack` is a simple acknowledgment function. The program is invoked with a `destination` to be pinged and a `source` to receive the acknowledgment. Remote evaluation occurs with the invocation of the `OnRemote` primitive. The result of an evaluating `OnRemote` is that a new packet is created which evaluates some code on a remote node. This primitive takes four arguments:

1. The name of the function to be evaluated remotely and the arguments to give it (evaluated locally). This construct is termed a *chunk*, and may be manipulated as data by PLAN programs. Chunk literals resemble regular function applications except that the function name is surrounded by |'s, indicating that the evaluation of the function itself is delayed.
2. The name of the site where the evaluation is to occur.
3. The amount of resource bound to give to the remote evaluation (we will explain this later).
4. A routing function name.

If we invoke `ping` with a specific source and destination on a machine that is not the destination, the call will evaluate

```
    OnRemote (|ping|(source, destination), destination,
              getRB(), defaultRoute)
```

This means that `ping(source, destination)` should be invoked on `destination`. When this is done, the program will then evaluate:

```
    OnRemote (|ack|(), source, getRB(), defaultRoute)
```

which indicates that the acknowledgment function should be invoked on `source`. The `defaultRoute` argument determines how the `ping` and acknowledgment packets reach their destinations; it names a function used to determine the 'next hop' on each intermediate node on the way to the destination. Note that `defaultRoute`, `getRB`, and `thisHostIs` are all node-resident service functions expected to be present on each node in the network.

PLAN programs are guaranteed to terminate locally because of the limited nature of the language. PLAN does not allow (direct) recursive function calls[1], and there are no general looping constructs with which to encode infinite loops. Of course, it is assumed that the service functions available to non-authenticated PLAN programs will also respect the termination property.

PLAN programs by nature will terminate globally as well. This is because each time a new computation is initiated with `OnRemote`, some amount of resource bound from the current computation must be provided to the new one. The current computation's resource bound can be obtained via the call `getRB()`. Once the resource bound of a packet reaches zero, no further hops or evaluations may take place. There is no function in PLAN for increasing the resource bound, and it is decreased on arrival to each router, so the computation is guaranteed to complete and will not migrate around the network forever. To see this in a stark example, consider the following program:

```
fun ping_pong(pingHost:host, pongHost:host) : unit =
    OnRemote (|ping_pong|(pongHost, pingHost),
              pongHost, getRB(), defaultRoute)
```

This program will globally terminate after hopping between `pingHost` and `pongHost` for long enough to exhaust the resource bound.

Traceroute

The ping program is comparatively simple as network operations go and does not illustrate active networks especially well because its evaluation is on the endpoints of the ping. A more interesting function is `traceroute`, which is implemented in IP as a series of messages with increasing Time To Live (TTL) values. Each time the TTL value is exhausted (presumably one hop further away than the preceding message), an ICMP [16] packet is sent back to the source indicating the point of exhaustion. This algorithm has the characteristic that if the destination is unreachable due to a failure in the network, all of the nodes up to that point will be reported. Figure 1 is a PLAN analog, where `ack` prints the hop count and host reached.

The `OnNeighbor` primitive evaluates its first argument on the indicated `dest`, where that destination must be on the same physical network as the sender (*i.e.* it is one hop away). Therefore, the routing function argument is replaced by the name of the link layer device which accesses the shared network.

This implementation of traceroute is better than the way it must be implemented for the IP network: it will always provide information about a consistent route. The use of expiring TTLs to trace routers can provide information about

[1] In particular, a 'recursive' call like the call to `ping` within the body of its definition will result in a reduction in the resource bound by `OnRemote`.

```
fun traceroute (source:host, dest:host, count:int) : unit =

 (* First send response back to source that we got this far *)
 (OnRemote(|ack|(count,thisHost()), source, count, defaultRoute);

  (* If we're at the end, we're done *)
  if thisHostIs(dest) then ()

  (* Otherwise, proceed to the next hop *)
  else
    let val next:host*dev = defaultRoute(dest) in
      OnNeighbor (|traceroute|(source, dest, count+1),
                  fst next, getRB(), snd next)
    end
```

Fig. 1. Tracing a route in PLAN.

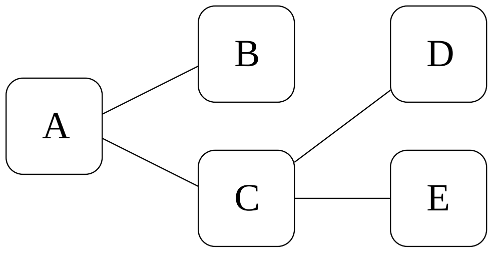

Fig. 2. A sample network topology

routers along more than one path between the source and destination if the routing topology changes mid-way.

Multicast

So far, our examples have been of diagnostic functions, but PLAN is also able to express more general network computations, such as multicast. The idea behind multicast is to trade off computation for bandwidth, so it is in keeping with the goals of an active network. Consider the topology depicted in Figure 2. If a program at node A were to send packets individually to nodes B, C, D and E, a total of 6 transmissions would occur: $A \rightarrow B$, $A \rightarrow C$, $A \rightarrow C \rightarrow D$, and $A \rightarrow C \rightarrow E$. A multicast packet takes advantage of common prefixes among destination nodes, resulting in only 4 transmissions: $A \rightarrow B$, $A \rightarrow C$, $C \rightarrow D$, and $C \rightarrow E$. However, the reduction in messages is compensated for by additional computation as the multicast tree must be computed by the routers.

Figure 3 illustrates a PLAN program which multicasts a computation to a list of destinations. The way the program works is as follows. Each time `multicast` evaluates, the local function `find_hops` is called for each node in the destination list `addrs` by the primitive `foldl` (explained below). The result is a pair containing the list of next hops in the multicast tree and a list of the remaining destinations. `multicast` is then invoked remotely on each hop parameterized by the new list of destinations. Note that `find_hops` also evaluates the designated task when the function is evaluating at a destination. This is done via a call to `eval` on the *chunk* `task`. Recall that a chunk is an encapsulation for a function name and a list of values, and is supplied as the first argument to `OnRemote` or `OnNeighbor`. `eval` simply resolves the name present in the chunk with the current environment and then evaluates it, subtracting 1 from the resource bound, as with `OnRemote`.

`foldl` is based on the well-known functional programming construct `fold`. Since PLAN does not (currently) support language-level parametric polymorphism or higher-order functions[2] the list iterators are provided as language primitives, rather than as proper functions. The meaning of

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

is

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

where f is the name of either a PLAN or service function.

3 Service-Level Programming

In each of our example programs to this point, the majority of the functionality is expressed within the PLAN program. The calls to services have been relatively simple: determining the local host address with `thisHostIs`, querying the remaining resource bound with `getRB`, etc. However, it is inconvenient or impossible to do most network programming entirely at the packet level. More generally, we envision the packet language as a 'glue' language for calls to programs at the service level. An interesting question to network programmers using the PLAN system is how to partition the functionality of their protocol between the services and the packets. This is best explained by illustration, using a networking protocol that we implemented for address resolution in PLANet.

To support universal addressing, internetworks impose a uniform virtual addressing scheme on top of the varying addressing schemes provided by the underlying link layers. Sending data requires a link-layer address, and so internetworks need a mechanism to resolve network addresses into link layer ones. The process of obtaining such an address is called *address resolution.*

The Address Resolution Protocol (ARP [15]) works as follows. The sender, which has network address n and link layer address l, knows the network address

[2] Some services, such as `print`, do support these features.

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

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

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

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

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

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

Fig. 3. Packet-directed multicast PLAN.

m of some node on its local area network and desires to acquire the link layer address k of m. It therefore *broadcasts* onto its LAN a packet which asks the question "(n,l) is asking who out there has network address m?" The node with address m responds with its link layer address k, which n stores in its ARP table for future use. In addition, m stores the binding l for n as contained in the packet. Beyond this basic exchange there is one additional bit of policy: every node other than m which receives the broadcast request packet checks its own ARP table for the binding of the n (the requester). If such a binding exists, then it is updated by the binding l contained in the packet.

Clearly, some elements of this protocol cannot be expressed by PLAN alone. In particular, each node must store and retrieve network address to link layer address bindings from a table. The fact that this table is long-lived and must be accessible from PLAN programs prevents it from being itself expressed as a PLAN program.

A very basic implementation of this protocol would have n broadcast the following PLAN program:[3]

```
fun ask(l:blob,n:host,m:host): unit =
  doARP(l,n,m)
```

In this implementation, all of the functionality of ARP is contained within the service call `doARP`: the check to see if the packet arrived at m, the sending of a response packet (itself a similar looking PLAN program), the storing of the binding for n, etc. In this implementation, PLAN packets are essentially being used as a vehicle for distributed communication. In fact, this function for PLAN programs is part of the advantage of the PLAN system—there is no need for special-purpose packet formats since the packet format is defined *implicitly* by the standard wire format of PLAN programs. In general, new network functionality may be provided entirely by the introduction of new service functions that are called from PLAN packets passing through the active routers. This is essentially the approach of ANTS [21].

Of course, implementing a protocol in this way does not take advantage of the programmability of PLAN packets. In particular, much of the policy of ARP can be encoded within PLAN rather than within a service. This allows the protocol implementor to present a more granular, flexible service interface which can then be used by PLAN programs implementing different policies. For example, we might imagine that some versions of ARP would not perform the binding of the requester on a node other than the destination.

As mentioned above, the only thing that really needs to be encoded as a service is the table itself. In our ARP implementation we provide the services:

$$\begin{aligned} bind &: blob * host \to unit \\ retrieveBinding &: host \to blob \end{aligned}$$

[3] Note that since there is no specific type for link layer addresses, they are represented by the PLAN type `blob` which represents arbitrary data.

where `bind` adds a binding to the ARP table (raising the exception `BindFailed` if the table is full), and `retrieveBinding` looks up a binding (raising the exception `NoBinding` if the lookup fails). Given these services, the ARP packet is written as follows:

```
fun ask(l:blob,n:host,m:host): unit =
  try
    if thisHostIs(m) then
      (try
         bind(l,n)
       handle BindFailed => ();
       OnNeighbor(|bind|(retrieveBinding(m),m), n,
                  getRB(),getSrcDev()))
    else
      let val pa:blob = retrieveBinding(n) in
        bind(l,n)
      end
  handle NoBinding => ()
```

When broadcast by the requester, this function is invoked on each host on the local network. The program first checks to see if it is evaluating on m. If so, the program should report a binding back to the requester. This is done by retrieving the link layer address k with `retrieveBinding` and remotely invoking `bind` on the requester n via `OnNeighbor`.[4] If the program evaluates on a node other than m (the `else` case), then it checks the ARP table for an entry for n. If a binding is present, it is replaced by the one in the packet; if no binding is present (as indicated by an exception being thrown) then no action is taken.

ARP is a simple example, but it illustrates well how functionality may be partitioned between PLAN and the services. Here, the reason for adding more functionality to the PLAN part is the enhanced overall flexibility. The drawback is that with added flexibility comes potentially increased risk. The solution here is to either mitigate the risk by requiring calls to services to be authorized (as is done in some versions of current Internet routing protocols), or by limiting their interface. Another drawback is the overhead of carrying the policy code in the packet. This is isn't a problem for ARP, but could be for packet flows in which the code is essentially constant and only the bindings change.

4 Active Applications

Thus far we have seen examples of network functionality which are all provided in some form by the current Internet. However, the PLAN programming language environment provides us with more flexibility than does IP; this section presents an example of a performance improvement made possible by this extra

[4] The service routine `getSrcDev` returns the link layer device through which the evaluating packet was received.

programmability. In particular, we will show how an application can adapt to network congestion by using active packets to route data packets intelligently.

Consider the network topology shown in Figure 4, and suppose we have an application trying to send data from a source S to a destination D. Usual shortest-hop-count routing metrics, such as the one employed by RIP [10], would send traffic along the route $S \rightarrow R_1 \rightarrow R_2 \rightarrow D$. Now suppose, however, that the hosts X and Y begin communicating at a high rate, thus saturating the link $R_1 \rightarrow R_2$; the bandwidth from S to D will be severely reduced.

Notice, though, that there is another path $S \rightarrow R_3 \rightarrow R_4 \rightarrow R_5 \rightarrow D$ which is unused. Unfortunately, RIP-like protocols will not see this extra route, even though there may be more than enough spare bandwidth to satisfy the application running at S and D. If the application could detect this alternate route and use it, however, the improvement to bandwidth would be well worthwhile.

PLAN's flexibility permits the application to make this adjustment by using a routing scheme called *flow-based adaptive routing*. The application periodically sends out "scout packets" which explore the network searching for good paths. These scout packets perform a distributed graph computation to find the best currently available route, and then adjust state at the routers to reflect the discovered route, thus establishing a *flow*. The application then simply needs to send regular transport packets along the flow, with scout packets periodically sprinkled in to keep the flow up-to-date.

Figure 5 shows the full PLAN code for a sample scout packet, but we will only highlight some of the key portions here. In this example, the scout packet and its descendents will perform a depth-first fanout to find the least congested path, which we roughly quantify as a combination of the length of the intermediate routers' input queues and their distance in hops from the source:[5]

```
fun computeMetric(lastMetric : int):int =
  lastMetric + (10 * getQueueLength()) + 1
```

The main workhorse of the algorithm is the `dfs` function. At each node, an incoming packet computes its metric to the current node with a call to `computeMetric`. It then atomically compares its metric with the best metric found so far to the current node by accessing some soft state that other packets may have left behind:

```
setLT("metric",session,newMetric,15)
```

(the 15 here indicates the number of seconds which should elapse before the soft state is reclaimed by the node). If the provided metric is less than the old one then the check succeeds and the old value is replaced with the new one; if the check fails, some other packet has been at the node previously with a better route, so the current packet terminates. On success, the depth-first search continues, with the current packet spawning calls to `dfs` on all the neighboring nodes by using the `foldl` construct:

[5] Note that the constant 10 here is somewhat arbitrarily tuned to be effective in our experiments.

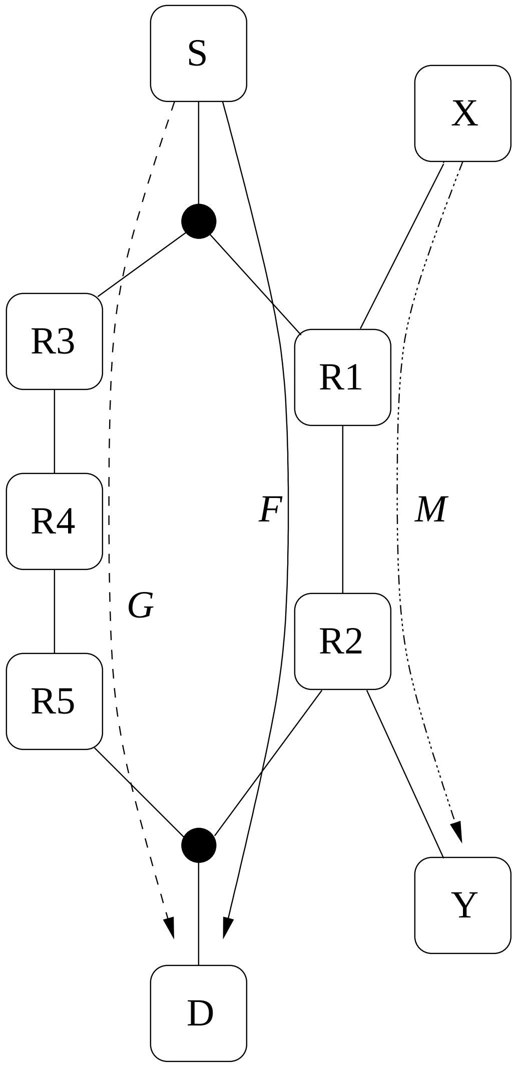

Fig. 4. A network topology with alternate paths

```
fun computeMetric(lastMetric : int):int =
  lastMetric + (10 * getQueueLength()) + 1

fun setupRoute(session:key, flowKey:key, path:(host * int) list,
               last:host, lastMet:int, finalMet:int) : unit =
    if (not(thisHostIs(fst(hd(path))))) then ()
    else
      try
          if (snd(hd(path)) <> get("metric",session)) then ()
          else
            (flowSet(flowKey,fst(defaultRoute(last)),
                     lastMet,snd(defaultRoute(last)));
             if (tl(path) <> []) then
               OnRemote(|setupRoute|(session,flowKey,tl(path),
                                    fst(hd(path)),snd(hd(path)),finalMet),
                fst(hd(tl(path))), getRB(), defaultRoute)
             else
               (deliver(getImplicitPort(),(flowKey,finalMet))))
      handle NotFound => ()

fun processNeighbor(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)

fun dfs(source:host, dest:host, path:(host * int) list,
        session:key) : unit =
  let val lastHopMetric:int = try
                                snd(hd(path))
                              handle Hd => ~1
      val newMetric:int = computeMetric(lastHopMetric)
  in
    if (setLT("metric",session,newMetric,15)) then
      let val hostname:host = hd(thisHost()) in
        if (thisHostIs(dest)) then
          let val flowKey:key = genKeyHost(dest) in
             try
               OnRemote(|setupRoute|(session,flowKey,path,
                                     dest,newMetric,newMetric),
                fst(hd(path)),getRB(),defaultRoute)
             handle Hd => (deliver(getImplicitPort(),(flowKey,newMetric)))
          end
        else
          foldl(processNeighbor,
                ((source,dest,(hostname,newMetric)::path,
                  getRB() / length(getNeighbors()),session)),
                getNeighbors())
      end
    else ()
  end
```

Fig. 5. PLAN code for a scout packet

```
foldl(processNeighbor,
      ((source, dest, (hostname,newMetric)::path,
        getRB() / length(getNeighbors()), session)),
      getNeighbors())
```

where `processNeighbor` creates a packet with `OnNeighbor` to do a recursive call to `dfs` on a single neighboring node. Note that the resource bound of the current packet, retrieved with the call to `getRB`, is divided equally among the child packets.

Finally, we reach the case where a packet arrives at the destination with the best metric. In this case, the packet is responsible for establishing the flow for the later transport packets to follow. This is done by creating a globally unique key and then calling `setupRoute` to work backwards. At each node on the way back, the packet adjusts the routing table:

```
flowSet(flowKey, dest, lastMet, nextHop)
```

Thus, given a transport packet with a flow identifier `flowKey` and destination `dest`, the router will forward the packet on to `nextHop`, after decrementing its resource bound. The final step of the protocol occurs when the setup packet reaches the source: it reports the new `flowKey` to the controlling application via the `deliver` construct. The application may then begin sending transport packets with the new flow identifier.

Using this approach in our experimental testbed we achieved two favorable ends. First, the bandwidth between S and D was raised to the application's desired level. Second, the overall utilization of the network increased: where originally path G was idle and paths F and M were congested, we transitioned to G and M both being utilized at the desired levels. Details are presented in [14].

This approach provides a "free market" style to quality of service in a best-effort network. Of course, there is nothing to prevent vendors from writing router software which implements this particular adaptation. However, note that it is easy to change the metric which determines the best route: one can simply redefine the `computeMetric` function in an appropriate way. Different metrics do not require new router software, so long as the nodes present appropriate diagnostic information through services such as `getQueueLength`, and even these can be straightforwardly added by downloading new service implementations. Furthermore, applications may set metrics which are *individually* appropriate, rather than having the infrastructure impose a "one size fits all" routing topology. As the Internet evolves and more and more different types of applications come into use, this style of flexibility will be invaluable.

5 Related Work

There are a number of active network projects, each of which provides its own programming model:

- The *Active Bridge* [5] is part of the SwitchWare Project [19] at the University of Pennsylvania, and provides a study of the kinds of services that PLAN is meant to provide glue for. It is an extensible software bridge where OCaml [6] services can be added to provide new functionality, including dynamic service cutover between different network topology standards. Work on the Active Bridge evolved into ALIEN [1]. In ALIEN, packets can be programmed in a general-purpose programming language (Caml). This requires more heavy-weight security (based on SANE [4]) on a per-packet basis than PLAN, as reported in [2].
- *Sprocket* is a language from the Smart Packets project at BBN [17]. It uses a byte-code language and is similar to PLAN in sharing a design goal to provide flexible network diagnostics. It differs from PLAN in that it targets a fixed set of service routines, and is designed with maximum compactness in mind.
- *ANTS* [21] is a toolkit for deploying Java protocols on active nodes. Packets carry hash values that refer to a service. Services are demand loaded if they are not present on the node. This model provides much less flexibility at the packet programming level than PLAN.
- *PLAN-P* [20] is a modification of PLAN to support programming services rather than packets. The PLAN-P work focuses on studying the use of optimization techniques based on partial evaluation to provide fast implementations of these service routines.
- *JRes* [7] is a system that concerns itself with resource bounding in Java. While it is not aimed at active networking directly, it is likely that some of its resource bounding techniques will be applicable in this domain.

There are a variety of projects related to networks, distributed computing, and operating systems that are related to PLAN's philosophy of active networks. Other related projects include Liquid Software [9] and Netscript [22]. The reader is referred to the SwitchWare architecture paper [3] for more details.

6 From the Internet to PLANet

PLAN was developed to replace the inflexible network-layer of the current Internet. However, there is something to be said for the real problem of upgrading an inactive network to an active one. A hybrid approach is feasible: PLAN programs can be used to customize the operations of a traditional network. This is suggested by our example in Section 4; active programs are used to discover and establish custom routes that may be traversed by inactive packets. A generalization of this idea is that of a Active Router Controller [18] in which active nodes are used to administer the control-plane (i.e., for routing, configuration, etc.) while switches are used for packet-forwarding. As fast-path operations can be made more efficient in an active context, more functionality may be made fully programmable.

There are still many areas in our current architecture that require more study. These roughly fall into the categories of security, scalability, and performance.

Security. While it is true that all PLAN programs terminate both locally and globally, such a guarantee may not be enough. In particular, one can write PLAN programs that execute in time and space exponential in their size. With some loss of *a priori* reasoning, we can strengthen this bound to be linear, using some straightforward alterations to the language, including one presented in [12]. We are currently researching the effects of such alterations and whether they are, in fact, enough, or whether further restriction is necessary. Additional security mechanisms for PLANet are considered in [11] and [13].

Scalability. So far, we have only experimented with small topologies, thus far avoiding problems of scalability. One area of needed improvement is in the lack of organization of service namespaces. Currently, all services exist in a flat namespace with the effect that newly installed services shadow older ones with the same name. While this is a useful property, since changes in technology or circumstance might necessitate upgrades of the service library, namespace organization is essential to make this process less *ad hoc*. Namespaces may also be used as a source of security, as described in [13]. Abstractions like Java packages or ML modules present possible solutions to this problem. It should also be noted that while the services suffer from this problem, PLAN programs themselves do not, as they are self-contained and may not interfere with each other.

Performance. Our performance experiments, as reported in [14], are promising: running in user-mode with an implementation written in byte-code-interpreted OCaml, we are able to switch active packets at around 5000 packets per second, up to a rate of 48 Mbps. This constitutes just over half of the useful bandwidth of our 100 Mbps Ethernet. However, as reported in that paper, there is much room for improvement. In particular, our scheduling strategy, wire representation, and evaluation strategy drastically affect performance. On the other hand, we have found that by designing PLAN to be authentication-free we can prevent costly cryptographic operations [2]. Furthermore, by including support for routing in the language (through the use of routing functions, like `defaultRoute`), we can avoid costly evaluations on every hop. Together, these optimizations represent up to a four-fold increase in performance.

For More Information. We invite readers to browse the PLAN home page:

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

The site provides downloadable software for both Java 1.1 and OCaml, detailed documentation, and a number of technical papers covering the issues presented here in greater length.

References

1. D. S. Alexander. *ALIEN: A Generalized Computing Model of Active Networks*. PhD thesis, University of Pennsylvania, September 1998.

2. D. S. Alexander, Kostas G. Anagnostakis, W. A. Arbaugh an d A. D. Keromytis, and J. M. Smith. The Price of Safety in an Active Network. Technical Report MS-CIS-99-02, University of Pennsylvania, January 1999.
3. D. Scott Alexander, William A. Arbaugh, Michael Hicks, Pankaj Kakkar, Angelos Keromytis, Jonathan T. Moore, Carl A. Gunter, Scott M. Nettles, and Jonathan M. Smith. The SwitchWare Active Network Architecture. *IEEE Network Magazine*, 12(3):29–36, 1998. Special issue on Active and Controllable Networks.
4. D. Scott Alexander, William A. Arbaugh, Angelos D. Keromytis, and Jonathan M. Smith. A Secure Active Network Architecture: Realization in SwitchWare. *IEEE Network Special Issue on Active and Controllable Networks*, 12(3):37–45, May/June 1998.
5. D. Scott Alexander, Marianne Shaw, Scott M. Nettles, and Jonathan M. Smith. Active Bridging. In *Proceedings, 1997 SIGCOMM Conference*. ACM, 1997.
6. Caml home page. `http://pauillac.inria.fr/caml/index-eng.html`.
7. Grzegorz Czajkowski and Thorsten von Eicken. JRes: A Resource Accounting Interface for Java. In *Proceedings of 1998 ACM OOPSLA Conference*, 1998.
8. James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification*. Addison Wesley, 1996.
9. John Hartman, Udi Manber, Larry Peterson, and Todd Proebsting. Liquid Software: A New Paradigm for Networked Systems. Technical report, Department of Computer Science, University of Arizona, June 1996. `http://www.cs.arizona.edu/ liquid`.
10. C. Hedrick. Routing Information Protocol. RFC 1058, June 1988.
11. Michael Hicks. PLAN System Security. Technical Report MS-CIS-98-25, Department of Computer and Information Science, University of Pennsylvania, April 1998.
12. Michael Hicks, Pankaj Kakkar, Jonathan T. Moore, Carl A. Gunter, and Scott Nettles. PLAN: A Packet Language for Active Networks. In *Proceedings of the Third ACM SIGPLAN International Conference on Functional Programming Languages*, pages 86–93. ACM, 1998. Available at `http://www.cis.upenn.edu/ ~switchware/ papers/ plan.ps`.
13. Michael Hicks and Angelos D. Keromytis. A Secure PLAN. In *International Workshop on Active Networks*, 1999. Submitted; available at `http://www.cis.upenn.edu/ ~switchware/ papers/ secureplan.ps`.
14. Michael Hicks, Jonathan T. Moore, D. Scott Alexander, Carl A. Gunter, and Scott Nettles. PLANet: An Active Internetwork. In *Proceedings of the Eighteenth IEEE Computer and Communication Society INFOCOM Conference*. IEEE, 1999. To appear; Available at `http://www.cis.upenn.edu/ ~switchware/ papers/ plan.ps`.
15. David C. Plummer. An Ethernet Address Resolution Protocol. Technical report, IETF RFC 826, 1982.
16. J. Postel. Internet Control Message Protocol. Technical report, IETF RFC 792, September 1981.
17. B. Schwartz, A. Jackson, T. Strayer, W. Zhou, R. Rockwell, and C. Partridge. Smart Packets for Active Networks. In *IEEE OPENARCH*, New York, New York, March 1999.
18. J. M. Smith, D. S. Alexander, W. S. Marcus, M. Segal, and W. D. Sincoskie. Towards an Active Internet. Available at `http://www.cis.upenn.edu/~switchware/papers/arc.ps`, July 1998.
19. SwitchWare project home page. `http://www.cis.upenn.edu/ ~switchware`.
20. Scott Thibault, Charles Consel, and Gilles Muller. Safe and Efficient Active Network Programming. In *17th IEEE Symposium on Reliable Distributed Systems*, 1998.

21. David J. Wetherall, John Guttag, and David L. Tennenhouse. ANTS: A Toolkit for Building and Dynamically Deploying Network Protocols. In *IEEE OPENARCH*, April 1998.
22. Y. Yemini and S. da Silva. Towards Programmable Networks. In *IFIP/IEEE International Workshop on Distributed Systems: Operations and Management*, L'Aquila, Italy, October 1996.