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

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 OCaml, a dialect of ML.

Currently, PLANet routers run as byte-code-interpreted Linux user-space applications, and support Ethernet and IP as link layers. PLANet achieves respectable performance on standard networking operations: 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 the utility of PLANet's activeness by showing experimentally how it can non-trivially improve application and aggregate network performance in congested conditions.

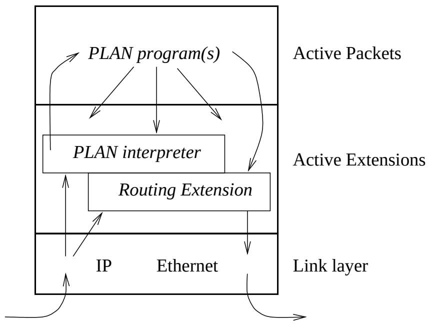


Fig. 1. PLANet node architecture

I. 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 programmable and extensible, or *active*, networks.

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 compromise its performance and safety? Finally, how can applications and service providers benefit from the network's expanded capabilities? In this paper we address these questions experimentally using a new internetwork, PLANet.

PLANet is indeed an internetwork: it implements network layer services directly on top of link layer technologies—without relying on the existing IP [1] infrastructure. PLANet is also 'purely active': *all* packets contain programs written in a special-purpose packet language called *PLAN* (Packet Language for Active Networks) [2], and nodes may be extended by dynamically loading additional code to change functionality [3]. PLANet is the first such purely active internetwork.

Let us consider the two ways in which PLANet is active. Firstly, PLANet uses *active packets* which contain PLAN programs, as mentioned above. 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 higher 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 *service routines* that provide general-purpose functionality not expressible in PLAN alone. Such service routines may provide simple information such as the address of the current host, or more substantial functionality, such as segmentation and reassembly.

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

Secondly, PLANet nodes may be programmed by dynamically loading *active extensions* written in OCaml [4], a dialect of the ML programming language. These extensions add to or enhance existing functionality, and may provide services available 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 Fig. 1. First, a packet arrives through the link layer interface. 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.

These design decisions flexibly position PLANet along two design-space axes that are important for active networking. The first axis spans possible mechanisms for activeness—from programmability at the packet level to extensibility at the node level. PLANet uses a mixture of these approaches in which packets carry programs that may refer to and invoke more general (and loadable) node-resident functionality. The second axis spans possible locations for active evaluation—from endpoints-only (like the Internet) to every-hop. Again, PLANet adopts a more flexible position that allows evaluation at *some* of the intermediate hops. Other active networking projects have only explored specific points on these axes (we defer a discussion of this related work until Section VI).

In measuring the performance of PLANet, we examined the costs of common, non-active network operations: we present both latency and bandwidth measurements for 'inactive' packets (whose only program is to deliver a payload) and compare the results to those of a kernel-based IP implementation. These measurements place an upper-bound on the overhead of using our active network architecture/implementation as compared to a nonactive one. We also demonstrate 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 show that programmability can yield potential performance gains for problems which are not likely to have a single best-fit solution.

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

II. PLAN

Much of the design of PLANet is based upon the language of its packets, the Packet Language for Active Networks, or PLAN. Here we highlight the salient features of the PLAN language, but we refer the reader to [2] for more details. PLAN is a small language that has elements in common with Haskell [5], Scheme [6], and ML [7], [4]. It differs most importantly from these in that it includes primitives for evaluating an expression at a remote node; invoking such a primitive will result in a newly spawned packet. Another special characteristic of PLAN is a resource-limited semantics that ensures that PLAN programs always terminate and that packets and their descendents visit only a fixed number of nodes.

PLAN was designed to be flexible enough to write useful programs, but limited enough that its programs will not pose a security risk. By contrast, the service routines available to PLAN programs are general-purpose and may need to be protected by cryptography or other means. (Here our coverage of security issues is limited due to space considerations. For more details, we refer the reader to [8] and [9].)

Some functions we have implemented as PLAN programs include 'route scouting' which seeks out low-congestion routing paths (as described in Section V), source-directed multicast, traceroute, and network-DFS, to name a few. More details about programming with PLAN may be found in [10].

III. PLANet

PLANet is our active internetwork implementation based on the PLAN environment. Whenever possible, we have drawn from the experience of IP [1], [11] in making implementation decisions, so as to leverage the experience represented by that design. Practical experience with the Internet indicate that the following core elements are required of 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 elements 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 core elements, 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, and network management, among others. Indeed, it is the flexibility of being able to augment active nodes with such 'nonstandard' 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 V. 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.

The following three subsections expand on our implementation of the core internetworking elements in PLANet as defined above. We begin with a discussion of our packet formats and addressing scheme, followed by our routing methodology, finishing up with a discussion on error handling and diagnostics.

A. 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 marshalling scheme for PLAN programs. This greatly simplifies the task of the implementor of a new network service, who only needs to be concerned with *what* information needs to be communicated, not with *how* that information is encoded.

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. An extreme view of active networks might allow packets to have only as much formatting as required to unmarshal a program for evaluation, preventing such routing optimizations. While more flexible, we feel that the cost in performance of this approach is too high.

The PLAN packet format is illustrated in Fig. 2 and described

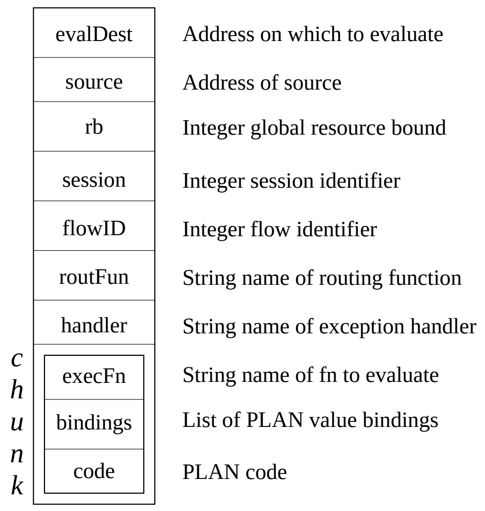

evalDest
Address on which to evaluate
source
Address of source
rb
Integer global resource bound
session
Integer session identifier
flowID
Integer flow identifier
routFun
String name of routing function
handler
String name of exception handler
c h u n k
execFn
String name of fn to evaluate
bindings
List of PLAN value bindings
code
PLAN code

Fig. 2. PLAN Packet Format

below. We begin by discussing PLANet addresses, followed by an explanation of the 'code' section of the packet (referred to as a *chunk*), and finally a summary of the remaining fields.

A.1 Addressing

The first two fields of the packet indicate the *evaluation destination* (*evalDest* for short), which is where the packet's program should be evaluated, and the *source*, which names the packet's oldest ancestor, used for error reporting.

Currently, PLANet uses 48 bit addresses, assigning 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 [12]. 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 routine like any other.

A.2 Programs as Data

The last three fields in the packet comprise the *chunk* (short for 'code hunk'), which is essentially its program: marshalled code, a function entry point, and initial arguments. The arguments are PLAN values that have standard marshalling formats; a PLAN programmer need not be concerned with representation issues. When a PLAN packet reaches its *evalDest*, its code is unmarshalled, and the appropriate function is located and called with the given values.

While requiring all packets in the network to be marshalled PLAN programs simplifies the question of packet formatting, there is a complication: what if the size of a marshalled chunk exceeds the path MTU? We address this within PLAN by allowing chunks to be manipulated as data. Notably, chunks may carry other chunks as data, providing an elegant encapsulation mechanism. Furthermore, with appropriate service routines, chunks can be fragmented, transported, reassembled, and then evaluated.

A.3 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 is a resource bound *rb* 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 the QoS routing example in Section V. 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 *handler* field provides the name of a service routine on the *source* that will handle certain error communications the router may choose to offer. Error reporting is described in more detail in Section III-C.

B. Routing

To determine the 'next hop' of a PLAN packet, an intermediate node evaluates the packet's specified *routFun*, which names a service routine that is resident on the intermediate node. We chose this approach as a means to explore the space between two extremes. At one extreme, we might require 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. At the other extreme, we might 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. 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 unmarshalled and/or evaluated. The only additional cost is the lookup of the routing function itself.

Most packets specify the `defaultRoute` routing function, which in PLANet is implemented with a simple distance vector routing service based closely on RIP [13]. On the other hand, more savvy applications may make use of customized routing functions, as illustrated in Section V.

C. 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 been exhausted. 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 [14]. The fundamental difficulty is that only a fixed set of errors or diagnostics can be reported until a restandardization 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 (such as reachability) or query it directly (perhaps to find the current packet queue length). If a service routine to provide 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.

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 program-level as opposed to packet-level. Errors that arise during program evaluation may be handled directly by the program, while errors occurring en route may be handled at the source by the service routine named in the packet's *handler* field. This ability to flexibly handle and diagnose errors will become increasingly important as the Internet becomes more widespread and harder to change.

D. Implementation Details

PLANet is implemented in OCaml v2.00 [4] in user space under the Linux kernel version 2.0.30. Our implementation is in part based on the Active Loader, described in [3]. The choice of implementing in user-space was largely motivated by expediency, 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 secure, web-based mobile code [15], [16]. In addition, its source code is freely available, making it convenient for us to make the necessary modifications to provide direct access to our networking hardware.

IV. PLANet Performance

This section addresses the second question posed in the introduction: can an active network attain reasonable performance, especially for common operations? 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.

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.

A. 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.

In order to determine upper bounds on the possible routing performance in user-space we wrote two 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, but has the additional overhead of interpreting OCaml bytecodes.

B. 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 V are the same, but only have 64 MBs of EDO memory. In all cases the

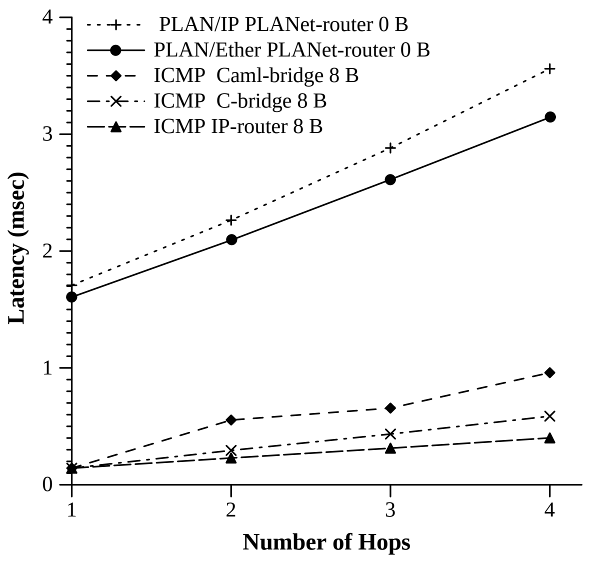


Fig. 3. Latencies for Switching Alternatives

machines have enough memory that they do not page fault. For Section V, 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 V 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 all presented measurements, the standard deviation and is less than 5% of the mean, but we do observe somewhat skewed distributions. For this reason, as recommended by Jain [17], we report medians, since they are less sensitive to influence by skewed distributions. Quartiles are not presented on the graphs to enhance readability. All times reported are elapsed times, and are measured with a clock having a 4 μs resolution.

C. 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.

Fig. 3 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.

In the following discussion, we consider the overhead due to kernel crossings, look at the node switching costs, and finally consider the overhead imposed by per-hop evaluation.

C.1 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[1]. Since both the PLAN ping application and the local 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 to be about 35 μs.

C.2 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 I 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 lowest per-byte costs. Since

[1] PLAN ports are a means of communication between the local interpreter and an application, implemented by Unix domain sockets. See [2] for more details.

TABLE I

Switching overheads

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

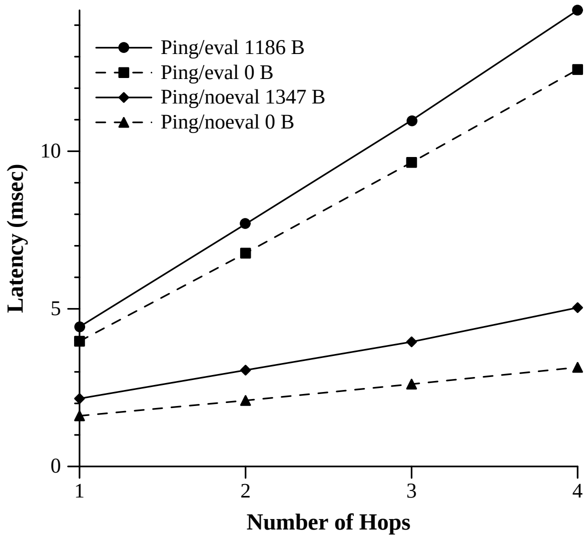


Fig. 4. Evaluation Overheads

the two bridges and the PLAN router do no copying internally, and make the same system calls, it is to be expected that all 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. Therefore Amdahl's law indicates that PLANet will require more optimization before moving it into the kernel will have a significant impact.

C.3 Ping with intermediate evaluation

We also wanted to find out the overhead of PLAN program evaluation. Fig. 4 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 additional code space overhead: 314 bytes per packet as opposed to 153 bytes for ping/noeval. For a fixed Ethernet packet size[2], 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.

[2]Note the graph shows usable payload sizes; 1186 bytes for ping/eval and 1347 bytes for ping/noeval both result in the same size Ethernet frame.

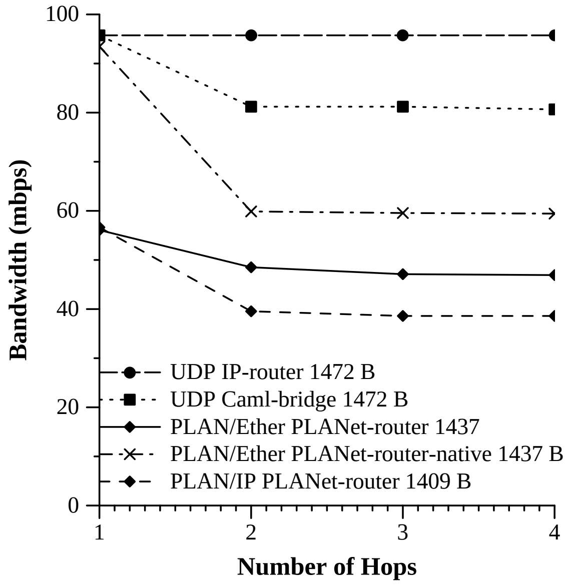


Fig. 5. Bandwidth versus Number of Hops

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 0.04 μs (using unrounded original values). This is clear evidence that avoiding evaluation on intermediate routers can have important performance advantages.

D. 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.

Fig. 5 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 this experiment, we additionally considered the case in which

PLANet routers are compiled to native code. This allows us further understand the additional overhead due to interpretation. Due to time constraints and implementation difficulties, we are unable provide further native code measurements or evaluation beyond what is presented in this subsection.

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. Though we have no concrete explanation for this drop in bandwidth per-hop, we suspect it is due to compounding loss rates at each router. These loss rates are likely exacerbated by being resident in user space (since we have no direct control over kernel socket buffers), and to jitter from OCaml's thread scheduler. 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 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.

When compiled to native code, PLAN/Ether achieves the full link rate of 93.4 Mbps for one hop (a 66.8% improvement over bytecodes), while switching at about 59.7 Mbps (a 23.6% improvement). When the OCaml bridge is compiled to native code (not shown), it is able to switch at the link rate. These measurements further indicate that the PLANet switching path should be optimized.

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 (bytecode) OCaml bridge is a bottleneck and limits performance to a little more than 80 Mbps.

E. 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 notes regarding future improvements.

E.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. While some portion of this cost is attributable to byte-code interpretation, the native code numbers in Fig. 5 indicate that there is still room for improvement.

E.2 Threads, Copying, and Garbage Collection

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 garbage collection (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.

V. 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 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 Fig. 6. 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.

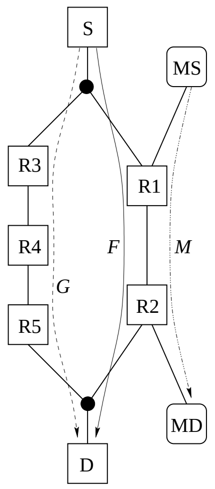


Fig. 6. Experimental Topology

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 to 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.

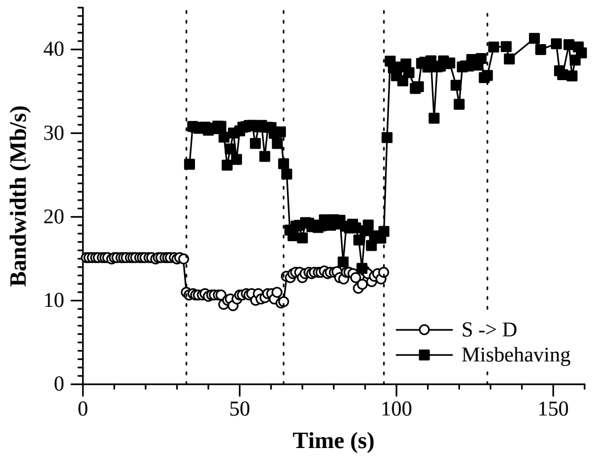
A. Modification of a Queuing Strategy

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 evenly 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[3]. 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 Fig. 7 illustrate this situation.

To more evenly 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 round-robin 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 round-robin 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 round-robin 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.

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



t	Event
0	$S \rightarrow D$ flow begins
33	$MS \rightarrow MD$ flow begins
64	installation of round-robin queue
97	$S \rightarrow D$ flow stops
129	re-installation of FCFS queue

Fig. 7. Dynamic Queue Modification

B. Selection of a Better Route

Let us re-examine our scenario. If we look at the topology shown in Fig. 6, 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.

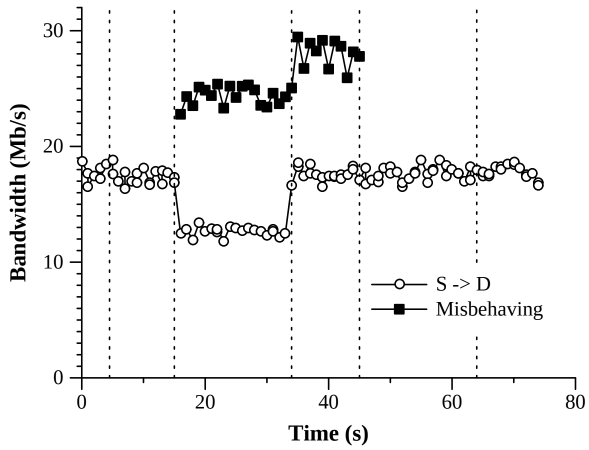
The approach we take in this demonstration is to allow the sender S to “shop around” for the route with the best available bandwidth. The bulk of the data will be carried in the same largely non-active *transport packets* that we used to measure bandwidth in Section IV. However, we will periodically 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 II; 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 [18]. 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 Fig. 8. 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 IV, 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.



t	Event
0	$S \to D$ flow begins, first scout packet sent
5	switch to flow-based routing
15	$MS \to MD$ flow begins
30	second scout packet sent
35	switch to new flow
45	$MS \to MD$ flow stops
60	third scout packet sent
65	revert to original flow

Fig. 8. Flow-Based Adaptive Routing

VI. Related Work and Conclusions

There are several other active networking projects under current development which are useful for contrast. The Active Network Transport System (ANTS) [19] and NetScript [20] rely entirely on dynamically extensible services which are invoked at every hop. The Smart Packets project [21], on the other hand, has programmable packets which evaluate only at the endpoints. Since PLANet allows programmable packets, dynamic extensions, and selective evaluation, it makes accessible a larger part of the design space than any one of these other systems by itself. The reader is referred to [22] and the Active Network Program home page [23] for a listing of other active network projects.

In addition, PLANet is the first purely active internetwork; there is no dependence upon the existing 'inactive' Internet for network-layer services. Other active networking projects such as ANTS and Smart Packets encapsulate their packets within TCP or UDP. Whereas such implementations are useful for simulation and application development, PLANet provides real insight into the issues that would arise in the creation of an active networking infrastructure.

Some 'inactive' networking projects share implementation ideas with PLANet. The Fox project [24], [25] implements the TCP/IP protocol suite for Ethernet and ATM link layers using the ML programming language, in the approach of the x-kernel [26]. OCaml has been used for other serious networking projects such as mobile programs for the MMM web browser [15] and the Ensemble Project's [27] generalization of TCP/IP to group communication.

Some work has been done concerning security in Active Networks. Most notable is the SANE [8] 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 [9].

PLANet is the best-performing active networking system in the literature to date. Wetherall *et al.* [19] 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.*, measure rates of 3.8 Mbps for M0 [28], and about half that rate for ANTS. Finally, Hartmann *et al.* [29] 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.

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] J. Postel, "Internet protocol," Tech. Rep., IETF RFC 791, September 1981.

[2] 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*. 1998, pp. 86–93, ACM, Available at `www.cis.upenn.edu/ ~switchware/ papers/ plan.ps`.

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

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

[5] "Haskell: A purely functional language," `www.haskell.org`.

[6] "Scheme home page," `www-swiss.ai.mit.edu/ scheme-home.html`.

[7] Robin Milner, Mads Tofte, and Robert Harper, *The Definition of Standard ML*, The MIT Press, 1990.

[8] 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*, vol. 12, no. 3, pp. 37–45, 1998, Special issue on Active and Controllable Networks.

[9] Michael Hicks, "PLAN system security," Technical Report MS-CIS-98-25, Department of Computer and Information Science, University of Pennsylvania, April 1998, Available at `www.cis.upenn.edu/ ~switchware/ papers/ plan_security.ps`.

[10] Michael Hicks, Pankaj Kakkar, Jonathan T. Moore, Carl A. Gunter, and Scott Nettles, "Network programming with PLAN," in *Proceedings of the IEEE Workshop on Internet Programming Languages*, May 1998, Available at `www.cis.upenn.edu/ ~switchware/ papers/ progplan.ps`.

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

[12] David C. Plummer, "An Ethernet address resolution protocol," Tech. Rep., IETF RFC 826, 1982.

[13] C. Hedrick, "Routing information protocol," Tech. Rep., RFC 1058, June 1988.

[14] J. Postel, "Internet control message protocol," Tech. Rep., IETF RFC 792, September 1981.

[15] Francois Louaix, "A web navigator with applets in Caml," in *Fifth WWW Conference*, 1996.

[16] Xavier Leroy and François Rouaix, "Security properties of typed applets," in *Proc. 25th symp. Principles of Programming Languages*. 1998, pp. 391–403, ACM Press.

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

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

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

[20] Y. Yemini and S. da Silva, "Towards programmable networks," in *IFIP/IEEE International Workshop on Distributed Systems: Operations and Management*, L'Aquila, Italy, 1996.

[21] "Smart packets home page," `www.net-tech.bbn.com/ smtpkts/ smtpkts-index.html`.

[22] 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*, vol. 35, no. 1, pp. 80–86, January 1997.

[23] "Active network program home page," `www.ito.darpa.mil/ research/ anets`.

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

[25] 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*, pp. 55–64. ACM, 1994.

[26] Sean W. O'Malley and Larry L. Peterson, "A dynamic network architecture," *ACM Transactions on Computer Systems*, vol. 10, no. 2, May 1992.

[27] Robbert van Renesse, Ken Birman, Mark Hayden, Alexey Vaysburd, and David Karr, "Building adaptive systems using Ensemble," Tech. Rep. TR97-1638, Cornell University, 1997.

[28] Albert Banchs, Wolfgang Effelsberg, Christian Tschudin, and Volker Turau, "Multicasting multimedia streams with active networks," Tech. Rep. 97-050, International Computer Science Institute, 1997.

[29] 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," Tech. Rep., University of Arizona, 1997.