Active Bridging

D. Scott Alexander, Marianne Shaw, Scott M. Nettles and Jonathan M. Smith*
CIS Department, University of Pennsylvania
{salex,marianne,nettles,jms}@dsl.cis.upenn.edu

Abstract

Active networks accelerate network evolution by permitting the network infrastructure to be programmable, on a per-user, per-packet, or other basis. This programmability must be balanced against the safety and security needs inherent in shared resources.

This paper describes the design, implementation, and performance of a new type of network element, an Active Bridge. The active bridge can be reprogrammed "on the fly", with loadable modules called switchlets. To demonstrate the use of the active property, we incrementally extend what is initially a programmable buffered repeater with switchlets into a self-learning bridge, and then a bridge supporting spanning tree algorithms. To demonstrate the agility that active networking gives, we show how it is possible to upgrade a network from an "old" protocol to a "new" protocol on-the-fly. Moreover, we are able to take advantage of information unavailable to the implementors of either protocol to validate the new protocol and fall back to the old protocol if an error is detected. This shows that the Active Bridge can protect itself from some algorithmic failures in loadable modules.

Our approach to safety and security favors static checking and prevention over dynamic checks when possible. We rely on strong type checking in the Caml language for the loadable module infrastructure, and achieve respectable performance. The prototype implementation on a Pentium-based HP Netserver LS running Linux with 100 Mbps Ethernet LANS achieves `ttcp` throughput of 16 Mbps between two PCs running Linux, compared with 76 Mbps unbridged. Measured frame rates are in the neighborhood of 1800 frames per second.

*This research was supported by DARPA under Contracts #N66001-96-C-852 and #DABT63-95-C-0073. Additional support was provided by the AT&T Foundation, the Hewlett-Packard Corporation and the Intel Corporation.

1 Introduction

"Active Networks" [TSS+97] are packet-switched networks in which the network infrastructure is programmable and extensible, and where network behavior can be controlled on a per-packet, per-user, or other basis. For example, a packet might carry executable code[TSS+97] that extends the network infrastructure. The goal in developing such networks is to greatly increase the flexibility and customizability of the network, and to thus accelerate the pace at which network software is deployed and evolves. Active Networks provide an infrastructure for implementing earlier approaches to evolving networks such as "Protocol Boosters" [FMS98].

SwitchWare [SFG+96] is an experimental active networking project with the goal of using active networks to facilitate rapid network evolution. This effort must begin with an architecture for the nodes that comprise the active network. To this end, we are building network components ("switches") that can be programmed remotely over the network. A key question in this effort and indeed generally in active networks is how to allow the network to be programmed remotely without compromising the safety and security requirements that are crucial to the shared network infrastructure. A significant aspect of our approach is the use of high-level type-safe programming languages as a basis for extensibility. These languages allow some basic and important low-level safety guarantees to be made by the programming language, thus providing a solid basis on which to build a safe, secure, and extensible software base.

In this paper, we present the results of our initial implementation experiment, an active network bridge. The bridge is programmed in Caml, a statically and strongly typed language. Caml is also used to extend the basic bridge functionality. To demonstrate the usefulness of active networks, we show that we are able to down-load a crucial new algorithm into a running bridge and to dynamically switch the bridge from one operating regime to another. Normally, such a conversion would require bringing down the network and disrupting users. The ability to avoid such disruption represents a significant advantage for active networking. Furthermore, the prototype achieves acceptable performance.

The next section, Section 2, motivates research into Active Networks. Section 3 points out the major safety and security risks of a programmable network infrastructure, and introduces our solution. Section 4 provides background on bridges for Section 5, which describes our Active Bridge, with its incrementally loaded functionality. Section 6 contains a performance analysis, Section 7 relates our work to that of others, and Section 8 summarizes the new results.

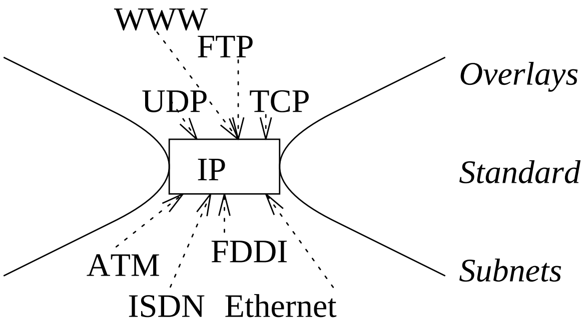

Figure 1: **Hourglass model of internetworking.**

2 Why Active Networks?

Network hardware and applications are evolving rapidly. Unfortunately, key parts of the network infrastructure evolve much more slowly, often taking more than half a decade to make their way from presentation at SIGCOMM to deployment by Internet Service Providers. For example, consider the five or more year time-line from RSVP conceptualization [CSZ92] to deployment [Pap96].

The existing network software infrastructure evolves slowly because of an important and fundamental design goal, the need for interoperability. Interoperability is achieved in the current Internet by using the hourglass model of networking shown in Figure 1. Essentially the idea is that a wide variety of high-level services and low-level network technologies can be made to interoperate if all of their functionality funnels through a common interface, the waist of the hourglass. In the Internet, this waist is the IP protocol, which defines a standard packet format, and a virtual source/destination addressing mechanism that allow a wide variety of systems to interoperate. The success of this idea is clear from its current penetration and acceptance in the marketplace, and its enabling of other schemes such as the world-wide web.

It is the need to standardize on the interoperability layer that makes network evolution slow. This is because when new functionality is needed that cannot be added either under or on top of the interoperability layer, then the interoperability layer itself must be changed. The implication is that some basic changes in the network must be made at the speed with which standardization proceeds, rather than tracking the much more rapid pace of the basic technology. A good illustration of this is the speed (or lack thereof) of adding various types of support for QoS to the Internet.

Active networks address this problem by making a fundamental change in the nature of the interoperability layer. An interoperability layer is still crucial; without it there would be no common ground upon which to communicate. However in an active network, instead of standardizing on the low-level packet formats and exchange protocols, the standard is a programmable interface that allows the low level details to be programmed and customized as needed. As long as two communicating entities can run compatible code they can interoperate. This change allows the network to evolve as rapidly as new software can be developed because now new protocols can be deployed without any mediation of standards bodies.

There are further advantages to active networks. For example, diagnostic functions can be inserted "as-needed," and proprietary protocols can coexist with public standards.

3 The Challenge of Active Networks

Making the network itself programmable offers great power and flexibility, but it also creates significant new safety and security risks. The challenge is this: *How can we provide this flexibility while preserving enough security for the network to be used by all?* A further question is whether the needed safety and security can be achieved with acceptable performance?

SwitchWare is exploring this challenge by focusing on the design and implementation of programmable network elements. With this approach, network elements that were previously "store and forward" become "store, compute, and forward" elements. Programming can be accomplished out-of-band, through an administrative interface, or in-band, through packets that are "capsules," as proposed by Wetherall et al. [TSS$^+$97]. Each packet can contain both data and code that operates on the data. We call such packets "switchlets". This model allows advanced applications such as DNS proxies, self-directed multicast, etc. to be programmed by users or network implementors remotely.

Switchlets must provide for safe and secure extensibility. Extensibility is intimately connected to the programming language that is used for the extensions. Thus, it seems natural to explore programming language-centric approaches to providing the basis of secure extensions, and we are doing so. Modern programming languages such as Java, Modula-3, and ML provide significant advantages in safety and hence security. These advantages derive from the use of strong typing, supplemented by automatic storage management (garbage collection), and array bounds checking. We discuss why strong typing provides safety advantages in some detail in Section 5.1.2.

Because ML [MTH90] is strongly typed, is well studied by programming language semanticists, and because we have some considerable local ML expertise, we have chosen to use Caml [Ler95], an ML dialect, for our work. Caml also has two additional advantages for our work: byte codes and dynamic linking. The byte code format provided by Caml provides us with a machine independent format that is compact for transmitting switchlets over the network. Dynamic linking provides the ability to add these switchlets to an executing program. Several recent networking efforts have also used dialects of ML. For example, Biagioni [Bia94] describes a TCP implementation based on Standard ML of New Jersey [AM87] and van Renesse [vR96] discusses a network stack implementation also based on Caml.

4 Network Bridges

Network designers must trade off complexities of addressing and routing. Bridges[1] are data link layer network elements that interconnect LANs to make extended LANs (ELANs) [Per92]. Bridging is a less flexible interconnection solution than IP internetwork routing, but it offers cost/performance advantages in many settings and is widely used.

When bridges interconnect broadcast LANs such as Ethernet, they must provide the illusion that the ELAN is a broadcast network. For LANs L_1 and L_2, a "dumb" bridge would broadcast all frames seen on L_1 to L_2 and vice-versa. A self-learning bridge [HKS84] optimizes this behavior by tracking packet source addresses; if the destination of a frame lies on L_1 then frames destined for that host need

[1] Throughout this paper, all bridges are transparent bridges, which are invisible to hosts, unlike the less common source routing bridges.

not be forwarded to other LANs $L_2...L_k$ connected to the bridge.

An important limitation of bridges is that since they cannot modify the packet (and therefore cannot use mechanisms like a time to live field), it is possible that they will direct packets in a loop, causing the packet to fail to make progress and wasting network resources. Worse yet, since a bridge that receives one packet may generate several packets, a loop can cause unbounded growth in the number of packets on the network leading to network collapse.

There are a variety of approaches to guaranteeing loop free bridging. A very simple technique is to physically construct the network so that loops cannot form and perhaps to provide detection to indicate when a loop has been created. The solution adopted by the bridge industry is essentially this one, but at the logical rather than physical level. The idea is simply to constrain the bridges so that they never forward packets in a loop. This is done by imposing a spanning tree on the graph representing links between bridges. A spanning tree has only one path between any two nodes, and thus as long as a bridge only forwards packets over ports that are part of a spanning tree, no loops are possible. Ports that are not in the tree do not have packets forwarded on them.

Bridges have two phases of operation, a configuration phase in which a distributed algorithm is run to establish the spanning tree, and an operational phase in which the bridges actively forward packets. For the first several seconds that the bridge is running, it is in the configuration phase. During this time, the local portion of the spanning tree is calculated and the bridge starts to learn the location of hosts by inspecting the source addresses of the packets that are received. During the forwarding phase, the bridge begins to forward packets using the learned locations when applicable and only if that path is part of the spanning tree. The learning and spanning tree algorithms continue to run, in case new hosts or bridges are added to the network.

5 The Active Bridge

Because they are simple, but not trivial network elements, we have chosen to implement a bridge for our initial active network experiment. In particular, ignoring whatever administrative features are added by the manufacturer, a bridge has a simple security model: it forwards frames. Enhancements such as self-learning do not fundamentally change this basic function, and typically the queue service discipline for input and output frame queues is FIFO. Since bridge functionality is simple it is easily tested, and this simplicity is an asset during data analysis. Moreover, the ease of dividing a bridge into the three component functions previously discussed [Per92] make it ideal for a layered architecture such as our implementation.

5.1 The Switchlet Loader

A central aspect of an active network is the ability to load executable code into the network elements. Thus, it is no surprise that a basic component of our system is our switchlet loader, which allows the user to load in new switchlets and to execute them. Another important aspect of the loader is that it establishes the environment in which switchlets execute. Controlling the execution environment allows us to exclude operations like opening disk files that are unnecessary for a switchlet and which may pose security risks.

To produce the restricted environment we use a feature of Caml called *module thinning* [Rou96]. In Caml, groups of related functions, along with their supporting data structures and types, can be aggregated into a *module*. Each module has a *signature* that describes which values, types, and functions are known to external functions. We have *thinned* the signature of the modules to be accessed by switchlets to exclude those functions that might allow security violations. This leaves the switchlet with no way of naming the excluded function and thus, no way of accessing it. We discuss this point in detail in the next section.

Caml supports this process by including signatures in the byte code that is used to transmit switchlets. When Caml compiles a set of sources into byte codes, it includes an MD5 digest of the interfaces required by this module as well as the MD5 digest of the interface exported by this module. If the byte codes are unaltered module thinning works as described. For simplicity, in our current experiment, we have not addressed the authentication issues, but these are an important avenue for further work.

5.1.1 Module Isolation

A key transition made by our security model is from using address spaces to name spaces as the basis of security. Aspects of such a model have been examined in the context of the *Nemesys* operating system for multimedia applications [LMB$^+$96], and in some sense by capability-based systems.

If switchlets were written in a weakly typed language like C, a switchlet would be able to access any part of its address space. Unchecked this ability would allow a switchlet to call any function in the address space, or worse to over-write the code or data of another switchlet. Enforcing security in a shared address space would be impossible. Traditionally, if multiple programs written in weakly typed languages are to share resources, security mechanisms such as separate address spaces are used to ensure that a program can only affect its own physical memory. Page protections ensure that a program only directly reads, writes, or executes unshared resources. When sharing is needed, it is mediated by the operating system, and when different processes need to execute, they require a context switch. These mechanisms operate while the program is executing and thus have a *runtime* cost.

Our prototype presumes a single-language environment and uses strong typing to guarantee that only certain operations are allowed. Furthermore, by using static typing, which enforces type safety during compilation and linking, we are able to substitute compile-time and link-time costs for runtime costs. For example, functions are immutable objects, so there is no operator that allows one to change a function. (One could change a *function reference* to refer to another function, but there is no way to modify the function itself.) Further, there is no equivalent of a C cast operator[2], so there is no way to "trick" Caml into thinking a function is an object that can be changed. Thus, there is no need for page protection since Caml provides equivalent protection on an object by object basis at compile and link times when it ensures that the types are all correct.

For type safety to be fully enforced statically, it is also necessary for the system to use automatic dynamic storage allocation (garbage collection) and to check that array

[2]There are transformation operators for making some safe, well understood transformations. For example, there is a function to transform an integer to a floating point number.

```
val pub_hash:
 (string, (int -> int)) Safestd.Hashtbl.t
val pub_func: unit -> unit
```

Figure 2: **example.mli**

```
open Safestd
let pub_hash = Hashtbl.create 15
let priv_func x = x - 7
let some_func x = (priv_func x) + 5
let pub_func () =
 Hashtbl.add pub_hash "func" some_func
```

Figure 3: **example.ml**

bounds are not overrun. Both of these features improve the safety and thus security of the system in general. For example, many common Unix security holes are created because C does not check array bounds. This allows a malicious programmer to overwrite the stack and upon returning from a call to execute arbitrary code. Similar tricks can be played if the programmer is allowed to free memory that is in use by other parts of the program. Using garbage collection (GC) avoids these problems, since only the system frees storage.

The lack of a cast operator or an address operator also makes it impossible to refer to any object without either its name or a string of legal pointer references from a known object. Name-space based security rests on this feature.

Figure 2 (the module interface) and Figure 3 (the module itself) contain some contrived code to illustrate this point. If another module wishes to reference objects from the module `example`, initially, it can only refer to those objects in its interface (signature). Attempts to access other objects result in compile time errors. If the other module were compiled against a signature built by an attacker that included some private objects, a link time error would result because the signatures would not match.

The values accessible from the interface are the function `example.pub_func` and the hash table called `example.pub_hash`; they can be referred to directly by name. Initially, `example.pub_hash` is empty and does not lead to any functions. When `example.pub_func` is evaluated, then the function `example.some_func` becomes accessible because there is a reference path to it through `pub_hash`, and evaluating `Hashtbl.find example.pub_hash "func"` would return `example.some_func`.

5.1.2 Switchlet Linking Model

For the bridge, we used the rather simple linking model offered by Caml. A `Dynlink` module is provided that supports dynamically linking byte codes containing a module into a running program. To use these facilities, one must first call `Dynlink.init` which creates an empty name space into which modules can be loaded. Next, to create an initial environment, one calls `Dynlink.add_available_units` which enters the names from specified modules that were linked into the loader when it was built into the name space. For the loader, the modules specified in this way provide functions ranging from integer addition to some networking functions. (See the next section for more details.) Finally, by calling `Dynlink.load`, one can load byte-codes and add them to the name space. There is no function to allow previously linked functions (whether linked dynamically or statically) to access the newly loaded functions, so the byte codes usually contain some top-level forms that call a registration function, that changes a data structure visible to previously linked functions.

For the prototype bridge, a single name space is sufficient, and it ensures that many types of accidental references do not occur. However, it does not protect against malicious references. When we start to build devices that we intend to be used by multiple users, we will need a way of managing multiple namespaces that incorporates a variety of security issues. Providing such facilities is another area for research that is important for active networks.

5.2 The Switchlet Loading Process

When the loader first starts, it is limited to those capabilities required to continue the loading process or which must be staticly linked for security reasons. By dynamically loading everything else, we can retain the maximum flexibility. In particular, the initial loader can only load switchlets from disk.

To overcome this limitation, we load a network loader. It consists of four layers. The lowest layer captures those Ethernet layer frames destined for an Ethernet card installed on this machines. It then demultiplexes these frames based on the Ethernet protocol identifier. The next layer implements a minimal IP [Pos81] sufficient for our purposes. (It does not, for example, implement fragmentation.) The IP protocol identifier field is used to demultiplex these packets to other switchlets. The next layer implements a minimal UDP [Pos80] in a similar fashion. Finally, the highest layer in this stack implements a TFTP [Sol92] server. This server only services write requests in binary format. Any such file is taken to be a Caml byte code file and, upon successful receipt, an attempt is made to dynamically load and evaluate the file.

This approach does not describe how we build a network capable of loading all of the switches. There must be a mechanism by which a host can load a protocol into a switch to which it does not have a direct connection. For our bridge, we can easily build up an infrastructure in steps by sending the bridge switchlet to all adjacent switches and then waiting for these switches to start bridging. At the diameter of the extended LAN grows by one at each subsequent step, we can load those switches whose shortest path is one link greater than was possible in the previous step.

If one desires a more concurrent protocol installation, there are several possible choices best chosen based on the protocol and local knowledge. For example, if the switches are currently not forwarding any traffic for lack of any protocol to do so, using a flood algorithm to send the protocol switchlet to all the switches in the network might work well. If there are forwarding protocols on the network, it may be possible to leverage off of these to route our switchlets to the desired locations.

5.2.1 The Interface Provided to Switchlets

Currently, the loader provides an initial set of eight modules. These modules define the basic environment in which a switchlet will execute. We expect this set to continue to evolve as we gain more experience programming switchlets.

The most basic of the modules provided is `Safestd`. This is a slightly modified version of the `Safestd` module from the

```
type packet = { len : int;
                addr : Safeunix.sockaddr;
                pkt : string }
type iport
type oport

exception Already_bound
exception No_interface

(* Input ports *)
val pkts_waiting_p_in: iport -> bool
val bind_in: string -> iport
val bind_addr: string -> iport
val get_next_pkt_in: iport -> packet
val unbind_in: iport -> unit
val unbind_addr: iport -> unit
val get_iport: unit -> iport

(* Output ports *)
val bind_out: string -> oport
val send_pkt_out:
     oport -> string -> int -> int ->
Safeunix.sockaddr -> int
val unbind_out: oport -> unit
val get_oport: unit -> oport
val ready_to_send_p_out: oport -> bool

(* Generic functions *)
val iport_to_oport: iport -> oport

(* Debugging aids *)
val debug_iport_to_string: iport -> string
val debug_oport_to_string: oport -> string
val debug_demux_num_devs: unit -> int
```

Figure 4: **unixnet.mli**

MMM browser [Rou96]. It provides a set of standard Caml functions ranging from integer operations to an implementation of hash tables. As the name implies, it has been thinned to only allow "safe" operations. Similarly, `Safeunix` is a very heavily thinned version of the `Unix` module from Caml. Our version of `Safeunix` provides access to some time related functions and to some types that are needed for networking. Since we provide no functions for generating output as part of `Safeunix`, we provide a module called `Log` that allows logging messages to be generated. It also allows us to change the method of logging, to a terminal, to disk, or not at all.

We also provide a set of thread related modules. These are built on top of the basic Caml threads package that works entirely in user mode. Thus, no speedup occurs due to our multiprocessor. We hope to be able to take advantage of the POSIX threads in the near future. The threads modules are `Safethread`, `Condition`, and `Mutex`. `Safethread` is only very lightly modified from `Thread`; because there are no ways to create an object of type `Thread.t` except by calling `Thread.create`, we can even leave `Thread.kill` in the module.

So far, we have had to create two new modules to provide support. The first of these, `Func`, contains glue routines to allow the loaded functions to properly register themselves. The register routine simply takes a string as a key and a function and enters them into a hash table. There is also a function that allows one to evaluate one of these functions.

Finally, we provided a module to allow access to the network. `Unixnet` provides a set of functions that allow access to the network interfaces on the machine as shown in Figure 4. Our model separates those functions used for input from those from output. In each case, a function is provided to connect to a given port, to connect to the next available port, to disconnect from a port, to check the status of the port, and to send or receive a packet on the port. Because we are building a bridge, whenever an input port is bound, it is put into promiscuous mode. Currently, we use a simple model in which the first switchlet to bind to a given port succeeds and all others fail. We plan to explore the appropriate way to arbitrate between conflicting claims as we continue our work.

5.3 The Bridge Switchlets

The three switchlets that make up the bridge are built on top of the interfaces just described. They are loaded in turn to build up the fully functional bridge.

The first, lowest level switchlet implements a minimal "dumb" bridge. It has three parts. Part one is a function that reads an input packet from a queue and sends it out through a given network interface. Part two is a function that takes an input packet and queues it to all network interfaces except for the one on which it was received. Part three is a function that reads packets from a network interface and demultiplexes them to the functions from part two.

This switchlet is actually performing the function of a buffered repeater. It cannot tolerate a network topology with any loops and will not support a network with aggregate traffic higher than the traffic limit for its slowest segment.

The second switchlet adds learning to the bridge. This switchlet replaces the switching function from the dumb bridge with one that learns the locations of the hosts on the network. For each packet received, the triple (source address, current time, input port) is placed into a hash table keyed by the source address, replacing any previous entry[3]. Next, the hash table is searched for the destination address of the packet. If a match is found and is current, the packet is sent out on the port indicated unless that was the port on which the packet was received. If no match is found, this bridge has not yet learned the destination address and the packet is sent out on all ports except the one on which it arrived.

The third and final switchlet implements the spanning tree functionality. This switchlet adds a function that registers with the demultiplexer requesting packets addressed to the All Bridges multicast address. All other packets continue to be sent to the learning function from the second switchlet. Based on the 802.1D protocol [IEE93], this function takes part in the calculation of the spanning tree for the network. Then it uses access points in the previous switchlets to suppress the traffic from certain input and output ports. With this switchlet, we have a fully functional bridge.

5.4 Automatic Protocol Transition

So far, we have demonstrated that active networks allow one to modularly build up a network element and to extend it

[3] Actually, if the source address is a multicast or broadcast address, this step is bypassed. Similarly, if the destination address is a broadcast or multicast address, the packet is sent out on all ports except the one on which it arrived.

remotely, enhancing its functionality. This is an important step towards our goal of improving network extensibility, but it does not illustrate how active networks allow us to replace basic functionality. To demonstrate this facility, we built a facility that allows our bridge to transition between different, incompatible protocols in a coordinated, automatic manner with automatic fall back if the new protocol should fail.

An important difficulty encountered in managing a network is making changes to the infrastructure. Generally, this requires bringing down the node to be upgraded. If the change is incompatible, the problem becomes much worse. In the case that none of the nodes is capable of acting as a gateway between the two versions of the protocol, the network must either be partitioned into a portion supporting the old protocol and another supporting the new protocol or the entire network must be brought down, upgraded, and returned to service as a unit. Generally, neither of these choices is very acceptable to the users of the network. Moreover, if it turns out that the new protocol does not work for some reason, the same process must be used in reverse to back out the changes.

In this experiment, we show how an active bridge can perform such a transition in an automated, coordinated fashion. The network partition still occurs, but at the pace the infrastructure sends and processes packets rather than at the pace of the network administrators can move from machine to machine. Thus, the transition can be expected to take time similar to what would occur if there were a power failure at each of the bridges. Moreover, active monitoring can occur while the network is stabilizing, detecting any failures in the new implementation and transitioning the infrastructure to a stable state such as the previous protocol.

In order to have a pair of protocols to transition between, we modified the spanning tree switchlet to send DEC spanning tree packets to the DEC management multicast address instead of 802.1D packets to the All Bridges multicast address[4]. This DEC-like protocol was used as the old protocol. The 802.1D protocol was used as the new protocol.

We also wrote a control switchlet that was capable of controlling the transition between the two protocols. This is the component which capitalizes on the ability to actively load switchlets by using locally available information. (If one were to deploy a facility like this generally, we would expect that the old and new protocols would be written by a single programmer and distributed and that the control switchlet would either be written or customized by each of the local network administrators based on their knowledge of their networks.) In order to load the control switchlet, both the 802.1D switchlet and the DEC switchlet must already be loaded. It checks that the DEC switchlet is operating and that the 802.1D switchlet is not. It then arranges to receive any packets addressed to the All Bridges multicast address. When an 802.1D packet arrives, the control switchlet assumes that the network is transitioning to the new protocol. It halts the DEC protocol and starts the 802.1D protocol. It also arranges to let the 802.1D protocol listen to the All Bridges address and it starts to listen to the DEC address. Any DEC protocol packets received during an initial transition period are suppressed. The 802.1D switchlet sends out configuration packets on all of its ports thus causing any bridge that is on a connected network and that has not transitioned to do so.

[4]To completely implement the DEC protocol would require changing some timings and states as well. We did not do this. We simply required an incompatible packet format so that we could make a transition.

action	DEC	IEEE	control
	running		
load IEEE	running	loaded	
load/start control	running	loaded	running
recv IEEE packet	suspended	loaded	suspend DEC; capture DEC state
	loaded	running	start IEEE
30 seconds	loaded	running/ forwarding	suppress DEC packets
60 seconds	loaded	running	perform tests per network admin
pass tests	loaded	running	fallback if DEC packet arrives
fail tests or fallback	running	loaded	stop IEEE; start DEC; terminate

Table 1: **Automatic Protocol Transition**

The control switchlet next moves into its monitoring phase. For this particular transition, the critical change is the protocol for computing the spanning tree. Based on local knowledge, we have determined that the portion of the spanning tree computed at each node should be identical for the old and the new protocols. As such, the control switchlet monitors the information about the spanning tree accumulated at the current node. This is compared with information captured from the DEC algorithm at the time of its termination. (If the changes were such that the information to be checked were not available at the node, it could be hand calculated for the expected case.) If the spanning tree does not converge to the expected values within a predetermined time, the control switchlet will determine that there must be a bug in the new protocol implementation. Similarly, the old host location table could be compared with the new host location tables if the network administrators were not planning to move any hardware during the transition period. If a failure is detected, the functions implementing the new protocol are stopped and the old protocol is restarted. The control switchlet again changes so that it receives new protocol packets (which it suppresses) and allows the old protocol to process its own packets. As a final measure to allow fall back to occur gracefully, if the control switchlet finds any old protocol packets after the initial transition period, it falls back to the old protocol assuming that a failure has occurred elsewhere in the network. Once this fallback has occurred, the network is considered "stable" and no further transition will occur without human intervention.

6 Implementation Details

The active node functions are carried out by programs executing in the Caml bytecode interpreter. The interpreter opens Ethernet sockets (a special type allowed by Linux), to create paths from an input device to the Caml interpreter, and from the Caml interpreter to a corresponding output device.

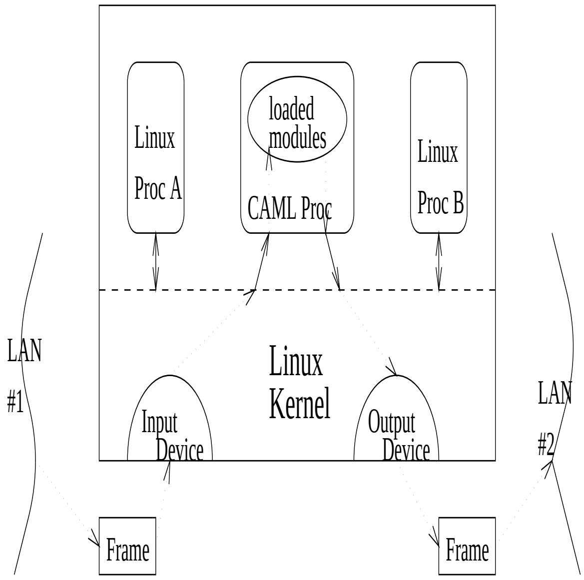

Figure 5: **Path for a packet in a SwitchWare active node**

The path for an Ethernet frame can be decomposed as seven steps, as shown in Figure 5.

1. Frame arrives on Ethernet adapter
2. Ethernet Interrupt Service Routine (ISR) woken; it collects a frame into a buffer chain of Linux network buffers
3. Linux wakes the bridge thread and delivers the frame through `recvfrom()`
4. The Caml program operates on the frame
5. The Caml program emits the packet to Linux network buffers via `sendto()`
6. Linux queues the frame to the Ethernet driver
7. The Ethernet driver emits the frame to the destination LAN

As currently implemented, the packets are all Ethernet frames. The CRC is returned on a read, but cannot be specified on a write. (This is one of our 802.1D incompatibilities.) The packets are represented as a record containing the length of the packet, a Unix `sockaddr` in a Caml form, and a string with the data. The user must unmarshall the data from the string. In the future, we plan to look at the relative advantages of providing unmarshalling functions instead.

7 Performance, Scalability and Flexibility

We study performance under three types of measures.

First are the traditional performance measures such as throughput, latency and packets per second. Latency in a bridge is additional delay incurred by passage through the logic and buffering of the bridge compared to an unbridged system. Bridge throughput will be affected by both per-packet costs and the per-byte costs [CJRS89], or what

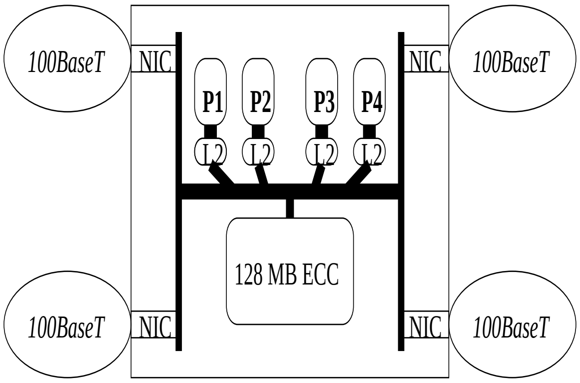

Figure 6: **Active bridge machine architecture**

Pasquale [KP93] has called data-touching costs and non-data-touching costs. Bridge throughput will vary across packet size mixes, for example, very small packets will incur almost all of their overhead as non-data-touching costs.

Second are performance measures specific to the type of networking device under study, for example scalability in the number of ports handled is an important measure of a switching architecture. For a bridge, the capacity to support many LANs and their associated endpoints can be stated as an aggregate throughput, a number of line cards, etc; the important point is to get a sense of where adding another bridge makes more sense than attempting to augment an existing bridge with additional busy hosts, or LAN-attached line cards.

Finally, peculiar to "on-the-fly" programmable network infrastructures is the rate at which changes in the infrastructure can be made and become effective. This is a limiting factor for the *function-agility* of the new network infrastructure.

7.1 Experimental Setup

The hardware platform for the active bridge is a Hewlett-Packard Netserver 5/166 LS4 Model 1, a 4 processor shared-memory multiprocessor. The processors are 166 Mhz Intel Pentiums and each processor has a first level cache that is 8KB of data and 4KB of instruction, write through and a 1 MB second level cache. The machine has 128MB ECC memory, 2 PCI slots, 4 EISA slots, 2 PCI/EISA slots, 2 Fast/Wide SCSI-2 controllers, and 1024K video memory/ SVGA controller. Multiple 100 Mbps Ethernet adapters are used as bridge port controllers. The configuration is shown in Figure 6.

The software architecture is shown in Figure 5. The Linux is Red Hat version 4.0 with a version 2.0.23 kernel.

The active bridge implementation has been operating successfully in our laboratory (replacing a DEC LANbridge) for over 3 months. For performance measurements, we removed it from the laboratory network and interconnected two 100 Mb/s LANs as shown in Figure 7.

The hosts were Intel Pentiums running with a version 2.0.28 Linux kernel. To obtain baseline performance measures, we duplicated latency and throughput measurements on the "best case" configuration of two hosts interconnected by a single LAN, as shown in Figure 8.

Figure 7: **Bridging Setup**

7.2 Latency

We measured latency with the `ping` facility for generating ICMP ECHOs, using various packet sizes to generate frames on the LANs. Results are given in Figure 9. With additional instrumentation, we were able to determine that the Caml code execution adds 0.34ms per frame. We suspect that the additional per frame latency of the bridge is due to Linux and the need to transfer the frame into user space.

7.3 Throughput

Throughput for various packet sizes was measured with repeated `ttcp` trials. The baseline performance, measured on the bridgeless network, was 76 Mb/s. With an 8 KB IP packet size (resulting in multiple back-to-back LAN frames), the active bridge achieves a throughput of 16 Mb/s. The performance in frames/second was calculated for the same frame sizes shown and ranged from about 360 frames per second for small frames (ca. 50 bytes) to 1790 frames per second for 1024 byte frames.

Additional instrumentation showed a cost per frame within Caml of 0.47ms on average during a ttcp trial. This translates to a limiting rate of 2100 frames per second or about 32 Mb/s before accounting for operating system and transmission overheads. We have not yet had an opportunity to isolate the source of the Caml overheads. Three possibilities seem likely. First, some of the cost is certainly due to the cost of the bridge functionality, which is not reflected in our very simple C-based repeater. Second, our current Caml system uses a bytecode interpreter, which is certainly likely to have a severe performance penalty compared to native code. Caml does have a native code compiler, although some work will be needed to adapt it so that it can deal with dynamically loaded switchlets. The final possibility is that we are seeing interference from the garbage collector. The Caml collector is based on the concurrent collector described in [DL93], but the released implementation does not use multiple threads to run the collector. If measurements show that the collector is the bottle neck, then this is a likely source of improvement. Other concurrent collector technologies are available as well, many based on Baker's algorithm [Bak78], and others on techniques that are especially well suited for ML [NO93, ON94].

We also built a very simple buffered repeater in C to try to determine the smallest overheads that a user mode

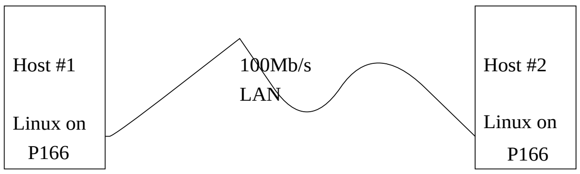

Figure 8: **Baseline Setup**

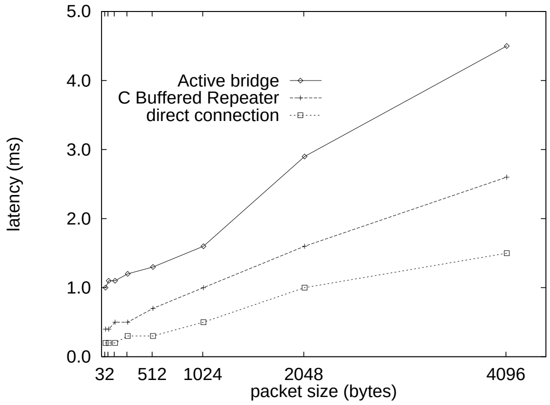

Figure 9: **Ping Latencies**

program could expect to see. This program simply opens two Ethernet devices in promiscuous mode and, for each packet received on one of the interfaces, writes the packet on the other. This gives some idea of the costs caused by bringing the data through the Linux kernel into user space. We intend to examine an approach like that used by the U-Net project [vEBBV95] in the future. They allow protected user mode access to network devices which has reduced the boundary crossing costs.

7.4 Scalability

Using a general-purpose multiprocessor as a switch is an established technique for experimental packet-switching networks [EBE+86, KEM+78]. The major performance limitations associated with such a platform are bus and memory bandwidth limitations inherent in an architecture not specialized for scaling or aggregation. As discussed by Edmond, *et al.*, once the hardware architecture is set, the major performance limitations come from the software architecture. For our system, the major limit is the concurrency we can access in our implementation. First, while it would be advantageous to have threads support for our Caml system to take advantage of the multiprocessor, this is not yet operational. Second, since Caml is a garbage-collected language, there are occasional pauses which force the system to serialize the threads. This is another place where the GC techniques mentioned above may well become important.

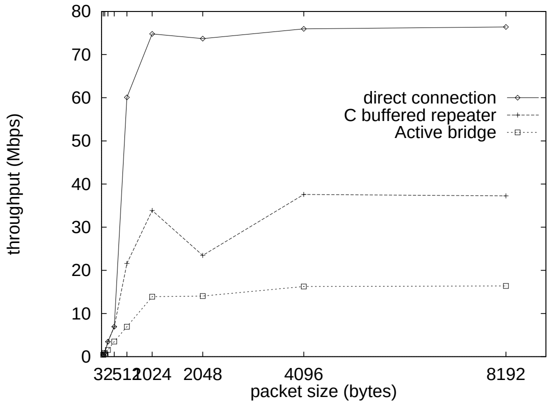

Figure 10: `ttcp` **Throughput**

7.5 Agility

The *function-agility* of a system is the latency for a functional transformation. In the context of active bridging, we can measure this as the time needed to load a module, and the time needed for it to take action. For the spanning tree alternatives we developed, a good measure of the agility is the ability of the active portion of the system (the loaded Caml code) to switch spanning tree protocols.

We performed a final test using a ring shaped network. The HP Netserver acted as an end-node to take measurements. It was configured with two Ethernet cards, `eth0` and `eth1`. Attached between these cards were three of the 166Mhz Pentiums each with two Ethernet cards. These three machines were each running the bridge software with the control switchlet to allow automatic switch-over.

A test program running on the HP sent out an 802.1D spanning tree packet on `eth0` and then waits to see one on `eth1`. (This indicates that each of the bridges in the path between `eth0` and `eth1` have switched to the "new" algorithm.) The program then starts two threads one of which sends out a prebuilt ICMP ECHO on `eth0`, then delays for 1 second, and repeats. The other thread reads packets on `eth1` until it sees one of these pings. `gettimeofday()` is called before sending the IEEE frame after receiving the IEEE frame, and after both threads have died.

For a set of trials, the average start to IEEE time measured was 0.056 seconds, and the average start to received ping time was 30.1 seconds. Thus, the active bridge's reconfiguration was much faster (<0.1 second) than timeouts (accounting for the additional 30 seconds) built into the bridge protocols to ensure that temporary loops do not occur.

8 Related Work

Wetherall and Tennenhouse [WT96] have constructed an Active Network node architecture that uses a TCL interpreter operating on the Linux operating system. The ACTIVE_IP option is used to flag IP packets containing *capsules*, which are packets and data passed to the TCL interpreter. The scheme has been applied to some IP tasks, such as implementing `traceroute` with capsules. This work has been very focused on a proof-of-concept for an active node, and demonstrating viability of the idea. Our research is concentrated on the design and construction of the programming environment for a robust extensible node, and would be as useful for capsule support as it is for adding bridge functionality. Other work at MIT [WEK96] has demonstrated methods for loading network modules (Application-Specific Safe Handlers, or ASHs) into O.S. kernels. Like our work, ASHs rely on pre-module loading static analyses; we believe that the Caml approach offers a better long-term opportunity for formal verification.

Zegura, *et al.* [BCZ96] have designed a similar IP-based system, which demultiplexes an arriving packet with the option flag, IPOPT_AP, to a pre-loaded function under SunOS. This approach has two flaws from our perspective. First, it appears inflexible with respect to applications loading new functions on-the-fly. Second, to address this flexibility limitation with modifications to SunOS, there will be security risks unless languages and tools are used to validate loaded modules, as in our approach.

Liquid Software [HMPP96] extends the capabilities of the Java language bytecodes as mobile code fragments. Major foci are technologies for fast compilation of mobile code (to ensure high packet processing rates) and runtime support using the Scout operating system as a basis. The interaction between the Scout system and the Java API should provide some valuable lessons in the nature of a virtual machine that could support Active Nets. The choice of Java is a potential weakness, as a number of serious security problems have been discovered. Unlike ML [MTH90], Java lacks a mathematical definition, making formal analysis of programs difficult.

BBN's Smart Packets [PJ96] is focused on the efficient construction of the packet interpreting programming system for active networks. Smart Packets, like the capsules proposed in the MIT design, contain code in some form. The initial effort seems targeted at efficient byte code (e.g., a refined Java or intermediate language) or even machine code in the packet. The intent is to ensure that loadable active technologies will be viable on even the highest-performance networks.

Netscript [Yd96] provides a model for network programming, but is less focused on the construction of active network nodes than the MIT, Georgia Tech, Arizona and BBN efforts, and more on defining examples of network programming.

MMM [Rou96] is a browser that uses Caml as its applet language. They also use the strongly typed features of the language for security, but are able to assume a model in which all applets cooperate. Further, their applets extend a browser rather than extending the functionality of a network switch. Some of our infrastructure, in particular support for module thinning is derived from their efforts.

9 Conclusions

We have made significant progress towards a robust and flexible programmable network infrastructure. The active bridge described in this paper is written in a modern programming language, Caml, with strong static type-checking and memory safety through garbage collection. Several different bridging styles were enabled using loadable modules ("switchlets") that we injected into the active node on the fly.

In the experiments we performed, the bridge was able to support about 44% of the throughput seen by a C program that provided repeater, but not bridge functionality. The measured throughput is 16 Mbps, with an intervening operating system and interpreted Caml modules. Optimizations such as compiling switchlets into native code for faster operation, shortening the Linux path between interrupt arrival and switchlet operation, improving GC performance, and increasing concurrency, all offer possibilities for improving this result.

An important result to take away from this paper is the flexibility our Active Networking technology provides, even in the restricted domain of transparent bridging. We started with a simple repeater, and extended it with switchlets to become self-learning, to run spanning tree protocols, and to adapt the choice of protocols using information encoded in the Ethernet frame. Advanced algorithms for scaling bridged LANs [SC88] using a multiplicity of spanning trees or LAN interworking functions [VP88] could be added as switchlets to the current system.

Next, we plan to see what steps can be taken to minimize Linux overheads, increase concurrency, and to extend the ideas described here to an active router. The active router has the advantage, from a switchlet design perspective, that it need not be transparent. This opens up a much larger set of functions that can be used. Among these functions are various forms of application-specified Quality-of-Service, and since Caml is garbage-collected, we are investigating Caml run-time issues such as a real-time garbage-collector for the router. As an example of a problem facing current systems that could be solved with such technology, consider the problem of a bottleneck link in the Internet, where a policy dictates a 25% link fraction for a particular user. The user could load a policy for working within this limit, leading to both better performance for the user and possibly less effort on the part of the policing function. Another problem that could be addressed, in a manner similar to our active protocol transition, is support for multiple versions and parallel network infrastructures in a single network element. Thus, IPv4, IPSEC, IPv6, and experimental IP modifications such as support for mobility could be loaded, configured for fail-soft operation, and operating concurrently.

In summary, network elements should be secure and robust. The active bridge described here performs reasonably, is flexible and extensible, and illustrates an attractive path towards an active network infrastructure.

10 Acknowledgments

The original SwitchWare ideas were developed in collaboration with Dave Farber, Dave Feldmeier, Carl Gunter and Dave Sincoskie. Christian Huitema suggested the application of user-loadable policy modules for an Internet with a bottleneck WAN link. Jonathan Smith would like to thank Bellcore and the University of Cambridge Computer Laboratory for hosting him while some of this work was done. Scott Alexander would like to thank Bellcore for hosting him while some of this work was done.

References

[AM87] A. W. Appel and D. B. MacQueen. A Standard ML Compiler. In *Functional Programming Languages and Computer Architecture*, pages 301–324. Springer-Verlag, 1987. Volume 274 of *Lecture Notes in Computer Science*.

[Bak78] Henry G. Baker. List processing in real-time on a serial computer. *Communications of the ACM*, 21(4):280–94, 1978.

[BCZ96] Samrat Bhattacharjee, Ken Calvert, and Ellen W. Zegura. Implementation of an active network architecture. Technical report, Georgia Institute of Technology, July 1996. White paper presented at Gigabit Switch Technology Workshop, Washington University.

[Bia94] E. Biagioni. A structured TCP in Standard ML. *Proceedings, 1994 SIGCOMM Conference*, pages 36–45, 1994.

[CJRS89] D. Clark, V. Jacobson, J. Romkey, and H. Salwen. An analysis of tcp processing overhead. *IEEE Communications Magazine*, 27(6):23–29, June 1989.

[CSZ92] D. Clark, Scott Shenker, and L. Zhang. Supporting real-time applications in an integrated service packet network: Architecture and mechanism. In *Proceedings, 1992 SIGCOMM Conference*, pages 14–26, August 1992.

[DL93] Damien Doligez and Xavier Leroy. A concurrent generational garbage collector for a multithreaded implementation of ML. In *Conference Record of the Twentieth Annual ACM Symposium on Principles of Programming Languages*, ACM SIGPLAN Notices, pages 113–123. ACM Press, January 1993.

[EBE+86] W. Edmond, S. Blumenthal, A. Echenique, S. Storch, T. Calderwood, and T. Rees. The butterfly(tm) satellite imp for the wideband packet satellite network. In *Proc. 1986 ACM SIGCOMM Conference*, pages 194–203, 1986.

[FMS98] D. C. Feldmeier, A. McAuley, and J. M. Smith. Protocol boosters. *IEEE JSAC Special Issue on Protocol Architectures for the 21st Century*, 1998.

[HKS84] W. Hawe, A. Kirby, and B. Stewart. Transparent interconnection of local area networks with bridges. *Journal of Telecommunication Networks*, September 1984.

[HMPP96] John Hartman, Udi Manber, Larry Peterson, and Todd Proebsting. Liquid software: A new paradigm for networked systems. Technical Report TR 96-11, University of Arizona, June 1996. http://www.cs.arizona.edu/liquid/.

[IEE93] IEEE. Media access control (mac) bridges. Technical Report ISO/IEC 10038, ISO/IEC, 1993.

[KEM+78] D. Katsuki, E. S. Elsam, W. F. Mann, E. S. Roberts, J. G. Robinson, F. S. Skowronski, and E. W. Wolf. Pluribus: An operational fault-tolerant multiprocessor. *Proceedings of the IEEE*, 66(10):1146–1159, October 1978.

[KP93] Jonathan Kay and Joseph Pasquale. The importance of non-data touching processing overheads in tcp/ip. In *Proceedings ACM SIGCOMM Conference*, pages 259–269, September 1993.

[Ler95] Xavier Leroy. *The Caml Special Light System (Release 1.10)*. INRIA, France, November 1995.

[LMB+96] I. M. Leslie, D. McAuley, R. Black, T. Roscoe, P. Barham, D. Evers, R. Fairbairns, and E. Hyden. The design and implementation of an operating system to support distributed multimedia applications. *IEEE Journal on Selected Areas in Communications*, 14(7):1280–1297, September 1996.

[MTH90] R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML*. MIT Press, 1990.

[NO93] S. M. Nettles and J. W. O'Toole. Real-Time Replication Garbage Collection. In *SIGPLAN Symposium on Programming Language Design and Implementation*, pages 217–226. ACM, June 1993.

[ON94] J. O'Toole and S. Nettles. Concurrent Replicating Garbage Collection. In *ACM Symposium on LISP and Functional Programming*. ACM Press, June 1994.

[Pap96] D. Pappalardo. BBN to test RSVP. *Network World*, 13(50):1,14, December 1996.

[Per92] Radia Perlman. *Interconnections: Bridges and Routers*. Addison-Wesley, 1992.

[PJ96] C. Partridge and A. Jackson. Smart packets. Technical report, BBN, 1996. http://www.net-tech.bbn.com-/smtpkts/smtpkts-index.html.

[Pos80] Jon Postel. User datagram protocol. Technical report, University of Southern California, Information Sciences Institute, Marina del Rey, CA, USA, 1980.

[Pos81] Jon Postel. INTERNET protocol. Technical report, University of Southern California, Information Sciences Institute, Marina del Rey, CA, USA, 1981.

[Rou96] François Rouaix. A web navigator with applets in Caml. *Fifth WWW Conference*, May 1996. http://pauillac.inria.fr-/mmm/papers/mmm.ps.gz.

[SC88] W. David Sincoskie and Charles J. Cotton. Extended bridge algorithms for large networks. *IEEE Network*, 2(1):16–24, January 1988.

[SFG+96] J. M. Smith, D. J. Farber, C. A. Gunter, S. M Nettles, D. C. Feldmeier, and W. D. Sincoskie. SwitchWare: Accelerating network evolution. Technical Report MS-CIS-96-38, CIS Dept. University of Pennsylvania, 1996.

[Sol92] Karen R. Sollins. The TFTP protocol (revision 2). Technical report, MIT, 1992.

[TSS+97] D. L. Tennenhouse, J. M. Smith, W. D. Sincoskie, D. J. Wetherall, and G. J. Minden. A survey of active network research. *IEEE Communications Magazine*, pages 80–86, January 1997.

[vEBBV95] T. von Eicken, A. Basu, V. Buch, and W. Vogels. U-net: A user-level network interface for parallel and distributed computing. In *Proceedings of the 15th SOSP*. SIGOPS, 1995.

[VP88] G. Varghese and R. Perlman. Transparent interconnection of incompatible local area networks using bridges. In *Proceedings, 1988 SIGCOMM Conference*, pages 381–389, August 1988.

[vR96] Robbert van Renesse. Masking the overhead of protocol layering. In *Proceedings, 1996 ACM SIGCOMM Conference*, pages 96–104, Palo Alto, CA, 1996. SIGCOMM.

[WEK96] D. A. Wallach, D. Engler, and M. F. Kaashoek. Ashs: Application-specific handlers for high-performance messaging. In *Proc. 1996 ACM SIGCOMM Conference*, 1996.

[WT96] David J. Wetherall and David L. Tennenhouse. The ACTIVE_IP option. In *7th ACM SIGOPS European Workshop*, September 1996. http://www.tns.lcs.mit.edu-/publications/sigops96ws.html.

[Yd96] Y. Yemini and S. daSilva. Towards programmable networks. In *IFIP/IEEE International Workshop on Distributed Systems: Operations and Management*, October 1996. http://www.cs.columbia.edu-/ dasilva/netscript.html.