Operating System Support for Generalized Packet Forwarding

Yitzchak M. Gottlieb

A Dissertation
Presented to the Faculty
of Princeton University
in Candidacy for the Degree
of Doctor of Philosophy

Recommended for Acceptance
By the Department of
Computer Science

November 2004

Abstract

Computer networks provide communications services to applications. The most well-known example of a computer network is the Internet—a network of computer networks that provides a point-to-point, best-effort, packet-delivery service. Recently, there has been an increased interest in expanding the set of services that the Internet provides. However, as a given networking technology becomes entrenched, it becomes exceedingly difficult to modify. Most new services are therefore implemented in applications that create overlay networks—virtual networks overlaid on the Internet.

Many overlay networks ignore the well-established networking principle of distinguishing bewteen control and data, thereby limiting their flexibility and performance. The principle identifies two classes of traffic at a network host: data that passes through the host and control that is received by the host. Control messages may provoke expensive computation, while data should require only relatively simple forwarding. Router designers have leveraged the distinction between control and data to make routers more flexible and faster by offloading control computations to a separate processor and optimizing data forwarding in dedicated hardware. Overlay networks that ignore the distinction between control and data cannot derive similar benefits.

Overlay networks are mostly application-specific. They are tailored to meet the needs of a single service, making it difficult to use the network for another service. This reality conflicts with the lesson learned from the Internet that a single network can easily support many different applications.

This dissertation makes three contributions. First, it shows how network services, especially overlay networks and their applications, can be decomposed into control and data planes, and further decomposed into general and application-specific parts. Second, it proposes an architecture, Plug Board, that provides a suitable framework for building network services that make use of this decomposition. Third, it describes the potential benefits reaped by a network service written for Plug Board.

Acknowledgments

Before presenting the contents of this dissertation, I would like to spend a few paragraphs thanking those who helped make it possible.

I would like to thank my adviser, Larry Peterson, and the other members of my thesis committee: Brian Kernighan, Vivek Pai, David Walker, and Randy Wang. As my adviser for the past few years, Larry has provided me with encouragement, guidance, and advice. I would like to thank my primary readers, Brian and Vivek, for helping me to hone the presentation of my thesis. Special thanks go to Melissa Lawson, the department's graduate coordinator, for magically resolving administrative difficulties big and small.

I would also like to thank my colleagues and fellow students including Andy Bavier, Scott Karlin, Björn Knutsson, Akihiro Nakao, Xiaohu Qie, Nadia Shalaby, Tammo Spalink, Mike Wawrzoniak, Limin Wang, and Wen Xu. Andy and Mike helped initiate me into the world of Scout and SILK and worked with me to maintain it. Scott developed the Linux and StrongARM code for using the IXP1200, while Tammo wrote the initial version of the microcode.

My family deserves special thanks. I would like to thank my parents for their love and support throughout the years. Thanks to my sister, Sigal, who was always there as a sounding board and to my brother-in-law, Lenny, for all his help. Thanks to my brother, Adi, for allowing me to state the obvious. Thanks to my daughter, Vered, without whom this dissertation would have been completed much sooner. Last, but most certainly not least, I want to thank my wife Margalit. Without her support, encouragement, and love none of this would have been possible.

This work has been supported in part by NSF grant ANI-9906704, DARPA contract F30602–00–2–0561, and the Intel Corporation. Portions of this work were originally published as Gottlieb and Peterson [24] and Spalink *et al.* [55].

To My Dearest Love

Contents

Chapter 1

Introduction

Computer networks have become ubiquitous over the last forty years; they are found in businesses, schools, and homes. A computer network is predominantly viewed as a communications medium. However, it would be more accurate to say that a computer network is a medium that provides communications services to applications. The Internet, for example, is a network of computer networks that provides a packet-delivery service.

Recently, there has been an increased interest in expanding the set of services that the Internet provides. However, as a given networking technology becomes entrenched, it becomes exceedingly difficult to modify. Most new services are therefore implemented in applications that create a virtual network overlaid on the Internet. This dissertation explores how the inherent structure of network services can be leveraged to make services more flexible, easier to develop, and more efficient in their use of network resources.

This chapter introduces the context of this disseratation's thesis. Section 1.1 describes the Internet's current services and some of the new services recently propsed. Section 1.2 then discusses methods for deploying new functionality in the Internet. Next, Section 1.3 explains the classic distinction between control and data and its importance in maintaining high performance in the Internet. Finally, Section 1.4 presents the thesis that the distinction between control and data can be generalized to apply to these new network services and their deployment methods.

1.1 Network Services

The basic service that the Internet provides to applications is point-to-point, best-effort, packet delivery. The Internet is a network of networks, with each network possibly further divided into subnetworks, or *subnets*. To contact a remote host on the Internet, an application executing on the local host specifies the remote host's address as the destination for the data it is transmitting. The local host's operating system breaks the data into individual packets and places a header on each packet. The Internet attempts to deliver each packet to the remote host. If the two hosts are on the same subnet, the packet is transmitted over the shared network. However, the remote host may be on a different network than the local host. To contact a remote host on a different network, the local host transmits the packet to a host on the local subnet that acts as a gateway, or *router*, connecting networks together. The packet is forwarded through a sequence of routers—the *forwarding path*—until it reaches the subnet of the remote host, where it is delivered directly to that host. It is these routers that implement the Internet's provided service.

When a packet arrives at a router, the router examines the destination address in the packet's IP header and chooses a single destination for the packet. This destination can be either the desired remote host or the next router in the sequence, the *next hop*. The router then modifies two fields in the packet's header. First, it decrements the time-to-live (TTL) field, a counter that limits the number of hops through which the packet should be forwarded. Second, it recomputes the checksum that is used to detect corruption of the packet's contents. It then sends the packet to the selected destination. Throughout the process, the router makes an effort to deliver the packet, but it makes no guarantees. If the router cannot choose a next hop, it will drop the packet. If the router decrements the value of the TTL field to 0, it will drop the packet. If the router runs out of space to store packets, it will drop the packet.

Several novel network services have been proposed in the past few years. The new services can be categorized into four types by the operations that each performs on a packet. First, there are services that offer new forwarding algorithms, such as broadcast or multicast. Second, there are services that monitor the state of the network and filter the traffic instead of blindly forwarding and forgetting it. Third,

there are services that mark packets with application-specific tags, modifying more of the packet's header than just the TTL and checksum. Last, there are some services that manipulate a packet's payload, delivering a changed packet to its destination. Some of these services are completely new while others are merely slight modifications to the existing service.

1.1.1 Forwarding Algorithms

One method of extending the service that the Internet provides is to use a new forwarding algorithm. The forwarding algorithm that a router uses is fairly simple. It chooses a single destination for a packet based on the subnet to which the packet's destination address belongs. This subsection gives some examples of new forwarding behaviors that have been proposed to supplement the Internet's service. The first two examples describe how a router can deliver a packet to more than one host. The next two change how the router chooses a next hop for the packet. The last example shows how applications can define their own, completely new, forwarding algorithms.

Sometimes a point-to-point service is not enough. Some applications require contacting many hosts throughout the Internet or all the hosts on the same network. If each of these destinations were to be contacted individually with the same data, the network would carry multiple copies of the packet on the same link, wasting network capacity. The two subcases of this problem, broadcast and multicast, have been addressed in slightly different ways.

The advantage of being able to broadcast an IP packet to an entire subnet was recognized early in the design of the Internet. The broadcast scheme was designed to work well within the Internet's basic, point-to-point service and was deployed concurrently with it. The Internet's standard forwarding algorithm can broadcast a packet to a particular subnet by forwarding the packet in the usual way until that subnet's router. If that subnet's network supports a broadcasting mechanism, the router uses that mechanism to deliver the packet to all the hosts on the subnet. Otherwise, that router and all the routers in that subnet must transmit a single copy of the data along each of the links that would have been used in transmitting

the packet to each host on the subnet in the point-to-point scheme. In this way, each host receives a copy of the data, and network use is minimized.

The second subcase, multicast, is more complicated. As defined in the Internet Standard [16], "IP multicasting is the transmission of an IP datagram to a 'host group', a set of zero or more hosts identified by a single IP destination address." These hosts need not all be on the same network, so multicast routers cannot use the same forwarding algorithm as the broadcast scheme. Instead, routers forward a packet with a multicast address only through those links that lead to hosts who have subscribed to the group. To determine where the appropriate hosts are, routers and hosts use the Internet Group Messaging Protocol (IGMP) to join a multicast group. This process of joining a group lets the router record which interfaces should be used for the multicast address. Once the router knows the appropriate interfaces to use, it can forward multicast packets through all the appropriate links.

The previous examples extend IP's forwarding algorithm by allowing a single address to refer to many destinations. Another, much newer and mostly undeployed, variant of the Internet's point-to-point service allows a single address to refer not to a specific host, but to any one of a set of hosts. An anycast [39] address is an IP address that is shared by a group of hosts. When a client sends a packet to an anycast address, routers are free to forward the packet to any member of the group using the standard IP forwarding algorithm. This freedom allows sites to distribute services more easily and to mirror data transparently.

In another example of a service that decouples an IP address from a specific physical subnet, mobile IP [40] defines how an Internet host can physically move from network to network yet still maintain the illusion that it is always connected to the same IP address. One scheme for supporting mobile IP requires a "home agent" to receive all traffic bound for the mobile host and to forward it to the mobile host's current physical location. In essence, the home agent is a router that forwards IP packets using an algorithm different from the standard IP forwarding algorithm.

Instead of modifying the IP forwarding algorithm at the routers, applications can also define new forwarding algorithms that operate at the hosts. Proxies are application-specific gateways that seem to provide a service, but actually only mediate the service from a client to a server. Using a proxy enables certain benefits for the user and administrator. For example, Web proxies can cache responses to page

requests and serve subsequent requests locally, thereby reducing network use and response times. To use a proxy, the client connects to the proxy and requests access to a service. The proxy then contacts the server that provides the service on behalf of the application. Stated somewhat differently, the proxy forwards the request for service to the proper server using an application-specific forwarding algorithm.

Peer-to-peer applications are also examples of application-level services that define new forwarding functions. Peer-to-peer applications typically create an application-level network in which each peer forwards other users' requests based on an application-specific protocol. The protocol can be as simple as sending a copy of the message to all known peers or as complicated as computing a prefix matching algorithm to find the next hop. Peer-to-peer applications and their networks are discussed in more detail in Chapter 4.

1.1.2 Network Monitoring and Filtering

Routers are in a unique position to provide information about the instantaneous status of the network from an internal perspective. While an individual router cannot know the state of the entire network, or even of the network in the immediate neighborhood, it alone has full knowledge of the traffic that flows through it. A router can examine this traffic and, to some extent, control it to enhance the security of the network and maintain its availability.

A router's position in the network makes it the perfect platform from which to gather data on the nature of the network traffic. The Internet Architecture Board recommends that all routers keep track of certain statistics about the traffic they forward [34]. This Management Information Base (MIB) contains basic traffic data like the number of forwarded packets and the number of errors encountered. More detailed traffic statistics, like the relative frequency of data from a particular application or from a particular set of hosts, can allow network administrators to engineer the network for its traffic.

Routers can also use their view of the network to react to abnormal network conditions. For example, intrusion detection systems (IDS) for networks can discover whether a network has been compromised by an attacker [37]. By studying the pattern of packets flowing through a specific point in the network, an IDS can

identify hosts that are behaving suspiciously and can either notify a system security officer or isolate traffic coming from the affected host.

A router's ability to forward, or selectively not forward, packets makes it an ideal candidate for acting as a firewall, permitting only certain traffic into a network. By limiting the number and type of packets that can access the hosts on the network, firewalls can mitigate the risk of unauthorized access to the protected hosts. They therefore provide a safer packet delivery service.

Routers can also drop packets to maintain the network's availability to more users. One method for avoiding congestion in a network is to randomly drop packets even when the router is not congested. This method, called Random Early Detection (RED) [22], and its variants [6, 7, 13] force the hosts transmitting data through the router to behave as if the router is congested, thereby reducing the amount of actual congestion at the router.

In the face of actual network congestion that requires packets to be dropped, a router can either drop random packets from a stream or it can preferentially drop less important ones. Application-specific filters can identify how important a particular packet is to the data stream and instruct the router to preferentially drop less important packets, allowing the application's performance to degrade gracefully. One example of this technique is encoding video streams using a wavelet encoding scheme [20]. This scheme encodes a frame in many layers, each consisting of data at a particular "frequency." Higher frequency is correlated with finer details in an image. A congested router could drop packets containing higher-frequency, detailed features of the image instead of those containing lower-frequency, base features. This approach allows a large number of packets from the video stream to be dropped while maintaining a usable video stream.

Routers can also control the order in which they forward packets. Instead of forwarding all packets in the order in which they arrived, routers can give preference to traffic coming to or from particular hosts, enhancing the nominally equal, best-effort delivery that characterizes Internet traffic. Specifically, flows that have been registered for specific rates using the Resource Reservation Protocol (RSVP) [66] could be guaranteed a particular bit rate or loss rate at the expense of other traffic.

1.1.3 Packet Tagging

While many methods of monitoring network status require reporting the results to a managing entity in a special, dedicated stream, routers can also communicate using an in-band mechanism—*packet tagging*. Packet tagging refers to adding a small message, or *tag*, to a packet's header as it is forwarded through the router. The tag can be as small as a single bit, but may be larger. Other routers and the receiving hosts can use this message to alter their sending behavior or to determine the state of the network. As the next two examples show, tagging packets as they enter an administrative domain allows network administrators to add data that has meaning only within that domain.

As noted above, one of the limitations of the Internet's traditional best-effort service is that all packets are treated equally. The Differentiated Services (DiffServ) [38] architecture provides a method for providing different qualities of service to packets. As defined by the IETF Differentiated Services Working Group, DiffServ provides up to 64 different service levels or "per-hop behaviors." For each packet, the value of the type-of-service (TOS) field in the packet's header specifies the per-hop behavior that a router must give it. Given the importance of the value of this field, network administrators must guarantee that the value is appropriate to the packet. Therefore, routers at the boundaries of administrative domains decide the appropriate service level for the packet and tag the packet with the corresponding type of service.

Multiprotocol Label Switching (MPLS) [14] provides a standard way to add a label to an IP packet. Once a Label Switching Router tags a packet, it can be forwarded by other routers in the same administrative domain using a simple lookup on the fixed size label instead of IP's longest prefix matching algorithm. This method is an arguably faster forwarding algorithm that eases administration when that packet's destination is not in the local administrative domain.

In the previous examples, the tags only had reliable meaning within a particular administrative domain. The Internet's Transport Control Protocol (TCP) [2] requires that a host reduce the rate at which it sends when it detects that the network is congested. The traditional method of discovering congestion is to assume that a packet will be acknowledged within a particular amount of time and to conclude

that the network is congested if it is not. Explicit Congestion Notification (ECN) allows routers to tag packets explicitly to warn the sender and receiver that the network is congested even if packets have not yet been lost. As a result, the sender can reduce its sending rate without having to retransmit data. Since the entire path between sender and receiver is congested even if only part of it is congested, the ECN tag must cross administrative boundaries.

ECN provides a coarse-grained measure for congestion at a particular router—the router is either congested or it is not. To provide more fine-grained congestion-avoidance decisions, the eXplicit Congestion Protocol (XCP) [30] allows routers to tag the packet with a measure of how much congestion it experienced during forwarding. Applications can then reduce their sending rate in proportion to the congestion at the routers, thereby reducing congestion in the network while maintaining a high bit rate.

Packet tags can also reveal the forwarding history of a packet. One problem in defending against denial of service (DoS) attacks against Internet hosts is finding the source of the attacking traffic. While it is relatively simple to find the path a packet would take to reach a destination, it is not easy to trace a packet back to its source in the same way. Savage *et al.* [52] proposed a traceback technique that relies on routers randomly tagging packets with an identifier. Given enough packets, this identifier can be used at the receiver to recreate the packets' path with high probability. Once the packet's source is found it is possible to block the attack at a point before it affects the targeted host.

1.1.4 Payload Manipulation

While tagging packets with single bits or small values suffices for many tasks, some new services alter a packet's contents more dramatically. A router may rewrite any part of a packet, including its header and any or all of its payload. These alterations can be used to disguise a packet's origin or to convert the data to a more efficient form.

The current version of the Internet Protocol, version 4, uses 32-bit addresses. While this means that there are as many as 4 billion addresses for hosts, the way that IP addresses have been allocated has led to a perceived shortage of IP addresses,

especially for the use of small networks. Many home and small office networks therefore use addresses in the range set aside for private use on networks not connected to the rest of the Internet. A technique called Network Address Translation (NAT) [57] provides general Internet connectivity to hosts on a private network.

Network address translation allows users to maintain the illusion that all the hosts on the private network are a single host on the public Internet. To do this the router connecting the private network to the public Internet must alter the address and checksum fields in the IP, UDP, and TCP headers on incoming and outgoing packets. This manipulation is sufficient for applications that do not make assumptions about network addressing. However some applications, like the File Transfer Protocol (FTP) [1], specify TCP port numbers in their communication. The NAT router must support these applications directly in order for them to operate properly.

A similar type of content rewriting is used in content distribution networks [3]. A content distribution network is a set of hosts on the Internet that cache popular data. Companies publishing content on the Internet, typically using the World Wide Web, can improve the observed response time of their servers by redirecting requests to a cache near the user. One technique for redirecting a user to a cache is to rewrite the Uniform Resource Identifiers (URI) recorded in the packet to refer to cached copies of the data. Using this technique at a Web proxy is a powerful method of increasing performance while bounding dedicated resources.

Some applications derive their benefit by rewriting the entire contents of the packet. Consider a video stream that is being transmitted over the Internet to a personal digital assistant (PDA) connected by a wireless link. If the video stream was encoded for playback at a PC, some of the video stream will be wasted transmitting useless data since the PDA's video display capabilities are clearly inferior to those available at a PC. If a router along the path decodes the video stream and then reencodes it for the PDA, it may decrease network use with no adverse effect on the user.

1.2 Deploying Novel Router Functionality

Deploying a new network service can be difficult. The primary problem is that since the programming interfaces for most routers are closed, one must wait until a router vendor implements a new service before it is available. A considerable amount of time can elapse from when a new service is proposed until a vendor is convinced that it should be supported, then develops, tests, and releases it. Services that require universal router support can only be deployed after all, or most, routers have been replaced or upgraded. Further, since new features are disabled by default, a service may not be deployable until the new feature is widely accepted. However, if the service is not available, there is not likely to be much demand for the new feature. This vicious cycle has doomed many proposed features over the years.

The long time-to-market of new services prohibits researchers from experimenting with new services to discover unforeseen weaknesses and to develop solutions. Over the years, researchers, including those in the industrial research community, have developed a series of approaches to this problem. However, as these solutions are themselves subjects of active research, they too have yet to see widespread deployment. Of the three approaches that presented here, only overlay networks are becoming accepted as a solution to the problem.

1.2.1 Programmable Networks

The first approach that enables administrators to inject new code into routers and switches is standardizing a programming interface to the routers, thereby creating a network that is inherently programmable. Routers in a programmable network provide a programming interface for operations such as adding and deleting routes, changing queue sizes, and reading statistic-gathering counters. A standard interface allows users to implement a new feature for the routers that would work on all compliant hardware without waiting for router vendors to implement that feature.

The Nortel Networks Openet [33] project is one example of a standard, programmable layer available for deployment on Nortel's Accelar routers. Their approach takes into consideration that routers have two types of processing hardware: general purpose processors and application-specific integrated circuits (ASIC). The ASIC permits forwarding at high bit rates but is not programmable. The general

purpose processor is programmable but cannot forward packets as quickly. Openet allows an administrator to program the router's general purpose processor with any number of *Oplets*, Java objects that implement some new functionality. These Oplets make use of the Oplet Runtime Environment and associated services to control the router. To maintain a high forwarding rate for packets that do not require additional processing, the router hardware provides programmable filters that trap certain packets but forward the remaining packets through the ASICs.

Intel's Phoenix [65] architecture for programmable networks provides a method of distributing code to network elements. Phoenix defines a Java-based proactive environment that provides a programming interface to the device for use by proactive services and mobile agents. Proactive services are Java objects that are resident in the element and export an interface for use by mobile agents. Mobile agents are Java objects that contain a script to be executed at each point along the agent's itinerary. Each script can program any of the network element's devices through a standard interface. While the interfaces are defined in Java, the environment, services and device interfaces may be implemented in native code to enhance performance.

1.2.2 Active Networks

Active Networks define a more flexible approach for distributing new code in a network. In a programmable network, an administrator explicitly loads the code into the network or network element under their control. In an active network the packet is responsible for identifying the code that forwards it. Two of the major issues in deploying active networks are code distribution and safety. The solution to the latter issue is typically related to the solution of the former, as the next few examples illustrate.

The Switchware architecture [5] uses an interpreter to solve both issues. Switchware's active messages, or *switchlets*, contain byte codes in the Caml [63] language. The Caml runtime environment, which is resident on each active element, interprets the byte codes and protects the host from certain security violations by enforcing strong typing, managing memory automatically, and restricting suspicious activity like modifying files. While most code can be loaded from switchlets, certain

security-sensitive modules must be loaded from the local disk. A problem with this approach is that in order to load a particular module, the interpreter must be configured to load that module. Boot-strapping the interpreter and network service on a node under different administrative control is challenging.

The Packet Language for Active Networks (PLAN) [25] takes a slightly different approach to security. Instead of merely relying on the runtime environment and limiting the supporting interfaces, PLAN restricts the language in which programs can be created. These restrictions, like the requirement that all programs terminate, provide some protection against mistakes and unbounded resource consumption. However, because of these restrictions, PLAN is primarily intended to act as a glue between services resident on the active node. Installing those services requires intervention by the node's administrator and therefore detracts from PLAN's flexibility.

ANTS [64], an active network toolkit from MIT, also relies on interpreter based protection. In this case the language is Java, but the idea that the interpreter protects the host is mostly the same. ANTS solves the code distribution problem by marking each ANTS packet, or *capsule*, with a hash of the Java object that implements the necessary forwarding code. Each ANTS router can then download the forwarding function from the upstream router that handed it the capsule. Using extensive caching, this approach makes sure that the correct forwarding code is always available for the capsule "just-in-time."

By using one of these architectures, a new service can be deployed to routers embedded in the flow of packets that requires it and without requiring special privileges on the executing router. However, active networks have not been widely deployed due to some unresolved technical issues and a lack of strong demand. The freedom that active networks give to any user to execute code on any router needs to be balanced by the power of an administrator to terminate ill-behaved processes. Part of that oversight ability that many administrators want is the ability to control who can load code into their routers. To date, no system has been developed that could support the right mix of permissive and restrictive policies to make an active router secure as well as usefully flexible. Also, no one has yet developed a "killer app," a popular application that required active routers, so none are widely available. This is an example of the classic chicken-and-egg problem. There are currently no active

routers, so there are no popular active applications. Since there are no popular active applications, there is no demand for active routers.

1.2.3 Overlay Networks

Overlay networks are another solution for deploying and testing new network services that circumvent the limitations of the previous approaches. Overlay networks are virtual networks that impose a new topology over the underlying network, or *underlay*. They may be administered by a central authority or be completely decentralized. They may provide general connectivity or an ad hoc service.

Within an overlay, each host's view of the network is restricted to include only other participating hosts. An overlay typically defines an address space in which it provides its own connectivity. Adjoining hosts in the overlay may be many network hops away from each other in the underlay, while hosts on the same subnet in the underlay may not be directly connected in the overlay. To overlay a network on the Internet, adjoining hosts on the overlay encapsulate messages between one another in standard Internet protocols. Using these tunnels, the overlay can make use of the underlay's connectivity without being entirely subjected to it.

To join an overlay network, a host must execute the software that implements the network's protocols. As with programmable routers and active networks, distributing this code properly is a challenge. In a centrally administered overlay network with dedicated hosts, it is natural for the central authority to push the proper code and connectivity information to each host. This approach provides users with a managed set of hosts on which to execute the overlay as well as a central means of administering all the nodes on the overlay.

The X-Bone [60] project provides a set of hosts for use in creating overlay networks. Using a simple Web interface, a user can create an overlay with the requested number of nodes in one of a few topologies. Hosts on X-Bone overlays communicate using IP within IP tunnels allowing the users to use the standard network tools to administer the overlay. X-Bone also provides an experimental capability to push software to the participating hosts.

PlanetLab [41] also allows its users to create overlays from a subset of a large set of distributed hosts. Within these physical overlays, PlanetLab users can create

virtual overlays, or *slices*, using only part of the resources of the physical overlay. Again, users use a Web interface to set up a slice by choosing the number and types of hosts to participate in the overlay. PlanetLab does not constrain the types of overlays its users can run. While PlanetLab also allows users to push software to their slice, it allows each slice to customize its own method of distributing programs.

Overlays that do not use hosts dedicated to running overlays are less likely to be centrally managed. Decentralized overlays are typically a part of peer-to-peer networks. These overlays form when users who wish to become part of a service download the software for this overlay and execute the user-level program. The software does not typically require privileged access to the overlay host and is therefore more likely to tunnel their traffic using the User Datagram Protocol (UDP) [42] than directly over IP. The advantage of this approach is that overlays can grow very quickly—a few clicks of the mouse can increase the size of the overlay and its user base.

1.3 Control and Data

The Internet's packet delivery is a classical example of a service that distinguishes between two classes of network traffic: control messages and data messages. Each Internet router periodically shares its view of the structure of the network with its peers in order to determine which of them is the next closest to a particular destination address. The exchange, an explicit communication between two routers, is conducted in a standardized way defined by a *routing protocol*, like the Border Gateway Protocol (BGP) [48] or Open Shortest Path First (OSPF) [36]. Based on the information discovered through the exchange of routing information, each router modifies the tables it uses to forward packets. From a router's point of view there are two classes of messages on the Internet: control packets that contain routing information, and data packets that contain other hosts' communications.

Router designers have leveraged the distinction between control and data to make routers more flexible and faster. In the common case, forwarding packets is fairly simple since it only requires a table lookup and a few modifications to the packet. Routing protocols are conceptually difficult to implement and are more computationally expensive. However, routing messages are relatively rare compared

to the many packets that a router may need to forward. Since forwarding and routing, that is control and data, are separable, data forwarding can be optimized and implemented in hardware while routing can be implemented in software. This balance allows routers to achieve maximum forwarding performance while maintaining the flexibility to use different routing protocols.

The services described in Section 1.1 share the characteristic that they have separable control and data components. However, many services, especially those created for overlay networks, ignore this well-established principle of separating control and data. The designers of these overlays view the services they provide as applications, without exploiting the fact that they are creating a new network. This viewpoint leads them to combine control information, like routing and application-specific searching, with data to be forwarded. This conflation of concerns limits the services' flexibility and performance. A major contribution of this dissertation is a framework for constructing services that recognize and exploit the distinction between control and data.

1.4 Thesis Statement

It seems likely that in the foreseeable future new network services will be developed using overlay networks. In designing and implementing these services it is important to remember the lessons learned from experience with the Internet. The development of hardware routers demonstrates that control and data can and should be distinct, so their differences can be leveraged separately. The success of the Internet services like e-mail and the World Wide Web implies that a single network can and should be usable for more than one application. History has shown that services are likely to be extended, even if just slightly. Applying these lessons to creating overlay networks and other network services can be very challenging.

In light of these observations, this dissertation makes three contributions. First, it shows how network services, especially overlay networks and their applications, can be decomposed into control and data planes and further decomposed into general and application-specific parts. Second, it proposes an architecture, called Plug Board, that provides a suitable framework for building network services that make

use of this decomposition. Third, it describes the potential benefits reaped by a network service written for Plug Board.

In the proposed architecture, a network service is comprised of a control component and a data component that communicate via a well-defined interface. To encourage service designers to use it, Plug Board provides two mechanisms for developing new network services: modularity and extensibility. A modular construction permits replacement of one service by another. To allow minor modifications and extensions, the data component of each service is extensible, meaning that it provides a hook to which data components from other services can be attached.

There are several benefits to writing network services in this way. Modular services introduce the usual software engineering benefits of code reuse. Also, well-defined interfaces allow the designer to place the two components in different computation domains, potentially on different processors. Finally, the decomposition and placement in separate domains allows optimization in the performance-critical data component.

1.5 Organization

The next chapter surveys the existing work and analyzes the structure of software-based, extensible routers. The analysis provides the terminology that will be used to describe network services and the proposed supporting architecture. Chapter 3 discusses the proposed architecture, the Plug Board, in detail. Chapter 4 uses the terminology and architecture discussed in the preceding chapters to show how to decompose a network service into four components. Finally, Chapter 5 describes two implementations of Plug Board and describes the benefits made possible by the decomposition.

Chapter 2

Background

This chapter presents the terminology needed to describe a router. This terminology provides a framework for reasoning about the nature and operation of routers in an abstract way. Later chapters extend this terminology and use it to describe a new extensible router architecture, called Plug Board. This chapter begins by motivating the need for extensible software routers. Section 2.2 derives descriptive terminology for routers from first principles. Section 2.3 relates the terminology to current work by using it to describe three software-based, extensible-router architectures: Scout [35], Click [32], and Router Plugins [15]. This chapter concludes by highlighting, for each of the three systems, the strengths and weaknesses that the description exposes. This serves to motivate Plug Board's design as presented in the next chapter.

2.1 Extensible Software Routers

Routers connect networks together by forwarding packets from one network to another. Most routers in the Internet today use special-purpose hardware that forwards packets at a high rate. However, the inflexibility of hardware impedes adding new functionality. Researchers, therefore, turn to software-based routers to experiment with new algorithms, protocols, and services. The effort required to create a stable, operational router is tremendous. Similarly, modifying an existing software-based router is fraught with difficulty.

The operating system community has had to deal with a similar problem. Modifying an operating system to support a new feature can be a challenging process since changes in one part of the system may have unexpected consequences on a different part altogether. Similarly, it is impractical to create a new operating system just to test a single feature. The solution is to create an operating system that is designed to be modified and extended. Extensible operating systems [11, 19, 35] allow researchers to test new ideas on a stable operating system that is easy to modify.

Using extensible operating systems as a template, and motivated by the demand for routers with new capabilities, researchers have been building *extensible routers* that aid in the design and development of network protocols and services. Similar to extensible operating systems, extensible software-based routers have architectures designed to ease the addition of new services into an existing router. Extensible routers provide a platform for implementing programmable, active, and overlay networks on a software router.

When considering different extensible router systems in the literature, it is evident that the designers addressed a common set of issues: what functions should routers execute on packets, how to specify the function for a particular packet flow, how to identify a packet as part of a flow, and so on. Different architectures address these issues in different ways. It is instructive to compare these systems. To do so in a meaningful way, it is useful to describe each in a way that highlights significant differences and hides less important ones.

2.2 Terminology

This section describes extensible routers based on intuitive assumptions of how routers in general, and extensible routers in particular, ought to work. The terminology is derived by modeling the operation of an extensible router as a sequence of connected software components and describing each component. It is important to note that there are many details that could be added to the description to make it more precise. However, only those details needed to make the terminology usefully specific are included.

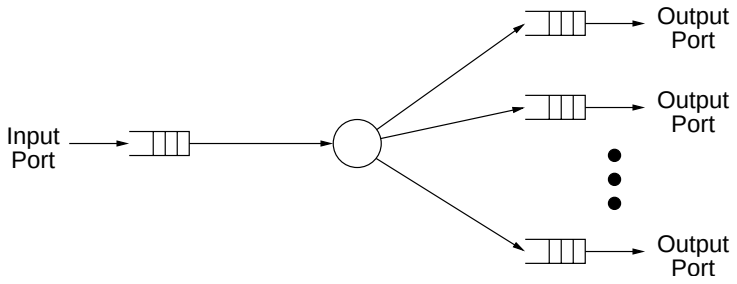


Figure 2.1: A simple, best effort IP router.

2.2.1 Components

In its simplest form, a router forwards packets from one of its input ports to one of its output ports. This process involves reading the packet from an input port, examining it to decide whether and whither to forward it, and writing the packet to an output port. Because packets from multiple input ports may contend for a common output port, packets are queued at the output port. The following description assumes that all queues are serviced in first-in, first-out (FIFO) order.

In this simple scenario, the router's primary task is to choose the packet's output port based on its destination address—a process called *classification*. To reflect this behavior, the first component of the model is a *classifier*, which dequeues a packet from a single input queue and places it on zero, one, or some subset of the available output queues. The classifier places the packet on no queues if it elects not to forward it (e.g., to implement a filter); on one queue in the common, single-destination forwarding case; and on more than one queue to implement multicast or broadcast. Figure 2.1 shows a simple configuration of a router with a classifier, denoted by a circle, associated with a single input port. In general, a classifier is bound to each input port.

While it is possible to associate a single FIFO queue with each output port, such a configuration is not general enough to describe complicated queuing strategies designed for preserving the quality of service guaranteed to a flow. In a router that allows certain packets to receive a different level of service than others—e.g., lower latency, a higher bit-rate, or more capacity—packets are not necessarily retransmitted in the same order in which they arrive. To allow modeling more complex types of queuing behavior, the model includes a second component that abstracts a

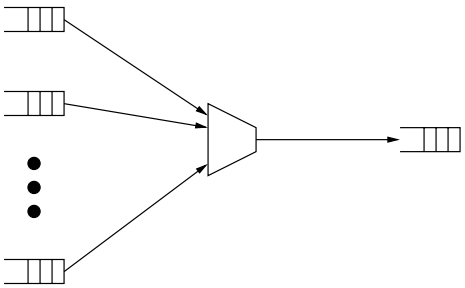


Figure 2.2: A packet scheduler.

router's queuing discipline. The scheduler can then be reasoned about as a separate component of the system. In this model, a *scheduler*, depicted in Figure 2.2, choses a packet from one of its input queues and places it on its single output queue.

Simply connecting a set of classifiers (one per input port) to a set of schedulers (one per output port) is sufficient to model a bridge or an Ethernet switch—network elements that do no processing on the packet beyond classification. A router is distinguished in that it modifies the packet in some well-defined way; we say it applies a *forwarding function* to the packet. A router may support more than one forwarding function. For example, most commercial routers have at least two forwarding functions: one that processes IP packets with options and one that handles option-free packets. Extensible routers extend this practice by allowing arbitrary processing. Thus, the last component in the model, a *forwarder*, encapsulates forwarding functions.

A forwarder removes at most one packet from its input queue and places one packet on its single output queue. The packet placed on the output queue need not be identical to that removed from the input queue, although in general most of the contents will be similar. Between reading and writing the packet, a forwarder may engage in any amount of arbitrary computation.

Forwarders implement a wide range of functionality. For example, the standard IP forwarder decrements the TTL field, recomputes the checksum, and replaces the source and destination addresses in the link-layer header. However, as already noted, routers are not limited to this base case. Applications like those described in Chapter 1 result in far more complex forwarders. For example, a proxy may modify the IP and TCP headers of a packet to reroute it to a different host, or the

proxy may rewrite the contents of the packet to request nearby cached versions of data. Several processing steps may also be composed to produce new forwarding behaviors. For example, a proxy forwarder that rewrites TCP headers may then pass the new packet to an IP forwarder.

A rich combination of forwarders implies that classification may also need to be complex. For example, a best-effort IP router connected via Ethernet uses the Ethernet type field to determine the correct processing for the packet and prefix matching of the IP destination address to choose the next hop for the packet. For many applications, classification can be based on both IP addresses and TCP ports. Other classifiers may require reading deep in the packet. For instance, a router being used as a front end to a cluster of load-balanced servers can read the HTTP [21] GET command from a packet and redirect it to an idle server. The classifier, therefore, must choose not just an output port, but also the appropriate sequence of forwarders.

Note that while many forwarders implement data plane functionality, others are part of the control plane. Control forwarders manage other forwarders in the router. They add and remove forwarders from the system, and they affect other forwarders by accessing and modifying state variables associated with them. For example, a control forwarder implementing the RSVP protocol may install a new forwarder with a particular resource reservation, or it may remove one that is no longer needed. As another example, a control forwarder that processes a routing protocol can install a new forwarder in response to the discovery of a new route, or it may add entries in the classifier to activate existing forwarders for a new class of packets. Control forwarders may also interact with schedulers by changing scheduling parameters.

Control forwarders superficially resemble data forwarders in that both have an input and an output queue, although the control forwarder's output queue may or may not be serviced by another component. The primary difference between control and data forwarders is in the interpretation of the final destination of the arriving packet. Control forwarders process packets that identify the local router as the final destination. Data forwarders process packets whose destination address is not interpreted as belonging to the local router.

It must be noted that merely the fact that the destination address of the packet is different than any of the router's addresses is not sufficient to render that packet

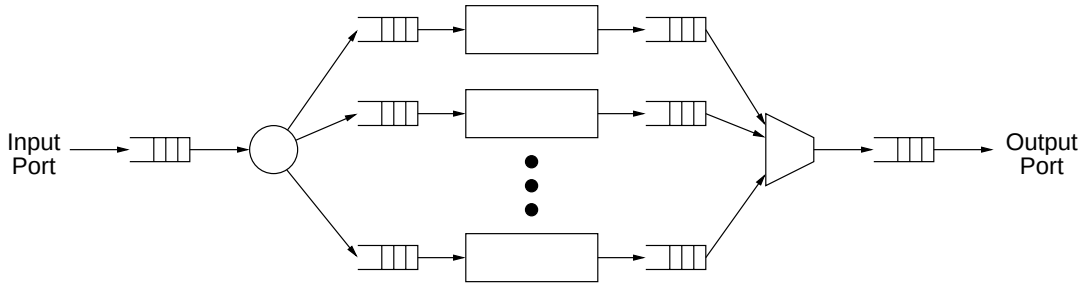


Figure 2.3: Modeling an extensible router.

subject to "non-local" processing. A router or host may decide to divert to a local process packets that would, based on their destination address, have been forwarded. Similarly, a host may forward a packet addressed to itself after suitably changing its headers. While these actions may violate some notions of network consistency, masquerading as another host can usually be tolerated. However, in circumstances where confidentiality is necessary, this ability could become problematic. In those cases, a host masquerading as another one can be treated as an attacker by the many mechanisms that exist to authenticate hosts.

Figure 2.3 shows a complete model of an extensible router. As mentioned above, this model is under-specified; for example, it does not specify how deep queues can be, whether other components can query a queue's properties and contents, nor how to model input and output devices. These and other similar details are not required to describe the general operation of routers, so the model leaves them unstated. Further, this figure shows only one possible combination of classifiers, forwarders, and schedulers; other combinations are possible. For example, a forwarder may modify packets then re-insert them in the classifier's queue. Also, sequences of forwarders can be linked together to create a complex forwarder, and a hierarchy of classifiers can be constructed by linking the output of one classifier to the input of another.

2.2.2 Extensibility

Describing the operation of a router in this way allows us to define extensibility. Extending a router means changing one of the primitive components. The details of how this "change" is accomplished are system-specific. From a platform-neutral

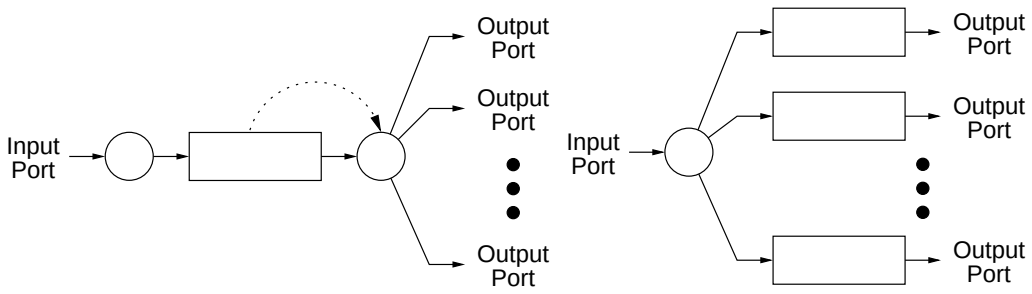
viewpoint, it can be said that an extensible router provides the framework and interfaces that make it possible to *modify* an existing component, *replace* one component with another, or *insert* additional components.

An administrator can modify a component either by giving it a new set of parameters or extending its functionality with new code fragments. A component may offer a programming interface that allows the administrator to set values on particular parameters used in its operation. For example, a classifier may export an interface to allow the administrator to add a forwarding table entry or modify a forwarding path's resource reservation. The ability to add code fragments to a component implies that it must have some externally visible structure that can be exploited.

If a component in an extensible router does not provide the desired functionality and cannot be made to simulate the desired functionality, it may be replaced. That is, the router might offer an interface that would allow exchanging the old component with a new one, one that does meet the new requirement. Consider an extensible router with a round-robin scheduler at one of its output ports. Assume that this scheduler has no externally visible substructure and no programmable interfaces. An administrator wishing to have a different kind of queue management, like Weighted Fair Queuing [17] or Deficit Round Robin [54], would simply replace the existing scheduler with a new one.

The power of extensible routers is that they offer the possibility of many components, operating in parallel, offering different types of behavior. To accomplish this, the router must allow the administrator to add components. For example, to add support for IP options to an extensible router that already has an IP forwarder that does not support them, the administrator would add a new forwarder capable of processing IP packets with options. The administrator would then change the classifier to choose the new forwarder for packets with IP options while leaving packets without options for the original forwarder.

While any component can potentially be extended in any way, some extensions are more common for particular components than others. An extensible router generally facilitates inserting additional forwarders. In contrast, classifiers and schedulers are typically either replaced or modified to support some new behavior.



(a) Simple Classifiers. (b) Simple Forwarders.

Figure 2.4: Two designs for a minimal IP router.

2.2.3 Engineering Tradeoffs

There is a range of possibilities for the power of the classifier and consequently the specificity of the forwarder. For a function of given complexity, a simpler classifier tends to require a more complicated, or general, forwarder. Conversely, a classifier with the ability to resolve fine distinctions between two substantially similar forwarders can drive a simpler forwarder to accomplish the same task. To illustrate this point, consider the pair of IP routers shown in Figure 2.4. Both routers implement the same minimal IP forwarding functionality. In the first design, Figure 2.4(a), the classifier always chooses the same forwarder for any IP packet. This forwarder, the IP forwarder, chooses the packet's output port and directs the second classifier to move the packet to that port. Another valid design would require as many IP forwarders as there are output ports. In the second design, Figure 2.4(b), the classifier chooses the IP forwarder associated with the proper output port. Since they need not choose an output port, each of these forwarders could be simpler than the one in the previous design. Simplifying the forwarder complicates the classifier and vice versa.

In general, any conditional that a forwarder may evaluate during packet processing could be implemented in the classifier. When choosing how much processing to put into classifiers and how much decision making to put into forwarders, there are two possible extremes. One extreme is to declare that all branches are classification and must be chosen before any processing is done on the packet; the other extreme

is to decide that only the decision that chooses an output port is defined as classification and that forwarders may have an arbitrary number of internal branches.

It is, of course, impractical to treat all branches in a router as the responsibility of the classifier and to associate a single forwarder with each combination of branch decisions. This strategy, followed naively, leads to a number of forwarders exponential in the number of branches in the router—in the number of "if" statements. However, in certain cases, most notably for IP packets that have no options set, having a specially optimized function could increase performance at a reasonable programming cost.

One good illustration of the choice is decrementing an IP packet's time-to-live (TTL) field. As part of forwarding an IP packet, a router must decrement a counter called the time-to-live, which is stored in a header field of the packet. IP routers must discard packets whose TTL decrements to 0. The packet could be processed and dropped by the forwarder, but since packets to be forwarded that arrive with a TTL of 1 will be dropped by the router, the classifier could decide to drop the packet by selecting no output queue for it.

A similar choice surrounds the option of permitting classifiers to modify packets. The description above implicitly describes the classifier as not modifying the packets flowing through it. Since forwarders can modify packets, there does not seem to be any reason to allow classifiers to do so as well. However, some computations are simple and universal enough to be included in a classifier.

Choosing the complexity of forwarders can also impact the number of forwarders that process a packet serially. An operation that is logically a single step but has many components can be in one complex forwarder or split across many forwarders. Again, a prime example is the simplest IP processing. As described above, even the simplest IP processing involved three steps: decrementing the TTL, recomputing the checksum, and changing the link-layer header. These steps can be in one forwarder, or they can be split across three different forwarders.

In the end, it is the system designer or programmer who must choose the component in which to place a specific block of code. On real systems, in which compromises must be made for efficient operation, a middle ground is usually chosen for pragmatic reasons. In the choice between complex classification and complex forwarding, one possible criterion is maximum computation time. Classifiers and

schedulers must complete in a short, bounded time in order to receive or send packets at the highest speed supported by the hardware. Therefore, any difficult classification decision should be left for a forwarder. Another criterion might be an architecture's inherent extensibility: if it is difficult to replace or modify the classifier at run time, then the function is best encapsulated in a forwarder.

2.3 Example Systems

This section describes three extensible routers using the terminology developed previously. The three systems—Scout, Click, and Router Plugins—were developed independently. Scout, developed at University of Arizona and, later, at Princeton University, was designed as a stand-alone operating system for network appliances but is also available as a Linux kernel module. Click, developed at MIT, uses a kernel module to replace the Linux kernel's networking subsystem. The Router Plugins architecture, developed by ETH Zürich, Washington University, and Ascom under the Crossbow project, extends the networking subsystem of a kernel based on BSD Unix to provide hooks for new forwarding functions.

2.3.1 Scout

Scout is a modular, communication-oriented operating system. Its central abstraction is the *path*: a linear flow of data that starts at a source device and ends at a destination device. Paths are composed of *stages*, which are instances of *modules*. Each Scout module implements a well understood protocol, such as IP or TCP. A module may contribute stages to many paths, but each stage is unique to a single path. The system creates paths on demand at run time. Scout's infrastructure provides tools to create, modify, schedule, and control paths.

In addition to providing stages to paths, each module also implements a classification function. The programmer defines this function based on the module's intended capabilities and operation. As part of path creation, each module places a reference to the path being created in a private data structure that it can access during classification. If the module cannot uniquely identify the path, the module stores a reference to another module's classification function that might be able to

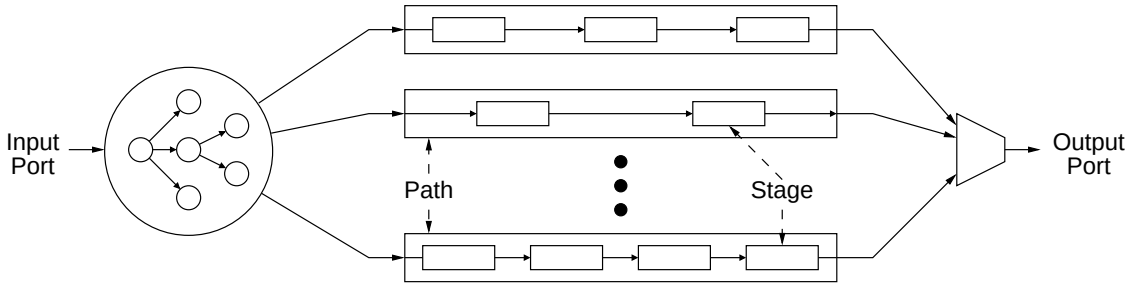


Figure 2.5: Modeling Scout: Paths are forwarders. (Queues elided for readability.)

resolve the ambiguity and find a path for the incoming packet. Scout's hierarchical, extensible, packet classifier queries each module for a partial decision based on the packet, then recurs to the next module. The recursion continues until either a path is selected or it is determined that the packet belongs to no path.

Once a path has been selected, the packet is placed on that path's input queue, and the path is scheduled to run. When Scout's thread scheduler decides that the path's thread is ready to execute, the thread dequeues the packet and hands it to the first stage in the path. This stage processes the packet and invokes the next stage where the process repeats. The stages process the packet in sequence, with the last stage placing the packet on an output queue. Eventually, the scheduler selects the packet from this queue and transmits it.

Figure 2.5 is a pictorial representation of Scout in terms of classifiers, schedulers, and forwarders. For each input port, Scout has one classifier with an internal hierarchical structure. For each output port, Scout has one replaceable scheduler. Each of Scout's paths implements a forwarder. Since each path is composed of stages, each stage could be modeled as a separate forwarder. However, equating paths and forwarders is more appropriate for two reasons. First, paths are distinct entities in their own right; they are not just collections of stages. Second, identifying paths with forwarders captures the important properties of Scout and allows Figure 2.5 to parallel Figure 2.3, the canonical representation of an extensible router.

2.3.2 Click

Like Scout, Click is a modular architecture for building routers. Its design is based on composing many simple *elements* to produce a system that implements the

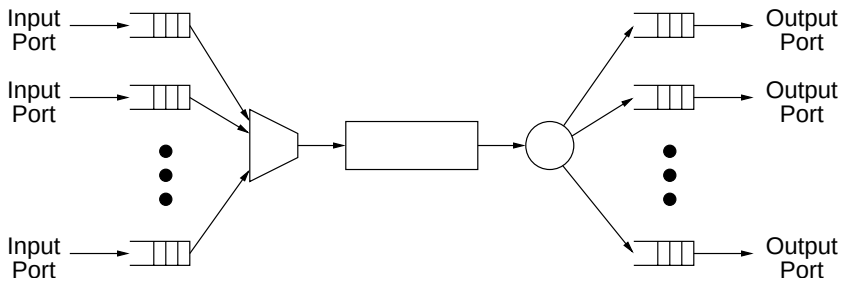


Figure 2.6: Modeling a Click element. (Some queues elided for readability.)

desired behavior. Each element may have multiple ports to connect it to other elements. Input and output ports may be either *push*-type (through which a packet must be sent) or *pull*-type (from which a packet may be requested). Pull-type and push-type output ports can connect only to pull-type and push-type input ports, respectively. Packets may be stored only in a queue, an element that has a push-type input port and a pull-type output port.

When a packet arrives at the router, the device driver pushes it through a series of elements. Since elements may have more than one output port, each element chooses the proper outgoing port as part of the packet processing. The packet is pushed through elements until it hits a queue, where it waits until it is pulled. When an output device is ready to transmit a packet, the output device driver pulls a packet from one of its upstream neighbors. The neighbor chooses an input port and pulls the data from the upstream neighbor on that port, and so on recursively until some element queries a queue. The packet is then removed from the queue and will traverse the same sequence of elements that selected it before reaching the device driver and being transmitted.

Each Click element encapsulates one scheduler, one forwarder, and one classifier, as illustrated in Figure 2.6. The scheduler is necessary because an element has to select a packet from potentially many input ports. Similarly, the classifier is necessary because the element may have multiple output ports. The forwarder simply connects the scheduler to the classifier. By leaving any two of these three components empty, a Click element can implement each of the primitive components of the proposed model, making it a very general architecture. It is more common, however, for a given element to define two of the three components, leaving only one empty. For example, elements with only push-type interfaces have the input port

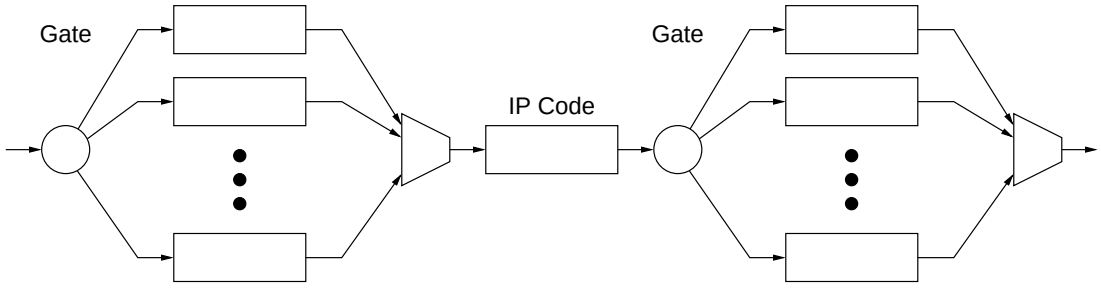


Figure 2.7: Modeling Crossbow: Two gates shown. (Queues elided for readability.)

chosen for them, so their scheduler is effectively null. Elements with only pull-type interfaces have their output selected for them, so they need not classify the packet at all.

2.3.3 Router Plugins

The Router Plugins architecture, or Crossbow, is designed to allow limited extensions to an IP router. Implemented in the NetBSD operating system, Crossbow allows users to write extensions, called *plugins*, that can be placed at well known points of the router's IP execution. These points, called *gates*, were chosen to suit a wide variety of applications, such as routing, packet scheduling, and security processing. The user invokes command line tools to load plugins at run time.

Plugins may be associated with distinct flows. Since not every plugin is appropriate for each packet, Crossbow allows the user to specify a key with each plugin. The key specifies IP address pairs, protocol number, ports, and incoming interface; any part of the key can be defined so that all possible values match. At each gate, Crossbow attempts to match the packet to the keys registered for that gate. If a key matches, the packet is passed to the plugin instance that is registered for that key. When the packet leaves the plugin, the next part of the IP processing path executes until the next gate. This sequence of events repeats until there are no more gates and the IP processing terminates.

As shown in Figure 2.7, Crossbow can also be modeled as a sequence of the basic components. A plugin operates on a single packet then emits it into the processing path, and is therefore a forwarder. Crossbow must choose a plugin at each gate, so a gate is a classifier. Upon leaving the gate, the packet must transit the next part

of the IP processing code, which is just another forwarder. Since all packets must execute the same IP code after choosing only one forwarder, the proposed model dictates that a scheduler must be present. Since Crossbow makes no scheduling decision at this time, a FIFO scheduler comes before the next fragment of IP code. Each defined gate in Crossbow is a classifier and a scheduler (FIFO), and each plugin a forwarder. The scheduler's output queue is connected to the input queue of the the next classifier by the next piece of NetBSD's IP infrastructure.

While this description of Crossbow is accurate, in some instances there is a better one. Assuming that the packet's IP and TCP headers are not modified by any plugin, the packet need be classified only once. The entire sequence of plugins executing on this packet is then known before the first one executes. The forwarding function executing on this packet can then be defined as the concatenation of all the plugins registered for that packet and all the IP infrastructure code. The FIFO schedulers between gates are no longer necessary, and the only remaining scheduler is the packet scheduler at the output port. Crossbow then looks like Scout, as depicted in Figure 2.5, except plugins replace stages.

2.4 Comparison

This section uses the descriptions of Scout, Click, and Crossbow presented in the previous section to compare them. The comparison focuses on two attributes common to all the systems: classification and extensible forwarding. The goal of the comparison is to highlight the strengths and weaknesses of each system in these areas.

2.4.1 Classification

A router's classifier determines the set of packets on which a given forwarder operates, and consequently, fundamentally limits the granularity at which flows can be distinguished. For a router to support a new network service, its classifier must be modified to recognize the set of packets belonging to that service. Scout, Click, and Crossbow all offer methods of extending classification. Crossbow and Scout both

prescribe a classification infrastructure that can be leveraged for extension, while Click defines its classifier for the task for which it is configured.

Crossbow's classifier selects a plugin at each gate based only on the IP addresses, TCP port numbers, and network input port. The classification algorithm allows fast lookups even in the presence of full or partial wildcard values in any field. Even with this flexibility, the function itself is fixed. Crossbow's classifier can be extended only by giving it new keys. The classifier will not act based on values in any other fields and cannot be made to act so within Crossbow's architecture.

In contrast to Crossbow's fixed classifier, Scout's single hierarchical classifier can be extended by providing it with new code fragments to execute during classification. During path creation, each module that contributed a stage to the path has the opportunity to add a function to the classifier to help identify the path. Delegating classification to the modules allows packets to be readily identified on the basis of any criteria the module programmer chooses. For example, packets for applications using TCP are identified by their port numbers; IP packets to be forwarded are identified by destination address; Ethernet frames are distinguished by their type number, and so on. Although modifying the classification function to recognize new types of packets requires writing new code, changing a specific module's classification function is easier than altering Crossbow's classifier. Modifications to Scout's classifier are confined to a single module, as opposed to affecting the entire system.

Unlike the other two systems, Click has no global classification structure. Instead, each element acts as a classifier to one of its outputs during packet processing. Inserting a new element into the element graph creates new paths through the router and explicitly extends the available classification. Reprogramming a specific element's "routing" decision is fairly easy, but one must be careful that the module still implements the function for which it was designed.

Both Scout and Click merge classification and processing. Click elements classify while they process by choosing output ports for the packet. While Scout modules do not classify as they process, they do define processing and classification together—they create a processing stage and define the criteria for selecting the path that owns that stage at the same time. Crossbow, however, neatly decouples the two. Crossbow's plugins do not implement any part of its classification. Instead, the

administrator, or user, specifies a particular key when instantiating a plugin. This separation makes plugins easier to implement and allows Crossbow's classifier to evolve without needing to reimplement plugins.

2.4.2 Extensible Forwarding

Since all three systems were designed for extensibility, it is not surprising that inserting new forwarders in each is fairly straight-forward. In Scout and Click, the administrator adds a new module or element to the configuration file. In Crossbow, the administrator installs a kernel module and instantiates a plugin. In each case, the new forwarder is inserted at the boundary between existing forwarders.

The number of available boundaries determines how much the router's operation can be extended by inserting new forwarders. Crossbow allows plugin insertions only at gates. Therefore, Crossbow can only be extended in very limited ways. Scout modules implement whole protocols, so adding entirely new protocols and layering one protocol on top of another is easy. However, since one can add stages to a path only between other stages, modules cannot modify the operation of existing modules. Each Click element usually implements a very simple functionality, so Click configurations require many forwarders to accomplish most tasks. However, the many boundaries makes it easy to modify the router's operation slightly just by adding additional forwarders.

Note that none of these systems offer a method of extending an individual forwarder beyond reimplementing it. Each system was designed to allow extensions to a larger unit of operation than an individual forwarder. Crossbow was designed to extend IP forwarding, so it has well-known points at which forwarding could be extended. Scout was made to ease protocol design and development, so it allows replacing and rearranging protocols. Click was designed so its elements would be too small to extend.

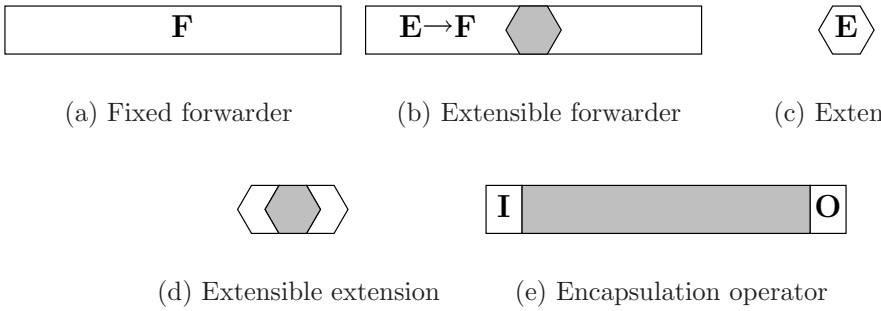
Chapter 3

Plug Board

This chapter presents Plug Board, a framework for creating and extending network services. Plug Board is an architecture for an extensible router that is a synthesis of the systems presented in the previous chapter. A router that implements Plug Board's interfaces provides explicit support for extension, including adding new network services and extending existing ones.

Plug Board's model of operation is based on the simplest variant of a router described in Chapter 2. That is, it has a single classifier, a set of forwarders, and a scheduler for each port. When a packet arrives, Plug Board queries the classifier for a forwarder to process the packet. Plug Board then invokes the returned forwarder, and the packet is processed until it reaches the scheduler at the output port.

This chapter extends the terminology presented in the previous chapter to describe Plug Board's interfaces for the forwarders and the classifier. Since schedulers are well understood, this chapter does not discuss them or their interfaces further. As an architecture, Plug Board permits more than one implementation, two of which are presented in Chapter 5. Therefore, Plug Board's interface specification presented in this chapter is somewhat abstract, highlighting the relationships between the components. The chapter concludes by exploring some implementation concerns and optimizations available to Plug Board-conforming routers.



(a) Fixed forwarder (b) Extensible forwarder (c) Extension

(d) Extensible extension (e) Encapsulation operator

Figure 3.1: Forwarders, extensions, and encapsulation. Shaded regions indicated places to insert a function of the appropriate type.

3.1 Extensible Forwarders

As described previously, forwarders in a router implement the router's functionality. Plug Board supports extensibility not just by hosting many forwarders but also by hosting many kinds of forwarders. This section defines the kinds of forwarders that Plug Board supports: fixed, extensible, and encapsulated. The definitions are followed by examples that make use of all three kinds of forwarders.

3.1.1 Forwarder Definitions

Forwarders are functions that are designed to process packets. They can implement either data-plane or control-plane functionality. The forwarders mentioned in the previous chapter have all been fixed forwarders, that is, they provided no method for extending the function they implement. Plug Board introduces *extensible forwarders*, forwarders that provide an explicit means to extend their functionality.

Extensible forwarders call an extension function, or extension, at a well-known point during their execution. The extension modifies the forwarder's operation by overriding parts of the forwarder's behavior or supplementing it with additional computation. If the type of a fixed forwarder is denoted by **F**, then an extensible forwarder is a higher order function of type **E→F**. That is, the administrator can construct a fixed forwarder by supplying the extensible forwarder an extension of type **E**. Using different extensions with the same forwarder yields a whole family of related forwarders.

Similar to the way that forwarders can be extensible, Plug Board allows extensions themselves to be extensible. An extensible extension permits nesting another extension within its operation. This nesting allows the router administrator to build complicated extension functions from smaller components. Extensible extensions, whose type is $\mathbf{E} \rightarrow \mathbf{E}$, can also be reused to create families of extensions that differ only slightly from one another.

Encapsulating a protocol in a packet of another protocol is fundamentally different from extending the encapsulating protocol. Extensions operate during the forwarder's processing. That is, the forwarder keeps state relevant to the processing of the current packet while calling the extension, so it can finish processing after the call. In contrast, due to the principle of separating protocol layers, functions that unpack a packet have finished their operation before the function for the next layer is called. Additionally, viewing an encapsulated protocol as an extension to an encapsulating protocol would require some additional forwarder that the outermost encapsulating protocol extends. Further, this view restricts the creation of gateways between two encapsulating protocols that both support a higher-level forwarding protocol.

Viewing an encapsulating protocol as the extension of an encapsulated protocol is similarly flawed. Packets encapsulated within packets of another protocol must be unpacked before they are processed and repacked afterward. This requires two distinct steps that must occur before and after the forwarding algorithm. Since an extension to a forwarder executes only once during protocol processing, it cannot both unpack the packet before processing and repack it afterward unless it implements the processing as well.

Fortunately, it is possible to leverage the internal structure of forwarders to provide a method for extending the set of protocols that can encapsulate the packet on which a forwarder operates. Forwarders, especially those whose packets are nested in multiple layers of network protocols, have distinct input and output phases that operate strictly before and strictly after the forwarding phase. These input and output phases can be separated from the forwarder and made available as distinct components. Although Plug Board does not provide a general capability for concatenating forwarders, it provides the *encapsulation operator* for this restricted purpose.

Name	Type	Description
Encapsulate	$\langle \mathbf{F},\mathbf{I},\mathbf{O}\rangle \rightarrow \mathbf{F}$	Encapsulates forwarder
Install Forwarder	$\langle \mathbf{P},\mathbf{F},\mathbf{D}\rangle \rightarrow ()$	Installs forwarder with predicate in domain

Table 3.1: Supported functions in Plug Board.

Kind	Type	Use
Forwarder	**F**	Processes packets
Extensible Forwarder	**E**→**F**	
Extension	**E**	Extends extensible forwarder
Extensible Extension	**E**→**E**	
Input	**I**	Processes incoming packets
Output	**O**	Processes outgoing packets
Predicate	**P**	Evaluates condition over a packet

Table 3.2: Types and Use of Functions in Plug Board.

The encapsulation operator is a function of type $\langle \mathbf{F},\mathbf{I},\mathbf{O}\rangle \rightarrow \mathbf{F}$ and can be used to create a forwarder from a forwarder, an input function, and an output function; see Table 3.1. Input functions, functions of type **I**, process a packet in a manner consistent with delivery to a local process. Usually input functions strip a header from a packet, verify that there are no errors using the protocol's error detection and correction scheme, and handle protocol specific processing such as reassembly. Output functions, of type **O**, process the packet in a manner appropriate to transmission on a network, typically adding a header to the packet. Using this operator, an administrator can extend a forwarder to operate in a new context.

The function types described above, which are shown in Figure 3.1 and summarized in Table 3.2, naturally lead to a simple notation for describing forwarders. Throughout the rest of the dissertation, named forwarders and extensions will appear as a pair of names, MODULE.name(). The first name is the name of the group, or module, from which the function comes; see Subsection 3.4.1. The second is the name of the function, which is unique within the module. The parentheses after the function's name contain the extension being applied. If the function is either not extensible or not currently being extended, empty parentheses follow the name.

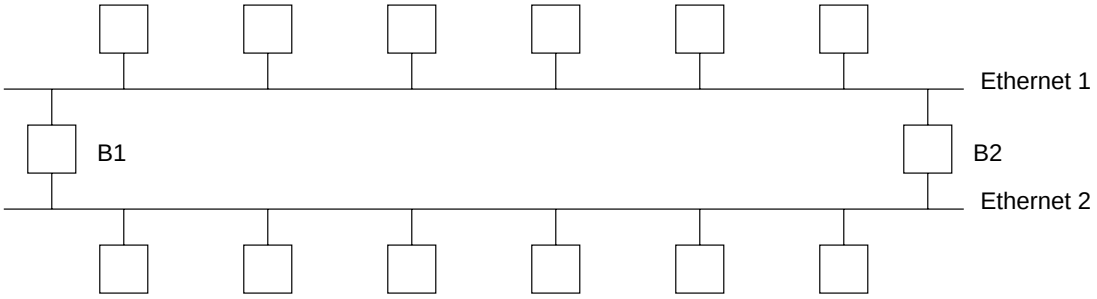


Figure 3.2: Two Ethernets connected by two bridges.

3.1.2 Examples

Plug Board is an architecture for supporting network services, both new and existing. While emergent services are discussed in Chapter 4, the following examples illustrate how to use forwarders and their related extension mechanisms to implement existing network services. Each of these examples is a well-understood network service that is currently deployed and can be implemented using Plug Board's forwarders and extensions.

Ethernet Bridging

An Ethernet bridge is a network device that is connected to at least two separate Ethernets. It connects the Ethernets by making copies of any frame received on one of its ports and sending it to the Ethernets connected to all of its other ports. An Ethernet bridge can be implemented by a simple Plug Board that has an Ethernet forwarder, ETH.copy(), for each output port.

The presence of multiple bridges connected to the same Ethernet can lead to frames being repeatedly echoed throughout all the connected Ethernets. Consider two Ethernets connected to each other via two bridges, as in Figure 3.2. Each bridge will echo all messages it receives on one Ethernet to the other. The other bridge will note the presence of the first bridge's message on the second Ethernet and echo it back to the first Ethernet, causing an infinite loop. To avoid such loops in Ethernet networks, Ethernet bridges compute the minimum spanning tree of bridges connecting all Ethernets. The vertices in the tree represent bridges and Ethernets while the edges represent ports that connect bridges to Ethernets. All

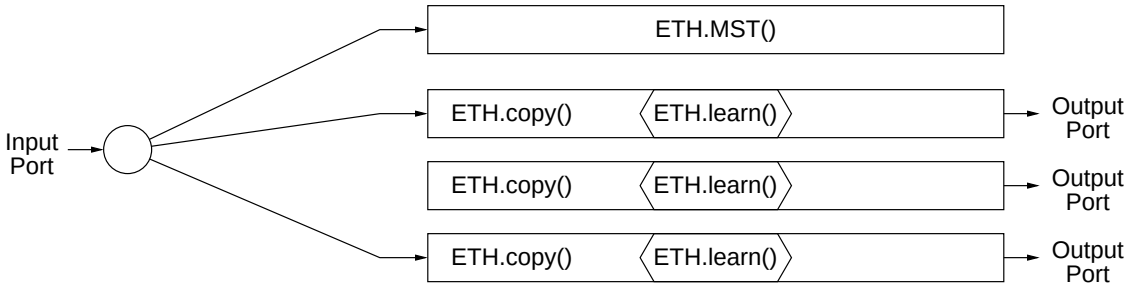


Figure 3.3: A learning Ethernet bridge: One control forwarder and many extensible data forwarders, one of which (the middle one) is disabled.

ports on the tree remain active while those not on the tree are considered inactive for the purposes of forwarding frames. Each Ethernet has only one bridge with an active link associated with it at any given time. A Plug Board that implements this behavior has a control forwarder, ETH.MST(), that inserts and removes Ethernet forwarders based on the current status of the tree.

With a minimum spanning tree in place, a bridged Ethernet transmits the same frame to all connected Ethernets only once. However, it sends the frame to all Ethernets even if the destination host is not reachable through a particular Ethernet. A learning bridge uses the flow of traffic to learn which hosts are available at a given port. It examines each frame to discover the source address and assumes that the source host is available on the Ethernet connected to the incoming port. To limit unnecessary network traffic, the learning bridge will only send data to the port at which the destination address can be found. Using extensible Ethernet forwarders, an administrator can create a learning bridge from a dumb bridge by installing an Ethernet forwarder extended with a learning extension, ETH.copy(ETH.learn()). The extended Ethernet forwarders will drop any frames headed to Ethernets that do not contain the destination. Figure 3.3 shows a learning Ethernet bridge with three network connections of which two are active, as shown by the forwarders' connection to the classifier.

Internet Protocol and Extensions

IP forwarding and routing provides another example of how to use Plug Board's extensible forwarding. In a simple IP router, there would be a single control forwarder,

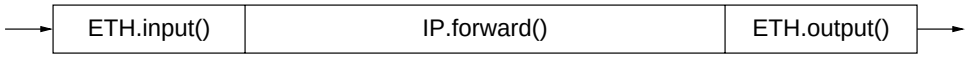


Figure 3.4: IP forwarder encapsulated in Ethernet input and output functions.

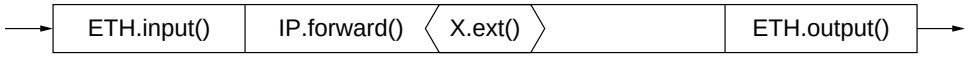


Figure 3.5: An extended IP forwarder for packets encapsulated in Ethernet frames.

IP.routing(), that implements a routing algorithm for determining proper forwarding. This control forwarder would add and remove IP data-forwarding forwarders, IP.forward(), in response to the routing state. For routers that are connected to the networks via Ethernet, the IP forwarders would be encapsulated within Ethernet input and output functions, as shown in Figure 3.4, like so:

[IP.forward(), ETH.input(), ETH.output()].

Basic IP forwarding consists of finding the network address of either the destination host or the next router, decrementing the time-to-live field, recomputing the checksum, and sending the packet out the chosen port. This minimum forwarding is frequently extended to perform other tasks, yielding forwarders of the form

[IP.forward(X.ext), ETH.input(), ETH.output()],

shown in Figure 3.5. There are at least three classes of extensions: traffic monitoring, data transformations, and rerouting.

Traffic monitoring typically involves gathering statistics about the traffic flowing through the router and reporting it to a monitoring application. Performance monitoring is a typical example [47]. In these services, the forwarding extension increments one or more counters based on some property of the packet. The property that triggers the counter can be the input or output port, the source or destination address, the packet's protocol number, the TCP ACK or SYN flag, etc. The monitor periodically aggregates these counters and sends summaries to a global coordinator. Based on analysis of many sets of these traffic statistics, the coordinator or the application then installs new forwarders. Intrusion detection often works in a similar way: the extension to the data forwarder records events; the monitor analyzes them and, in turn, installs filters in the router to block offending traffic.

In these services, the extension neither directly modifies the packet nor affects the operation of the forwarder.

Other services require that extensions modify the packet as it transits the forwarder. Data transformations include data compression, data transcoding, and packet elimination. For example, a pair of routers connected by a very low-capacity link may compress the data in packets before transmitting to avoid stressing the link. On these routers, the IP forwarding function would be extended with a compressor. Clearly the preferable solution to the problem of limited capacity is to add more parallel links or to replace the link with one with greater capacity. However, links that cannot be easily replaced, but have end-points that can be upgraded to support the memory and computation power necessary for compression, can benefit from this approach. Examples of such links are radio-frequency or satellite links between islands or ships.

Data transcoders are a generalization of compressors that leverage special knowledge of the capabilities of the receiving host. Consider a set of images available at a website. A PDA would likely not be able to display large images in many colors. The user would therefore benefit by receiving scaled or color-reduced images. This transformation would reduce capacity requirements on the PDA's network link and computation at the PDA. A signaling protocol that informs the router's control forwarder that a given destination is a PDA would allow that forwarder to install an extended IP forwarder that would transform data transiting the router. This enhancement would be especially useful at the router directly connected to the server since it would eliminate unnecessary data from the network. However, due to security and management issues, it is likely that only the router directly connected to the PDA would implement this type of extension.

A refinement of the transcoding technique can lead to removing the packet from the network. Consider a video stream that is encoded via the wavelet encoding scheme [20]. Stated briefly, in this scheme a video frame is encoded into several packets. The first packet contains data about those parts of the images that do not change frequently. Each subsequent packet contains more detailed information about the frame. The loss of one of the later packets in a particular frame causes some degradation of the details of the image, but not the loss of the entire, or large sections of, the image. This property implies that in the presence of congestion or

excessive delay on the network, specific parts of the video stream can be eliminated, that is some packets can be dropped, without affecting video quality too much. An IP forwarder with an extension that understands wavelet encoded video streams on a congested router can selectively drop those less important packets. The network's users would then see a gradual degradation of video quality as congestion increases instead of an immediate, sharp cut-off of the stream.

Some extensions change the fundamental behavior of the forwarder. Usually only the destination of an IP packet determines the address of the next host to which the packet is forwarded. This implies that the sending host has no control over the sequence of routers that receive a packet before it reaches its final destination. However, there exist two mechanisms, Loose Source Routing and Strict Source Routing [43], that allow the sending host to require that a given sequence of routers forward the packet. These mechanisms are activated as part of options field of an IP packet. A source routing extension to the IP forwarder allows packets to be rerouted from the most direct route to the source-specified route. This design is flexible and modular, making it easy to enable or modify the source routing option without directly affecting the design of the IP forwarder.

Proxies

Proxies are programs that run applications on behalf of their clients. From a user's perspective, a proxy is a server that provides access to data and services from the network. However, instead of providing these services itself, the proxy forwards requests to the server that can fulfill them. Proxies enhance network security by allowing network administrators to expose only a single host to an unsecured network like the Internet. They can also improve network efficiency by caching the results of requests and responding to subsequent, similar requests from a cache. Fundamentally, proxies transform and reroute data connections.

A classical proxy behaves as just described. The proxy is a forwarder that executes on a host with an address known to the users. When clients connect to the proxy, it forwards the packets containing the request to the proper host. The proxy reads packets that are encapsulated in application-specific protocols and forwards packets by rewriting them and encapsulating them in those same protocols. An

Figure 3.6: A classical HTTP proxy.

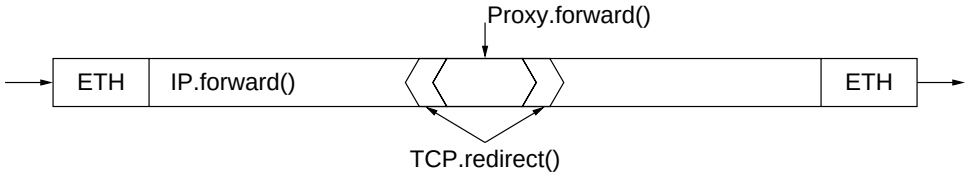


Figure 3.7: A transparent TCP proxy.

HTTP proxy's traffic is encapsulated inside TCP/IP packets inside Ethernet frames, so the forwarder (Figure 3.6) is represented as:

$$[[[\mathsf{HTTP.proxy()}, \mathsf{TCP.input()}, \mathsf{TCP.output()}], \mathsf{IP.input()}, \mathsf{IP.output()}] , \\ \mathsf{ETH.input()}, \mathsf{ETH.output()}]$$

A second class of proxy, called transparent proxies, modifies or reroutes packet flows passing through them. The proxy itself is an extensible extension of IP that redirects TCP traffic, as shown in Figure 3.7, and is represented as:

$$[\mathsf{IP.forward}(\mathsf{TCP.redirect}(\mathsf{Proxy.forward}())), \mathsf{ETH.input()}, \mathsf{ETH.output()}].$$

The extension rewrites the TCP header, taking care of the necessary changes to the checksum, sequence numbers and acknowledgment number. The extension can be further extended to support specific applications.

Although the two types of proxies can operate in much the same manner, they are different due to a quirk in their design: applications must be aware of classical proxies and send data directly to them. Transparent proxies, as their name implies, are completely transparent to the application. The difference is clearly visible in the notation and in Figure 3.8.

3.2 Classifier

Plug Board has a single classifier that chooses a forwarder for arriving packets. As in Scout, Plug Board's classifier operates before any processing occurs. An ideal

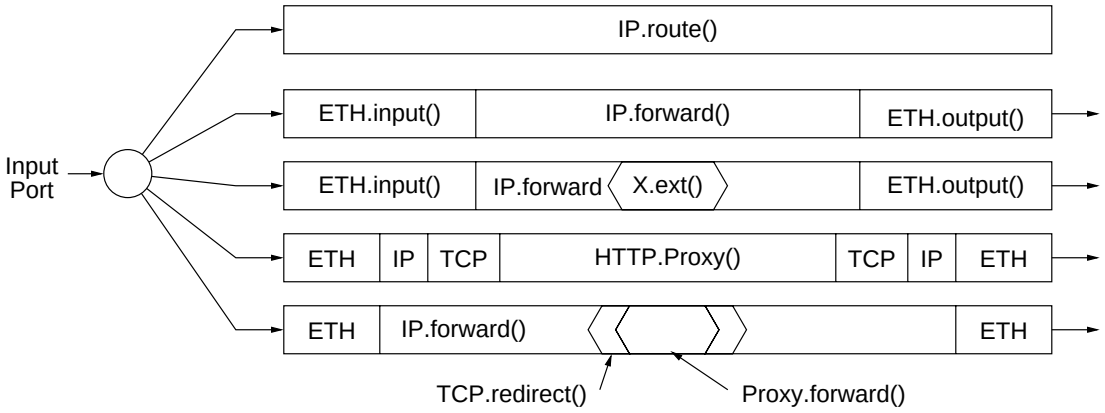


Figure 3.8: An extended IP router containing (from top to bottom): a control forwarder, an IP forwarder, an extended IP forwarder, a classical proxy, and a transparent proxy.

classifier for an extensible router would be an oracle that always returns the proper forwarder for a packet. Since any fixed set of classification rules fails to account for all possibilities that a router may address, to approach the ideal, Plug Board's classifier is modifiable.

An administrator can modify the classifier by associating a forwarder with a predicate defined over the full contents of a packet. For example, assuming the IP forwarding function, IP.forward(), and an appropriate predicate for choosing IP packets, $P_{\text{ETH.IP}}$, installing the pair $\langle P_{\text{ETH.IP}}$, IP.forward()$\rangle$ in the classifier creates a simple IP router. Predicates can be satisfied by any computable function. Some predicates are satisfied by single values, e.g., $P_{\text{ETH.IP}}$ that recognizes IP packets encapsulated in Ethernet frames using the Ethernet type field, $P_{\text{IP.IP}}$ that recognizes IP packets encapsulated in IP using the encapsulating IP's protocol field, and $P_{\text{IP.TCP}}$ that recognizes TCP packets. A more ambitious predicate is $P_{\text{Proxy.WEBA}}$ that recognizes HTTP requests to a site WEBA.

When a packet arrives at the router, the classifier returns the forwarder whose associated predicate is satisfied. Note that, as in Crossbow, classification and processing are completely separated—the forwarder associated with a predicate need not be functionally related to it. In the example above, the forwarder associated with $P_{\text{ETH.IP}}$ need not be an IP forwarder at all. It could be, for instance, a for-

warder that encapsulates the IP packet, including its headers, in another protocol like MPLS.

To make even more complex decisions possible, individual predicates can be combined with the boolean logical conjunctive and disjunctive operators—"and" and "or." For example, given a pair of predicates satisfied by IP destination addresses each to a particular network that is accessible from the same output port at the router, $P_{\text{IP.netA}}$ and $P_{\text{IP.netB}}$, a forwarder associated with the disjunctive predicate

$$P_{\text{IP.netA}} \cup P_{\text{IP.netB}}$$

can forward packets to either of those networks. Having a mix of compound and simple predicates makes it very likely that a given packet will satisfy more than one predicate. To allow the administrator to install forwarders that will catch otherwise unmatched packets as well as rules for narrowly defined flows, Plug Board resolves multiple matches in favor of the longest chain of conjunctive predicates. If the chains are of equal length, the packet could be given to all the forwarders, as in the case of multicast, some subset of forwarders, or just one. Since each policy is appropriate for some conditions, Plug Board allows the administrator to attach an exclusivity tag to the predicate.

Plug Board's exclusivity tag can specify that the predicate is exclusive, inclusive, or inclusive only to a group of predicates. If all satisfied predicates involved in a multiple match are inclusive, all the forwarders are given a copy of the packet. For example, a predicate for a network sniffer looking for traffic to and from a known host is satisfied by source or destination addresses to that host and is inclusive to allow IP forwarding to continue. If one of the predicates is exclusive then only its forwarder is returned. If more than one predicate is exclusive the predicate installed first is preferred, so forwarding does not break when an incompatible forwarder is installed. To support multicast behavior natively, a group of predicates may be named together such that members of the group are inclusive to each other, but exclusive to other predicates. Using group exclusivity, control forwarder implementing IGMP can install multiple data forwarders for an IP multicast group address. A multicast packet to that group will satisfy all those predicates and will be forwarded by all the associated forwarders.

3.3 Computation Domains

Modern routers provide more than one computation domain for executing forwarders. For example, Karlin [29] describes VERA, a router architecture with multiple, physically separate, computation domains. In general a computation domain is an abstract description of "where" computation takes place. Computation domains differ in their physical location, in the resources they provide forwarders, and in the trust they grant forwarders executing in them. Forwarders should execute in the domain that offers the best balance between the performance of the forwarder and protection from it. Therefore, Plug Board must support multiple computation domains.

Computation domains are separated by boundaries, which have a cost associated with their crossing. The cost includes delays caused by moving to a new physical location and those caused by changing the level of trust such as copying data to different memory regions, switching execution threads, and trapping into the kernel. Each domain in a router may have different resources available for processing packets. In some domains a forwarder's "cycle budget," the number of instructions a forwarder may execute, is large—many thousands of cycles. In others, fewer cycles are available. Since domains may be on different processors, the rate at which these cycles are apportioned may also be different.

Forwarders executing in separate domains may be given different levels of trust. In the most trusted domains, forwarders may execute any operations on any part of memory. These domains, for example, kernel-space in an operating system, provide the resources that allow maximum performance. However, they can only be used for trusted code that is well understood since the domain allows the forwarder to commit any action. Other, less trusting, domains restrict more and more of the instructions a forwarder may execute and enforce stricter limits on accessing regions of memory. The most restrictive domains may check each instruction before execution and allow limited access only to very specific regions of memory.

3.4 Implementation Issues

As demonstrated in the examples above, a Plug Board is uniquely appropriate for implementing network services. The rest of this chapter highlights some implementation concerns and some optimizations that the Plug Board makes possible.

3.4.1 Modularity

To fully reap the benefits made available by extensibility, a Plug Board must provide methods of adding new functions not conceived by its implementer. Good software-engineering practices dictate that logically related functions should be grouped together. Therefore, a Plug Board should support a modular extension architecture. Each module is a factory to create the various functions—predicates, forwarders, extensions, etc.—for handling a particular protocol. Placing all the functions related to a single protocol within the same module improves the functions' maintainability while maintaining the Plug Board's flexibility.

A module encompasses all the functionality required for a single protocol. For instance, the Ethernet module should provide functions for input and output processing and should also be able to provide predicates for distinguishing an Ethernet frame carrying an IP packet from one bearing an ARP query. Grouping closely related functionality encourages programmers to write modules that are independent of the implementation of other modules. Since all protocol-dependent code is grouped together, changes to one part of the protocol processing code will be unlikely to break code in another module.

A disadvantage of most modular systems is that the increased processing overhead due to function calls between modules and duplicated error checking leads to decreased performance. Since Plug Board uses modules only as a way of organizing related functions, it permits users to create optimized versions of forwarders to operate in specific circumstances. For instance, there is no restriction in Plug Board forbidding the creation of an IP forwarder that also handles all the necessary Ethernet processing. This forwarder would operate quickly and is a candidate for use in the fastest IP forwarding path in the router. The same factory module can create this forwarder as well as more general IP forwarders, thus providing a balance between modularity and performance.

3.4.2 Forwarding Functions

Plug Board's modules support creating data forwarders from any protocol. For protocols that purport to provide data forwarding behavior, like IP or peer-to-peer networks, this requirement seems intuitive, indeed necessary. For protocols not viewed as forwarding protocols, like Ethernet and TCP, this requirement seems very strange. This feature originates in the need to keep module interfaces consistent, but it can be used to ease implementation of bridges. For example, the Ethernet module can define an Ethernet forwarding function that allows a router to act as an Ethernet bridge, even a learning bridge, without modifying the hardware.

Allowing modules to create forwarders from protocols that do not usually define forwarding components can also lead to optimized versions of protocols that splice two connections. This optimization is known as the *cut-through* optimization from the similarity to the cut-channels provided by the NodeOS, an active network standard [4]. In the NodeOS, a program can open two communication channels, one for input and one for output. The application can then read data from the input channel, process it, and write data to the output channel. If the application reverts to a mode in which it retransmits the data it receives unchanged and unmonitored, it can splice the two channels together. The data then no longer arrives at the application, meaning that the cost of crossing into the application's computation domain can be eliminated.

One example of a protocol that can benefit from the cut-through optimization is TCP [12, 56]. Two TCP connections, one from host A to host M and a second from host M to host B, can be turned into a single connection from host A to host B; see Figure 3.9. Host M is now no longer an active member in a TCP session, so it would not need to guarantee reliable, ordered delivery—it need only try its best to forward packets from A to B. Consequently, host M would save memory required for buffering, computation used to track TCP state, and network capacity needed to send acknowledgments. Also, since the single splicing function need only adjust a few fields in the packet's header, it is not complicated. As described in Appendix A, it can be implemented in a few dozen instructions on a typical network processor.

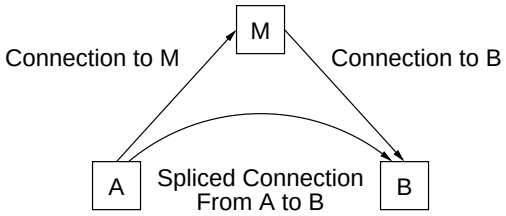


Figure 3.9: Splicing two separate TCP connections into a single connection.

3.4.3 Classification

Evaluating all of the available predicates at each packet arrival can be inefficient. While this effort might be necessary in the general case, usually a more efficient method is available. Network protocols are layered; one protocol is completely embedded in the payload section of another protocol's packet. Thus a hierarchical classifier is well-suited for classifying network packets. In a hierarchical classifier, each level of the hierarchy deals only with the header of a single protocol. A packet being classified would be shunted from node to node of a tree-like structure, each node containing classification code from a different protocol. A leaf in the hierarchy would contain the proper forwarder for the packet. Composite predicates over the headers of the packet can be ordered in a canonical form to force similar entries to have a similar prefix of predicates [8, 10], thereby allowing a hierarchical form and improving efficiency. [23]

Chapter 4

Decomposing Overlay Networks

The last several years have seen the introduction and popularization of peer-to-peer applications and their associated overlay networks. Overlay networks are virtual networks that impose a new topology on the underlying network, or *underlay*. This topology typically excludes routers in the underlay in favor of using hosts to forward packets for the overlay. These application-level networks provide features and capabilities that the Internet does not, like logarithmic routing and in-network caching, without waiting for standardization and deployment by vendors.

Applications that use overlays to implement services blur the traditional distinction between the network and application programs by turning hosts executing the application into routers as well as clients and servers. In addition to providing their service, overlay applications must maintain the structure of the network in the presence of dynamic membership, that is, hosts joining, leaving, or becoming unavailable. Overlay applications must also forward packets for other hosts on the overlay in a way that is consistent with the overlay's semantics. Supporting these tasks, in addition to providing the service they claim to provide, makes these applications complex and difficult to create. However, the application's additional complexity is offset by the flexibility that it has to define the overlay network's topology and the capabilities of the overlay's forwarders to assist in providing its service.

This chapter shows how to leverage Plug Board's architecture to ease the creation of overlay networks. The chapter begins by presenting a decomposition of network services in general, and specifically overlay networks, into four components.

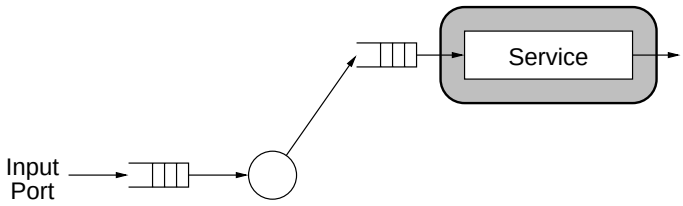


Figure 4.1: A network service implemented as a single forwarder.

Each of these components has interfaces it must support and a role it must play. By leveraging Plug Board's architecture, this decomposition can be used to create flexible and extensible overlay networks. Proceeding from the decomposition in the abstract, the chapter then presents three concrete examples of decomposing peer-to-peer networks. It concludes by discussing the interfaces between the decomposed components.

4.1 Implementation Forms

The applications that implement overlay networks take one of five progressively more complex forms. The forms differ in the performance they offer to clients, in the safety they guarantee to the executing host, and in the maintainability they claim to developers. The forms do not differ in their correctness or compatibility. In fact, it is expected that an overlay network would appear in more than one of the forms during its development lifetime. Thus, the implementation of a network service can become increasingly optimized on one host while existing in other forms elsewhere in the network.

4.1.1 Form I: A Network Service

In the first implementation form, a network service is a single functional block. This block implements the offered service as well as any protocol that the service requires, such as connectivity maintenance for an overlay network. In other words, the network service is implemented as a single fixed forwarder in Plug Board. This forwarder may be encapsulated in several pairs of input and output functions, typically those that implement standard transport protocols, such as TCP. Since this

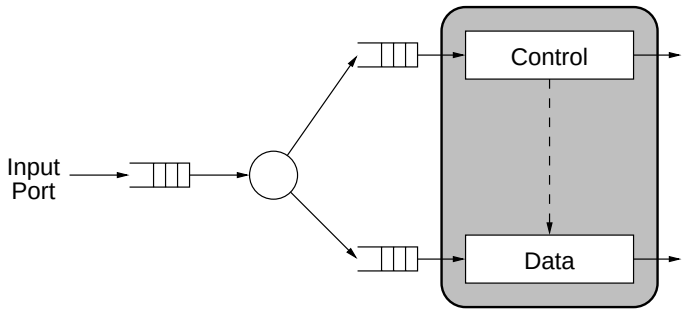


Figure 4.2: A network service differentiating between control and data.

forwarder processes all the service's incoming network traffic, its associated predicate is satisfied by any of the overlay's packets addressed to the local host in the underlay. The forwarder executes in its own computation domain, separate from the classifier which executes in the domain in which all other computation domains are embedded. Due to its simplicity, this form, shown in Figure 4.1, is a reasonable choice for a first-generation implementation of a network service.

4.1.2 Form II: Control and Data

Applications that implement overlay networks act analogously to routers in traditional networks. That is, as described in Section 1.3, they forward part of the incoming network traffic—the overlay's data stream—to other hosts and process the rest—the control stream. Recognizing that such applications act in two distinct modes leads to the second implementation form. In this form, shown in Figure 4.2, the network service is divided into two pieces. The first component, the control forwarder, is a slightly modified version of the forwarder in the previous form. Although the control forwarder handles most of the same tasks as in the previous form, it no longer forwards messages destined for other hosts on the overlay. The latter functionality is delegated to a second forwarder—the *overlay forwarder*. However, the control forwarder responds to messages and generates network traffic, so it still needs its own network connection.

The control forwarder installs one or more overlay forwarders to forward packets to other hosts in the overlay. The number of overlay forwarders depends on the service and its implementation. There may be one forwarder for all forwarded

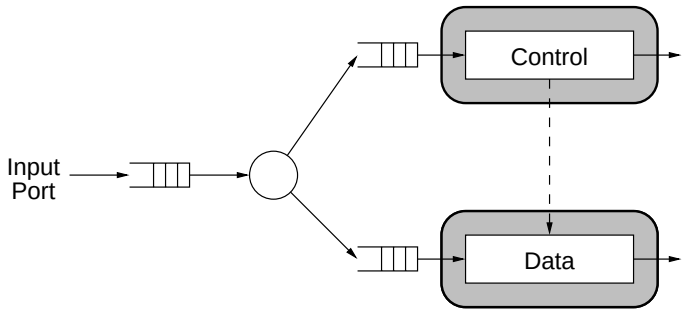


Figure 4.3: Optimizing forwarding in a network service using multiple computation domains.

traffic, or there may be a forwarder for each destination. The predicate that the control forwarder associates with each overlay forwarder is satisfied by packets that are addressed to the local host in the underlay but should be sent to another host in the overlay. When executed, an overlay forwarder modifies the arriving packet using a service-specific forwarding function and sends it to the next overlay host that should receive it.

4.1.3 Form III: Optimizing Forwarding Performance

In the second form, the control and overlay forwarders are distinct entities. As described in Chapter 3, Plug Board allows for multiple computation domains, with different costs associated with crossing into each one. Placing the overlay forwarder in a high-performance computation domain with lower crossing costs improves forwarding performance by reducing the time each packet spends at the host and increases efficiency by limiting the resources the host spends on forwarding each packet. However, for performance and safety reasons it may be undesirable to place the complex control forwarder in such a domain. Therefore, in the third implementation form, shown in Figure 4.3, the overlay forwarder is placed in a domain separate from that in which the control forwarder executes.

The resources available to forwarders in high-performance computation domains are necessarily scarce since forwarders in these domains are expected to forward many packets very quickly. Thus, even though the overlay forwarder can now begin execution without having consumed as many resources, the balance of the resources available for its execution may still be limited. However, just as IP forwarding is a

fairly simple procedure from a complex protocol, one would intuitively expect that other data forwarding functions would also be relatively simple. In fact, as shown in Chapter 5, data forwarding functions tend to be simple enough that reasonable limitations on the available execution time and memory consumption do not prove a hindrance to applying this optimization.

As a network service becomes more popular, and thus executed on many hosts and forwarding increasing amounts of data, the improved performance of this form becomes attractive. However, as the option of placing the two forwarders in separate computation domains is purely a performance consideration, it should be balanced by other concerns. The considerations underlying the choice of placing some forwarding code inside an operating system kernel, a possible computation domain, illustrate this tension.

Placing the forwarder in the kernel would save the overhead required to switch to the user-space forwarder and the cost of copying the data there. This potential benefit must be balanced against security and trust issues. Executing arbitrary application code in the kernel can be dangerous. The code may not be stable and may cause the host to be unavailable frequently. Even if the code is stable, it may interact badly with other applications, which can be dangerous given the additional authority that the kernel possesses. Even if the code is stable and operates exactly as designed, it may be actively malicious, compromising other applications and the data available to the host. Certainly there are techniques to minimize these risks, but the infrastructure required to protect against undesirable behavior may degrade performance for the entire system. The decision to place a forwarder in a trusted, but high-performance, domain must be carefully considered.

4.1.4 Form IV: Multiple-use Networks

A key flaw that services designed in the two previous forms inherit from the original monolithic application is that the offered service is entwined with the overlay that the service requires. This dependency makes it difficult to port the service to a different overlay network, even one with features, such as automatic dynamic membership management, that would benefit the service. Conversely, other applications that could use a similar overlay network would need to implement the network man-

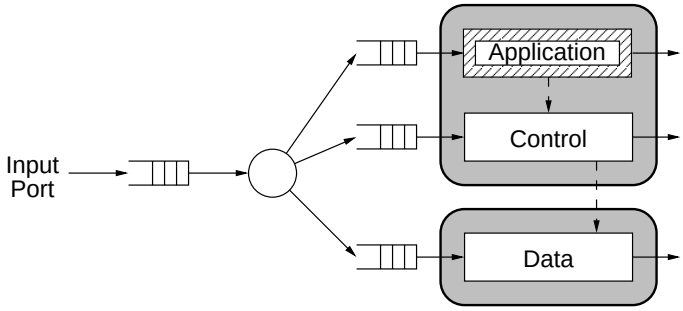


Figure 4.4: An application (shaded rectangle) using a separately defined overlay.

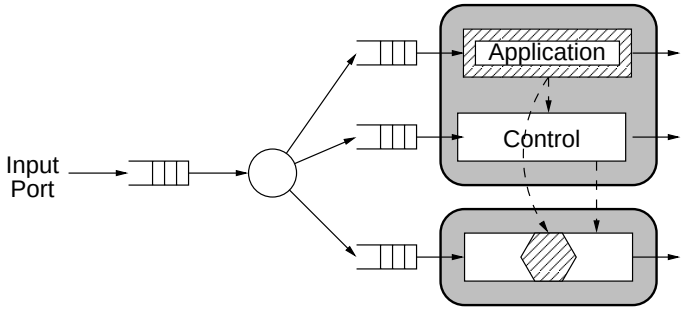


Figure 4.5: A network service with an application-specific forwarding extension.

agement functions independently. The solution is to decouple the task of providing a service from the tasks of managing the overlay network.

The fourth form, shown in Figure 4.4, further refines the control forwarder by dividing it into two distinct forwarders. The first forwarder, the *overlay daemon*, manages the overlay by maintaining routing and connectivity information and processing other network management messages; it also controls the overlay forwarders. The second forwarder is the application. The application implements the offered service, such as serving data, storing files, and answering database queries. The predicate associated with the application is satisfied by packets that are addressed to the current host in the underlay and contain application data. Packets containing network state and configuration information satisfy the predicate associated with the overlay daemon.

4.1.5 Form V: Application-specific Forwarding

Isolating the application from the overlay forwarder makes the overlay more general and the application easier to port. However, this same isolation prohibits the application from examining or modifying the data stream forwarded through the host. Some applications that use peer-to-peer networks rely on the application executing at each host having access to the forwarded data stream [18, 51]. Applications can use this access to improve their performance, for example by caching responses to network queries or diverting requests directly to the local host.

In the final implementation form, depicted in Figure 4.5, the fixed overlay forwarder is replaced by an extensible forwarder. The overlay forwarder can then be extended by an application-supplied extension—the *application forwarder*. The application forwarder can modify the standard behavior of the overlay forwarder, making it possible for the application to affect forwarding indirectly. In this way, a general-use overlay network can be customized for a particular application.

The ability of an application forwarder to customize a general-use overlay provides a useful guideline for distinguishing between overlay forwarders and application forwarders. An overlay forwarder provides the minimal mechanism necessary for moving a packet from one host to another in the overlay. It modifies the packet's headers only to the extent necessary to forward it, and it maintains only that state useful for forwarding other packets. An application forwarder, by contrast, modifies an existing forwarding algorithm in special cases to benefit the application. It accesses the packet's payload, and maintains any state, not just forwarding state, that is of benefit to the application.

Since application forwarders modify the overlay forwarder, it is reasonable to fear that they may compromise the safety of the overlay forwarders. Three factors combine to vouch for the safety of many application forwarders. First, as shown in Chapter 5 even useful application forwarders are not necessarily complex and are verifiably safe. Second, as discussed in Subsection 4.3.4, application forwarders have specific classes of tasks. Reasonable safety assurances can be provided by forcing access to the packet and these tasks to use a trusted implementation of an application programming interface (API). Third, even if the application forwarder does not implement its function correctly, the predicate that selects packets for the

extended overlay forwarder only selects packets from the application's data stream, so other applications do not suffer. Consequently, extending an overlay forwarder can be considered safe much of the time.

4.1.6 Summary

This section shows how an application that provides a network service based on an overlay network, such as a peer-to-peer application, can be decomposed into four components: an overlay daemon, an extensible overlay forwarder, an application, and an application forwarder. The overlay daemon maintains the application's network by managing routing and connectivity information, heartbeat signals, and other network management messages. The overlay daemon also controls the overlay forwarders. The overlay forwarders move packets addressed to other hosts in the overlay to the proper host. At the packet's destination, it is processed by the application at that host. In order to process messages that are still in transit, applications can extend the overlay forwarder with an application-specific forwarding extension, or application forwarder.

Plug Board is uniquely capable of supporting decomposed network services. The control forwarder, overlay forwarder, and application are each forwarders; Plug Board supports installing multiple forwarders. Plug Board supports using extensible forwarders to create families of applications that extend the same overlay network. It supports a programmable classifier that can match packets with the correct forwarder based on overlay-specific and application-specific criteria. It also supports multiple computation domains so that the forwarders can be executed in a domain with the right balance of trust and performance.

4.2 Example Peer-to-peer Applications

This section presents concrete examples of the decomposition presented in the previous section. This section decomposes three overlay networks and some of their associated services and recasts them in the final decomposition form above. Since any of the earlier forms can be reconstructed by merging components from the final form, these examples demonstrate that the entire decomposition process is feasible.

4.2.1 Pastry

Pastry [49] is a toolkit for building peer-to-peer applications. Each host in a Pastry network is assigned a 128-bit identifier, which is interpreted as a number in base 2^b where b is a parameter of the network. At each host, messages are forwarded to a host whose identifier shares at least one more base-2^b digit with the requested destination than the local host. Eventually, the message arrives at the node in the network whose identifier is the closest available to the requested destination. Pastry's routing behavior implies that messages are forwarded $O(\log_{2^b} N)$ times in an N-host network, leading to remarkable scalability. Pastry defines protocols and procedures to maintain network connectivity and routing, and it has an algorithm specifically for forwarding data through the network. Pastry applications can make use of the network and are involved in the forwarding of each packet.

Each of the preceding qualities of Pastry can be implemented in a separate component, as depicted in Figure 4.5. The overlay daemon processes Pastry traffic that is addressed to the local node on the Pastry network but is not specific to the application (e.g., requests to join the Pastry network) and maintains heartbeat signals to the hosts in the leaf set (i.e., the hosts with identifiers numerically close to that of the local host). The overlay forwarder handles those messages for which the local host knows of another host that has a closer identifier, regardless of whether the messages are standard across all Pastry applications or are application-specific. The overlay forwarder finds that host and sends it the message.

PAST [50] is an example decomposable application that uses Pastry. PAST provides a persistent storage service using a Pastry overlay. The overlay daemon and forwarder were already described. The PAST application provides the persistent storage service by processing file insertions, lookups, deletions, and so on. PAST's application forwarder caches responses it sees in the local host's storage and redirects requests that can be satisfied from the cache to the local host.

4.2.2 Chord

Chord [58] is a scalable, distributed lookup service. It allows hosts to be added and removed efficiently while the system is running and is robust in the presence of host failures. Chord's lookup algorithm guarantees a search time logarithmic in

the number of hosts in the network. The algorithm can be either iterative, with the originating host querying each successive host, or recursive with each host locating the next host that might be able to respond to a query and forwarding the query to it. In its recursive form, Chord forms an overlay network that forwards queries for the location of the object corresponding to the requested identifier.

In Chord, the overlay daemon processes join and leave requests from other hosts. It maintains the information that each host uses to forward packets (e.g., the finger table) and notifies the application of changes in the set of keys for which it is responsible. In the algorithm's recursive form, the overlay forwarder processes incoming queries and forwards them to the next host in the overlay whose identifier most closely matches the requested identifier. The application transmits the data associated with the query key to the host that initiated the search. In the basic recursive Chord algorithm, there is no application forwarder as in Figure 4.4.

4.2.3 Gnutella

Gnutella [31] is a peer-to-peer application for sharing files. The network it creates is unstructured; there is no clear relation between a file's contents or attributes and its location in the network. There is no routing protocol; hosts are aware only of the small number of peers connected directly to them. All queries are broadcast to the network with a counter indicating the maximum number of allowable hops. Successful responses to queries are returned via the same path the original query took. To prevent duplicate messages and forwarding loops, each packet is tagged with an identifier. A host seeing packets with the same identifier more than once removes the duplicate messages from the network. The same mechanism can remove responses that were not generated by any request.

The decomposition for Gnutella is depicted in Figure 4.5. The overlay daemon for a Gnutella network processes pings, that is, it handles joins to the network by adding hosts to the list of connected peers. The overlay forwarder forwards queries to all its other connected peers and returns responses to the port through which the associated query originated. The application replies to successful queries with the appropriate file information. The application forwarder diverts successful queries to the local application and stops their forwarding.

4.3 Interfaces

Given the decomposition from the previous section, this section now describes the types of interfaces that each component imports from the other components and exports to them. The example interfaces will use the type **packet_t** to refer to packets. The host system should provide methods to access and modify the contents of the packet.

4.3.1 Overlay Daemon

As explained previously, the overlay daemon is responsible for maintaining the host's connection to the overlay. It maintains the host's routing information, that is, the host's perception of the state of the overlay. Using Plug Board's interfaces (Table 3.1) the overlay daemon installs and removes overlay forwarders in response to routing changes. The overlay daemon may also modify the state that the overlay forwarders use in their operation. Specifically, the overlay daemon may update the overlay forwarder's forwarding table by adding, removing, or replacing an entry. More generally, the overlay daemon may modify any parameter, e.g., type of service, that the overlay forwarder associates with the forwarding of a particular class of packets by using the following interface to the overlay forwarder:

bool UpdateEntry(**void** $*$ key, **const char** $*$ parameter, **void** $*$ value)

The overlay daemon must maintain proper forwarding during the presence of error conditions in the network. For those error conditions that only the overlay forwarder can detect, like corrupted or misrouted data, the overlay daemon should export an error-reporting interface to the overlay forwarder. The error reporting interface should include the type of error encountered, which is overlay-specific, and the contents of the packet, including all its headers, like so:

void ForwardingError(**int** ErrorType, **packet_t** packet)

4.3.2 Overlay Forwarder

The purpose of an overlay forwarder is to transmit a packet to the next host on the forwarding path in the overlay. The process of forwarding can be described generally

as having four steps. When a packet arrives at the host, the overlay forwarder first computes the address of the next host in the overlay that should receive the packet. Second, the overlay forwarder finds that host's address in the underlay. Third, it modifies the packet in an overlay-specific way and, fourth, sends it to the next host. In the process of forwarding the packet, the overlay forwarder accesses some internal state, typically a forwarding table. As noted before, the overlay forwarder should provide an interface to the overlay daemon for modifying this state since this state is directly affected by routing changes.

The overlay forwarder is an extensible forwarder. That is, the overlay forwarder should call an application-specific extension function, the application forwarder, at some point during the execution of the forwarding algorithm. The exact point in the algorithm where the extension is called is discussed in Subsection 4.3.4. Since the extension may modify the packet, the overlay forwarder should be able to detect changes to the packet data and modify the data-dependent header fields to reflect those changes.

4.3.3 Application

In many peer-to-peer networks, the application can use knowledge of the current network topology to improve performance, scalability, or security. It is easy to disseminate this information in an application that features a traditional, monolithic design. In a decomposed service, however, the application does not have ready access to the state of the network since that state is maintained by the overlay daemon. The overlay daemon should therefore publish changes in the routing and forwarding tables and provide an interface for subscribing to those updates. The interface can be common to all overlay daemons, but the format of the data returned depends on the particular daemon.

void SubscribeUpdates(**void** (∗updatefunction)(**void**∗))

Since the overlay daemon installs the overlay forwarders and the application may extend the overlay forwarder, the overlay daemon should also export an interface for adding extensions to the overlay forwarders. This interface can be standard across all overlay daemons to take the extension which is then used in all of the overlay forwarders associated with the application, for example:

void RegisterExtension(**void** ∗ (∗extension)(**packet_t**, **void**∗))

Note that the concrete type of the extension depends on the local Plug Board implementation, in this case taking the packet and the address of the next hop as in Subsection 4.3.4. This interface allows the application to modify forwarding and makes the application simpler since it does not need to know how to install an extended overlay forwarder directly.

4.3.4 Application Forwarder

Application-specific forwarding extensions, or application forwarders, extend overlay forwarding in any of several ways. First, they may divert a packet to the local application for further processing, implementing behavior similar to Pronto [26]. Second, they may reroute the packet to a host other than the one chosen by the forwarding algorithm. Last, they may modify the packet before it is transmitted to the next host on the forwarding path.

The ability to divert a packet from the data plane to the control plane is necessary for when the application must be notified that a given packet transited the host or for when the application forwarder decides to reroute a packet to the application for local processing. In the first case, the application requires only a copy of the message, in many cases including all of the packet's headers. In the second case the application requires only the payload of the packet, but since it might respond to the sending host, it would probably require some of the header information as well.

The application forwarder should be able to affect forwarding by rerouting a packet to an intermediate hop other than the one the overlay forwarder chose. As discussed in Subsection 4.3.2, the forwarding process has a specific form: the forwarder computes the next hop, translates that value to an underlay address, modifies the packet, and sends it. In order to easily override the forwarding decision made by the overlay forwarder, the application forwarder should be invoked after computing the next hop, but before determining its address in the underlay. The application forwarder can override the forwarding decision by providing a new overlay address. Functions of type **E**, as described in Chapter 3, should therefore take the packet and the overlay forwarder's choice for a destination as inputs and return a new destination, like so:

void ∗ CallExtension(**packet_t** packet, **void** ∗ nexthop)

The application forwarder should use only the address space of the overlay. For example, an application forwarder executing in an IP forwarder would know nothing about the physical networks below IP. However, the address space that it uses should be enough to identify the next hop. For instance, a proxy, which is just an application forwarder executing within an TCP or UDP forwarder, knows IP addresses as well as port numbers since TCP and UDP rely on IP forming part of the address. That is, the port numbers are not global addresses, but local to the host.

Since the application forwarder must make decisions based on the contents of the packet, it must have access to them. In order to provide the application forwarder with as much information as possible and to make it that much more effective, it is granted read access to the entire packet. Specifically, it should see the entire packet, including the headers of the physical layer network. To allow applications such as compression, the application forwarder should also be able to modify the packet. However, it should be restricted to modifying just the payload, that is, anything after the overlay's header.

The application and application forwarder should have a direct communication channel. The functionality required of this interface is by definition application specific, so the only sufficiently general interface which a system may provide is a communication channel for sending blocks of data between the application and the application forwarder. For instance, the application forwarder may export an interface like:

void GetData(**void** ∗ dataout, **int** ∗ length)
void SetData(**void** ∗ data, **int** length).

Even with the existence of this communication channel, application forwarders should be designed with the understanding that they will execute in a separate domain from the application. If the application forwarder needs to cross the boundary to the application's computation domain for every message, the advantage of having the overlay forwarder in a separate domain is lost.

Diverting packets for local processing through the direct channel between the application and application forwarder leads to a more complex application. Specifi-

Component	Function	Parameters
Application		
Overlay Daemon	SubscribeUpdates	UpdateFunction
	RegisterExtension	Extension
	ForwardingError	ErrorType
		Packet
Overlay Forwarder	UpdateEntry	Key
		Parameter
		Value
Application Forwarder	CallExtension	Packet
		NextHop
	GetData	DataOut
		LengthOut
	SetData	DataIn
		Length

Table 4.1: Interfaces exported by components of decomposed services.

cally, the application is now required to do any protocol processing that an incoming message would usually undergo. To avoid this eventuality, the application forwarder is able to divert a packet to the application's network connection. This avoids the classifier, but still executes Plug Board's protocol processing. Note that this requirement entails writing protocol processing code that does not assume the packet is addressed to the receiving host.

Table 4.1 summarizes the interfaces that each component of the decomposed network service imports from Plug Board and the other components. These interfaces can be implemented as simple, local function calls or remote procedure calls, so components can be placed in different computation domains.

Chapter 5

Evaluation

This chapter concludes the evaluation of the proposed decomposition of network services and the Plug Board architecture. It first presents two point-implementations of Plug Board-conforming systems, each supporting multiple computation domains. The first implementation leverages the SILK [9] variant of Scout to implement a Plug Board that supports extensible IP forwarding and decomposed services in user space and kernel space. The second implementation uses the VERA [29] architecture to implement extensible IP forwarding in the Intel IXP1200 Ethernet Evaluation Board. The chapter then describes and quantifies the benefits that a decomposed overlay-network application can expect in different scenarios.

5.1 Plug Board on a PC

This section describes an implementation of Plug Board on a PC-class computer. It starts by describing the implementation in general and concludes by detailing two example applications using the Plug Board. The first example highlights Plug Board's extensible forwarders while the second highlights Plug Board's support for multiple computation domains.

5.1.1 General Architecture

Plug Board defines three components: forwarders, a classifier, and schedulers; the PC-based implementation supports all three. It also supports installing forwarders

at run time, associating them with predicates, and encapsulating them in input and output functions. The PC-based Plug Board implementation leverages the facilities made available by Scout, the software-based extensible-router architecture described in Chapter 2.

In this Plug Board implementation, forwarders are based on Scout paths. Forwarders are installed directly by name, which causes the Plug Board to create the associated Scout path. The PC-based Plug Board has a programmable, hierarchical classifier that chooses forwarders based on programmer-defined criteria instead of a completely general predicate evaluator. Predicates, or more accurately classification functions, are associated with the forwarders implicitly. As part of path creation, each path is stored in at least one of the leaves of hierarchical classifier, associating the path (forwarder) with a classification function (predicate). This Plug Board also has a configurable output scheduler associated with each port that is specified at configuration time.

Plug Board's forwarder encapsulation leverages two Scout mechanisms. First, Scout paths can be specified as concatenations of arbitrary stages that implement compatible interfaces. To encapsulate a forwarder, the names of the desired input and output functions are added to the forwarder's named template. As a result, the stages that implement those functions are added to the ends of the path implementing the forwarder. Second, for forwarders implemented as single stages that are encapsulated in pairs of input and output functions from the same module, the Plug Board creates a single stage per pair of input and output functions. The stage implementing the outermost encapsulating protocol is the first stage in the path. It is followed by the other stages implementing progressively more deeply nested encapsulating protocols. The stage implementing the forwarder is at the end of the path. Since Scout paths are bidirectional, the encapsulated forwarder can forward the packet by sending it on the same path on which it arrived in the opposite direction.

The Plug Board implementation extends the existing Scout implementation in two important ways. First, even though forwarders are based on Scout paths, they are not chosen implicitly from a predefined set of possible paths. Instead, forwarders are named explicitly during path creation. The name corresponds to a path template which is used to create the path that implements the forwarder's function. Second,

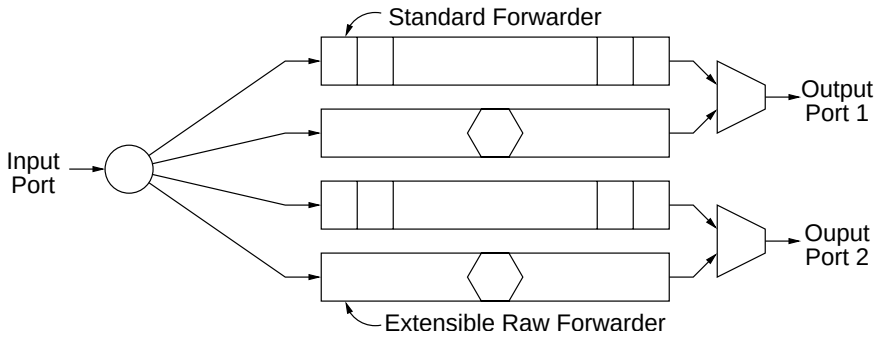


Figure 5.1: IP forwarders in the PC-based Plug Board.

Plug Board provides a configurable packet scheduler associated with each output. Scout provided a scheduler that was fixed at compile time.

5.1.2 Example Application: Extensible IP Forwarding

One of the features of the Plug Board architecture is its ability to support extensible forwarders. A high-performance extensible IP forwarder created for this Plug Board implementation demonstrates that support. The forwarder implements Plug Board's interface for communication between the application and the application forwarder. Additionally, several implemented extensions of the IP forwarding algorithm demonstrate the utility of the extensible forwarder.

The PC-based Plug Board's has separate IP forwarders for each ordered pair of network ports; see Figure 5.1. The forwarders are encapsulated in two sets of input and output functions. The inner encapsulating pair implements the Ethernet protocol but can be replaced by any protocol that supports IP as a networking protocol. The outer pair converts the raw packet data to a Scout Message structure on input and to a Linux socket buffer (`sk_buff`) on output. Processing a packet to be forwarded through these separate input and output steps makes sense architecturally, but negatively affects performance, so this Plug Board also provides an optimized IP forwarder called a "raw" forwarder [46]. The raw IP forwarder operates directly on `sk_buff`s, supports only Ethernet-encapsulated, option-free IP packets, and provides only minimal IP forwarding for those packets. To maintain the high-performance that this forwarder can provide, the optimized forwarder is selected only for those packets that have a cached next-hop address.

Function name	Return type	Parameters

`ProgIP_Install`	`__u32`	`bool (*f)(void * packet, void * data)`
		`__u8 datasize`
		`struct ProgIPPathSpec * demux`

`ProgIP_Remove`	`bool`	`__u32 filter_id`
`ProgIP_SetData`	`void`	`__u32 filter_id` `void * data`
`ProgIP_GetData`	`bool`	`__u32 filter_id` `void * data`

Table 5.1: Interfaces for extending Scout's fast IP forwarding path.

We created an extensible version of the Plug Board's raw IP forwarder. To support the extensible forwarder, the hierarchical classifier was modified to query a table for an extension function after the appropriate raw IP forwarder has been chosen. If an extension function has been registered for the packet's IP addresses and TCP port numbers, it is called before the IP forwarding function. Since the next-hop destination for the packet is already known, this point corresponds the one identified in Subsection 4.3.4 as the one at which extensions should be called. The extension has full read and write access to the packet. Instead of returning a new next-hop address, the extension function returns a boolean value to indicate whether packet processing should continue. Extensions can therefore drop or modify the packets, but not reroute them.

Extensions are assigned regions of reserved memory that they can use to store arbitrary state. Applications can communicate with the extensions using the ProgIP interface—an implementation of Plug Board's interface between applications and application forwarders; see Table 5.1. Applications use the ProgIP interface to read and write a portion of the extension's reserved memory. This interface also allows applications to register extensions to extend any raw IP forwarder as dictated by Subsection 4.3.3.

We have implemented a number of extensions to the Plug Board's raw IP forwarding path. The first extension is a simple timed blocking filter. This filter drops all the packets between a pair of IP addresses and TCP ports. The application installs this filter and removes it after a short time. This trivial extension demonstrates the ability to install and remove extensions dynamically. The next extension,

a SYN monitor with a conditional blocker, demonstrates an extension's use of its stored state. The monitor counts the number of TCP SYN packets that it sees in a period of time. If the number is greater than a certain threshold, the extension will drop any further TCP packets containing SYNs. This extension can act as a very simple firewall to disrupt denial of service (DoS) attacks against a particular host. The last implemented example is a TCP splicer—an extension that splices two TCP connections as described in Chapter 1. For each of these extensions, a test program, the application, installs, removes, and modifies the operation of the extension.

5.1.3 Example Application: Pastry

Plug Board serves as a platform for executing decomposed overlay networks. A decomposed version of the Pastry peer-to-peer application toolkit demonstrates how the PC-based Plug Board can support forwarders in two computation domains: user space and kernel space. This implementation has three elements:

1. A user-space application that uses the Pastry network,
2. A user-space library that provides applications access to a Pastry network and maintains the host's connection to the Pastry network,
3. A Scout kernel module that implements Pastry forwarding.

The following description begins with the kernel-space components, continues with the user-space library, and concludes with the application.

SILK is a Linux kernel module that provides a method for replacing a Linux kernel's networking stack with a Scout kernel. Since Scout kernels are extensible, this embedding is a way of extending the Linux networking stack. The Pastry module for SILK provides the Pastry input and output functions, the Pastry overlay forwarder for the PC-based Plug Board, and the classification code for identifying Pastry packets. The Pastry output function attaches a header to data messages from the local host creating a packet suitable for transmission and forwarding in a Pastry network. The header, shown in Figure 5.2, includes the version of the format, the message type (to distinguish between control and data messages), and the

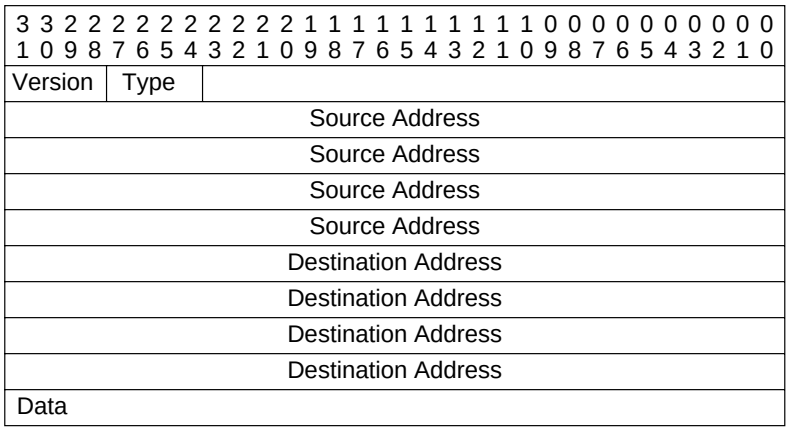


Figure 5.2: The packet format for the decomposed implementation of Pastry

message's source and destination addresses. This packet can then be encapsulated in a transport or other networking protocol for transmission over the underlay. The input function simply strips the header off the packet.

The implementation of the Pastry overlay forwarder is straightforward. The Pastry algorithm chooses the host on the Pastry network whose identifier shares a prefix with the packet's destination identifier that is at least one digit longer than the prefix the local host shares with it. In implementation, the Pastry forwarder finds the length of prefix shared by the local host and the destination identifier. The forwarder then uses that length and the value of the next digit in the destination identifier as indexes into a table that contains Pastry identifiers. If an extension function was configured, the forwarder then calls it with the value of this identifier. When the extension function returns, the forwarder translates the returned Pastry identifier into the address of a host on the underlay and sends the message to that host.

The Pastry overlay forwarder is controlled by the Pastry overlay daemon, which is implemented as a user-level library. The daemon uses SILK sockets [62] to communicate with the kernel-level Pastry components. SILK sockets are sockets of the `PF_SILK` protocol family that, when opened, create paths in the Scout Linux kernel module. In the PC-based Plug Board, opening these sockets installs forwarders.

When an application activates the Pastry user-level library, the library installs the overlay daemon into the Plug Board by opening a Pastry-specific SILK socket.

Computation Domain (Language)	Forwarding Rate (packets/s)
User space (Java)	3 000
User space (C)	18 000
Kernel space (C)	25 000
Network Processor (IXP)	90 000

Table 5.2: Pastry forwarding performance.

The socket creation causes the configured SILK kernel to create a path containing the Pastry input and output functions. The overlay daemon uses this socket to send and receive control messages from the Pastry network. The overlay daemon then installs the Pastry overlay forwarder by creating a second Pastry socket. The overlay daemon uses the `ioctl` interface of the second socket to modify the overlay forwarder's forwarding table, leaf set (a set of nearby nodes), and address translation table.

To create a connection to the Pastry network, applications invoke the Pastry user-level library's `Pastry_Init` function. As a result of the initialization call, the application receives a Pastry socket to use to communicate to other hosts in the Pastry network. The socket implements a standard Unix `recv` and `sendto` interface and encapsulates the data for transmission to Pastry hosts. If the application needs network state information, it can register a callback function with the overlay daemon to be invoked when the leaf set at the local node changes.

The advantage of having the Pastry forwarder in the kernel is the time gained by avoiding a kernel-user crossing to bring a message to be forwarded up to user space. In a SILK-based system passing a 100 B packet to user space and back from the kernel can cost as much as 15 µs [53]. In testing using a PC based on a Pentium-II processor operating at 450 MHz with a 100 Mbit/s Ethernet adapter, it was found that a kernel-level overlay forwarder could forward about 25 000 Pastry messages per second, spending 40 µs on each message. Just placing the overlay forwarder in the kernel and removing the kernel-user crossing allows us to decrease forwarding time by as much as 27%.

5.2 Plug Board on a Network Processor

Software-based routers have historically been built from PC-class machines with conventional network interface cards (NICs) [32, 46]. However, the emergence of network processors (NPs) [27, 28, 61] makes it possible to improve the performance of software-based routers significantly at a modest increase in cost. Spalink *et al.* [55] built a software-based router using the Intel IXP1200 Ethernet Evaluation Board, a network interface card with an integrated StrongARM core and six network processors called microengines. This router can forward minimum-sized, option-free IP packets faster than the Ethernet ports on the IXP1200EEB can supply them, leaving enough resources for extending the IP forwarding path. It can execute extensions that use up to 240 cycles, up to 24 memory accesses (each of 4 B), and up to 10 registers per 64 B segment of a processed packet. Since this router has the resources to permit extension, the network-processor-based Plug Board is based on it. This Plug Board supports three computation domains, the host processor, the StrongARM processor on the IXP1200, and the microengines. The following discussion focuses on the microengine computation domain.

The NP-based Plug Board supports a single forwarder with many extensions in the microengine computation domain. The base forwarder takes packets from the input ports and places them on a queue for transmission to the IXP1200's StrongARM processor. The primary extension to this forwarder is IP forwarding. The IP forwarding extension reroutes packets from the StrongARM's queue to the queues associated with the output ports. In a slight variation of the standard Plug Board architecture, the NP-based Plug Board supports adding extensions to execute before the existing extensions of the forwarder. It also supports creating a logical copy of the extended forwarder to which a single extension can be added. Due to resource limitations, this extended copy cannot be copied further. The classifier chooses extended copies based on a packet's IP addresses and TCP port numbers. When a packet arrives at the host, the classifier selects an extended forwarder for it. If there are no extended copies bound to the IP addresses and port numbers of the packet, the classifier chooses the default extended forwarder. If there is an extension bound to the packet's flow, that extension executes before the others. After the last

extension terminates, the Plug Board places the packet on the queue indicated in the result register.

Extensions to the base forwarder are code fragments written in the assembly language of the IXP1200's packet processing engines, the microengines. Each extension has available to it the 64 B chunk of the packet that the microengine is processing, a pointer to the extension's reserved memory, and several registers containing the microengine's state and forwarding decision. An extension can modify the entire contents of the available chunk. It can also reroute the packet to any of the IXP's output ports or divert the packet to the IXP's StrongARM core.

The Plug Board forwarder in the microengine computation domain allows the user to add extensions dynamically. The IXP compilation pipeline described by Karlin is not sufficient for compiling IXP assembly suitable for dynamic loading. The assembler for the IXP does not allow the user to assign some registers to named variables manually and others automatically. Also, the IXP assembler provides no method for learning the register assignments in the code that would call the extension. Thus, it is not possible to create a register-passing convention for extensions unless the base code and all extensions use manual register assignment. Manually assigning registers to variables is a demanding and error-prone task that is best avoided.

To allow automatic register allocation with the ability to reserve some registers, the compilation pipeline was extended by adding an additional register allocation stage that operates between the preprocessor and the assembler. This allocator keeps track of the values that are live at the point at which the forwarder calls extensions. It then uses a simple, greedy graph coloring algorithm to assign physical registers to the values. These registers are then reserved during the compilation of extensions, and only the free registers are assigned for their execution.

We have created a command line tool to install extensions into the NP-based Plug Board from the host processor. This tool communicates with a simple, special-purpose operating system executing on the IXP1200's StrongARM core. At system initialization, the forwarder can only send packets to the host processor. Using the command line tool, the user can install the IP forwarder and other extensions to the IP forwarding function. Applications can communicate with extensions they install using an interface nearly identical to the ProgIP interface described earlier.

All communication between the host processor and the IXP1200 is implemented using the VERA device driver.

The advantage of the network-processor computation domain is that entering it does not require crossing the PCI bus to the main processor on the PC. Karlin [29] has measured the cost of crossing the PCI bus to and from the Intel IXP1200 Ethernet Evaluation Board. For a 100 B message, the total time to cross the PCI bus in both directions is roughly 5 µs. Forwarding messages at the network card allows the local node to forward at least 10 000 more packets each second. Our implementation of the Pastry overlay forwarder for the IXP1200 (Appendix A) fits within the cycle and memory budgets for processing at maximum line speeds. The expected forwarding rate of the Pastry overlay forwarder in this context is therefore 90 000 Pastry messages per second as opposed to 25 000 through the host processor. The additional gains above the minimally expected 10 000 are due to eliminating the costs of handling the interrupt and finding the proper forwarder at the host processor.

To prove that the cycle and memory budgets provided in the microengine computation domain are sufficient for running a broad class of extensions, the extensions described in the previous subsection were ported to operate in the NP-based Plug Board. Additionally, minimal IP forwarding (denoted IP--) and the forwarding components of Chord and Pastry were also implemented. Table 5.3 lists the computational resources required by the forwarders and extensions ported to the NP-based Plug Board. It shows the number of bytes the forwarder accesses, the number of operations it performs (including pipeline bubbles) and the number of free registers it requires.

The forwarders and the extensions fit well within the cycle budget. IP-- forwards IP packets without options whose next-hop address is cached in the forwarding table. It requires only 43 cycles to recompute the TTL and the checksum and to locate the forwarding table entry for the packet. It accesses 24 B of data of 14 B for the new Ethernet addresses, 8 B to verify the cache entry, and 4 B for the address of the output queue.

The Chord forwarder implements the recursive form of the Chord forwarding algorithm for Chord packets encapsulated in IP packets. Chord forwarding takes fewer than 135 cycles and accesses 56 B of memory. The relatively large memory

	SRAM Read/Write (bytes)	Register Operations	Registers Needed
IP--	24	43	2
Chord	56	134	5
Pastry	20	50	3
TCP Splicer	24	49	7
Wavelet Dropper	4	34	5
ACK Monitor	24	26	6
SYN Monitor	8	13	3
Port Filter	20	51	5

Table 5.3: Memory use, instruction counts, and registers required for sample forwarders and extensions for the NP-based Plug Board. Instruction counts include exposed branch latencies.

requirements and large number of operations are due to Chord's use of a 128-bit address space. Since there are not enough free registers to hold the value during the rest of the computation, the host's own address must be loaded from memory twice, accounting for 32 B of memory use. Any single arithmetic operation, such as addition and subtraction, on each address takes four operations in hardware, contributing to the higher operation count.

The Pastry forwarder implements the Pastry forwarding algorithm for Pastry packets encapsulated in IP packets. The forwarder as implemented is for a Pastry network with $b = 8$, but can be easily extended to any b that is a power of 2. The forwarder uses an exclusive OR to find the first byte in which the destination identifier and the local host's address differ. The location and value of this byte are then used to look up the address of the Pastry host that should receive the packet next. Pastry forwarding requires only 50 cycles, mostly for finding the first byte in which the 128-bit addresses differ.

Both the Chord and Pastry forwarders place the IP address of the next host to receive the packet in the header of the encapsulating IP packet. The modified packet is then processed by the IP-- forwarder, which forwards the packet to the proper host. The instruction counts for the IP-- forwarder is not included in the individual counts for Chord and Pastry. Never the less, even the sum of the computational

requirements of the Chord and IP forwarders is still less than budget imposed by processing packets at line speeds.

The extensions also fit within the cycle budget; e.g. splicing TCP connections takes fewer than 50 cycles while filtering access to any port in any one of five ranges takes just 51 cycles. Appendix A contains additional details about the implementation of the forwarders and extensions. These examples demonstrate that it is feasible to push extended forwarders to computation domains with limited resources.

5.3 Scenarios

This section examines the benefits made available by decomposing a network service and placing the components in different computation domains. It considers four scenarios for the placement of a network service. It describes each scenario briefly, describes the benefits that the particular component placement provides, and comments on the implementation of the interfaces between the components.

5.3.1 User Space and Kernel Space

A standard PC executing a modern operating system provides two computation domains: user space and kernel space. Packets enter the kernel directly from the network and may enter user space only by transiting through the kernel. In this scenario, the overlay daemon and the application reside in user space while the extended, application-specific, overlay forwarder can be loaded into the kernel.

This placement removes two domain crossings from the forwarding path of the packets: to user space from kernel space and back to kernel space from user space. The cost associated with these crossings, denoted t_u, includes the time necessary to copy the packet to user space, the time needed schedule the user space process to run, the time until the process is actually run, and the time it takes the process to call back into kernel space. Denote the time to forward the packet through user space by T. Placing the overlay forwarder in kernel space reduces the forwarding latency by t_u and increases forwarding capacity by $\frac{T-t_u}{T}$ percent. Further, the placement frees $C \times t_u$ processor cycles for the user where C is the processor's clock speed.

Communication within a computation domain can be realized via direct function calls. Therefore, the application and the overlay daemon call each other directly, and the overlay forwarder calls the application forwarder directly. The kernel must provide a mechanism to send to communicate across the kernel-user boundary. Using this mechanism the two component pairs, the general components and the application-specific ones, can implement the communication protocols outlined in Chapter 4.

5.3.2 User Space, Kernel Space, and a Network Processor

Consider a PC whose network interface card contains a network processor. Packets must transit the processor on the NIC to enter the kernel from the network. In this scenario, the overlay and application forwarders can execute in the network processor.

The placement removes the cost of crossing into the host processor from the forwarding path of the packet. These costs, denoted t_h, include the time to copy messages across the peripheral bus twice and the time the host requires to handle the incoming data. Placing the overlay forwarder in the network processor reduces forwarding latency further by t_h. Denote the host processor's clock speed by C and that of the network processor by C_n. Assuming the architectures of the two processors can be accurately compared by processor speed, this placement increase forwarding capacity by

$$\frac{T - t_u - t_h}{T} \frac{C_n}{C}.$$

Additionally, the host processor now has $C \times T$ free cycles.

The interfaces between the component pairs in the previous subsection are the same in this scenario. However, the implementation now requires that the data be sent over the relatively slow peripheral bus to the network processor. If the kernel supports communicating with the network processor directly, the interfaces can be implemented by the control components. Otherwise the kernel must export the same interfaces as in the previous scenario. The first choice means fewer changes in the overlay daemon, the second choice provides for a more efficient implementation.

5.3.3 User's Desktop and an Overlay Server

An overlay server is a computer that provides computation time and memory to execute an overlay forwarder and associated application forwarders for users on the same local-area network. It provides an additional computation domain in which to place the overlay and application forwarders while the overlay daemon and application remain on the user's desktop PC. Consolidating all the LAN's forwarders in one reduces network traffic on the LAN and increases forwarding performance. Leaving the application on the user's desktop grants additional flexibility not available in centrally managed software.

It is expected that an overlay server would possess a more powerful processor and have better connectivity to the outside network than any of the users' desktop computers as it is a managed service not unlike a mail or web server. Denote the difference in network latencies from the closest common router between the overlay server and a user's desktop by Δt. If the overlay server's processor speed is denoted by C_s, the forwarding capacity increases by

$$\frac{T - \Delta t}{T} \frac{C_s}{C}.$$

In cases where forwarding capacity is bounded by the capacity of the network, placing the overlay forwarder at the overlay server may increase forwarding capacity by the difference of the capacities of the networks of the overlay server and user's desktop. The user's desktop has $C \times T$ free cycles, and the network sees fewer forwarded packets.

The communication interfaces among the component pairs are identical to those in the previous scenario. However, since the components now reside on separate machines, the interface would be implemented in a communication protocol like RPC [59].

An alternative placement for the overlay daemon is at the overlay server. With this placement overlay-specific traffic between overlay daemons would remain confined to the overlay server, further reducing the traffic on the LAN. Communication between the overlay daemons and the overlay forwarders, such as updates to the forwarding tables, also remains within the overlay server. Only packets destined for the

application and communication between the overlay daemon and the application, such changes to network topology, would be propagated to the user's desktop.

This latter placement can be viewed as deriving from migration from the server to the user's desktop. Consider a large, public overlay network where users execute applications. The overlay server would be part of this network, e.g. a PlanetLab node, executing the overlay service. Decomposing the service allows offloading the application to the user's desktop thereby reducing the resources required on the overlay server to support the service.

The crucial criterion is the identity of the scarce resource, network capacity or computation at the server. In a department setting, it may be more important to conserve the former. From the perspective of operating a network of overlay server, the latter is more precious.

5.3.4 User's Desktop and an ISP's Overlay Server

Similar to the way single sites may offer an overlay server, ISPs may also offer overlay servers to their clients. The ISP would likely not provide much additional storage and execution time, so only the overlay forwarder, not the daemon, could exist at the ISP's overlay server. The interfaces between the components are identical to those in previous scenario—even the implementation are similar.

Placing an overlay forwarder at the ISP reduces forwarding latency and increase forwarding capacity. Denote the latency of the link between the ISP and its clients as t_c. Since forwarded traffic avoids the client's site entirely, the forwarding latency is reduced by $2t_c$. As before, a powerful overlay server increases forwarding capacity. However, ISPs have significantly better connectivity than their clients. For example, a client site with an internal 100 Mbit/s network may be connected to its ISP via a T1 link with a capacity of 45 Mbit/s. If the overlay forwarder capacity was limited by the network, executing the forwarder at the ISP can increase forwarding capacity at low cost.

Two factors make this placement inappropriate for applications that require the use of application forwarders. First, the ISP may be less willing than the user's own network administrators to operate an application forwarder. The ISP may require more rigorous proof of a forwarder's safety, or they may be unable to offer the mem-

ory and computation resource required for particular application forwarders. Second, communication between the application forwarder at the ISP's overlay server and the application at the user's desktop nearly negates the advantages of executing the overlay forwarder at the ISP. However, for applications that do not involve executing an application forwarder, like those based on Chord, the ISP's overlay server would be effective.

Chapter 6

Conclusions

This dissertation shows how to create flexible and extensible network services. It presents a terminology for describing routers in general and uses this terminology to describe an architecture, Plug Board, that explicitly supports extensible network services. It then shows how to decompose overlay services to best leverage the extensibility of the Plug Board architecture, and finally describes and quantifies the benefits available to a decomposed service.

6.1 Research Contribution

The main contribution of this dissertation is the identification of the underlying similarities common to the great majority of network services and overlay networks. To demonstrate that these similarities exist and are meaningful, the dissertation (1) describes a simple architecture, Plug Board, that supports network services in multiple computation domains, (2) shows how to decompose network services, specifically overlay networks, to map onto this architecture, (3) demonstrates two implementations of this architecture, and (4) describes the benefits made available by leveraging the decomposition.

The Plug Board architecture is a synthesis of previous work in extensible router architectures that supports extensible network services. Plug Board is an architecture that permits more than one implementation and is therefore somewhat abstract. The contribution of this architecture is to make clear the nature of components in

an extensible router and the relationships between them. In Plug Board, a powerful classifier chooses a forwarder that operates on a packet and sends it to the classifier associated with an output port. Forwarders can be fixed, extensible, or encapsulated. These simple components are enough to support both existing and emergent network services.

Decomposing an overlay network into component parts makes it more flexible, easier to extend, and more amenable to optimization. This dissertation provides guidelines for decomposing an overlay network into four components: an overlay daemon to manage the overlay network, an application to offer the overlay's service, an extensible overlay forwarder to forward data through the overlay, and an application-specific forwarding extension to tailor forwarding to the needs of the application. Decomposing an application that uses an overlay network allows it to use different overlays without major retooling. It allows the overlay and the application to evolve independently. Placing individual components in separate computation domains introduces optimizations by lowering the costs of processing packets that are forwarded through the host.

Implementing Plug Board in two different hardware architectures, a PC-class computer and a network processor, demonstrates the trade-offs inherent in choosing a computation domain for a forwarder. On the PC-based Plug Board, the additional performance gained by executing the forwarder in the kernel requires trusting the forwarder to be safe. The additional benefits gained by executing the forwarder in the network-processor-based Plug Board are balanced by requiring that the forwarder be implemented in the assembly language of the network processor and be small enough to fit a tight cycle budget. Several examples demonstrate that the cycle budget available in the NP-based Plug Board is sufficient to allow non-trivial forwarding at line speeds.

Placing components of an overlay network in physically separate computation domains introduces many more potential optimizations. While this dissertation does not describe and implement an example of this placement, it does demonstrate the necessary interfaces that support such a placement. Further, it quantifies the benefits made available and some of the limitations of placing components of an overlay network in separate computation domains.

6.2 Future Work

There are several directions in which the work presented here can be expanded. First, the implementation of a Plug Board that supports many different computation domains could be used to further validate decomposing networking services as a method of improving efficiency. Specifically, the Plug Board implementations herein described can be merged into a single Plug Board that supports extension in three computation domains.

Second, Plug Board's interfaces can be further refined. Plug Board's interface between an application and its forwarding extension is, by necessity, very general—it provides for a simple bit pipe between the two. Cataloging a sufficiently useful subset of interfaces would make decomposition guidelines more concrete which would make decomposing services even easier. Plug Board's interfaces for installing and encapsulating packets could also be extended to provide some language-level guarantees that each input and output function has all the necessary data to complete it operation. For example, the Ethernet output function must be supplied with the correct destination address, the current interface does not enforce this requirement.

Finally, the classifier in the Plug Board implementations was implemented either as a programmable hierarchical classifier or as a simple filter on IP addresses and port numbers. Implementing an efficient, general-purpose predicate resolver presents challenges in devising efficient algorithms. Merging the advances made by packet classifiers into a general predicate resolver would allow greater flexibility in choosing packets, creating the opportunity for more finely defined network extensions.

Appendix A

Code Listings

This appendix contains annotated code listings for the IXP1200 implementation of example forwarders and extensions. All of the code fragments below use the packet format shown in Figure A.1.

A.1 IP-- Forwarding

The following code fragment is an implementation of simple IP forwarding. This forwarding does not operate in the presence of options and modifies the link-layer header only if a valid value can be found in the forwarder's cache. Each cache line contains a pair of Ethernet addresses, a pair of IP addresses and a port number, as shown in Figure A.2.

```
#include "libStd.uc"
#include "libVrp.uc"

;

;;; ipmm_match
;;; IN: in_cache_ip2 --- The ip2 value in the cache
;;; IN: in_cache_d07 --- The d07 value in the cache
;;; IN: in_d06 --- The d06 value in the packet
;;; IN: in_d07 --- The d07 value in the packet
;;; IN: in_d08 --- The d08 value in the packet
;;; Sets the condition code to 0 if the items in the packet match the
;;; items in the cache.
```

	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	09	08	07	06	05	04	03	02	01	00
D_{00}	$ETHDst_0$								$ETHDst_1$								$ETHDst_2$								$ETHDst_3$							
D_{01}	$ETHDst_4$								$ETHDst_5$								$ETHSrc_0$								$ETHSrc_1$							
D_{02}	$ETHSrc_2$								$ETHSrc_3$								$ETHSrc_4$								$ETHSrc_5$							
D_{03}	$ETHType_0$								$ETHType_1$								Version				Hdr Len				TOS							
D_{04}	$Length_0$								$Length_1$								ID_0								ID_1							
D_{05}	Flags			$Offset_0$					$Offset_1$								TTL								Protocol							
D_{06}	$Checksum_0$								$Checksum_1$								$IPSrc_0$								$IPSrc_1$							
D_{07}	$IPSrc_2$								$IPSrc_3$								$IPDst_0$								$IPDst_1$							
D_{08}	$IPDst_2$								$IPDst_3$								$TCPSrc_0$								$TCPSrc_1$							
D_{09}	$TCPDst_0$								$TCPDst_1$								Seq_0								Seq_1							
D_{10}	Seq_2								Seq_3								Ack_0								Ack_1							
D_{11}	Ack_2								Ack_3								Hdr Len				0				0	U	A	P	R	S	F	
D_{12}	Adver Win_0								Adver Win_1								$Checksum_0$								$Checksum_1$							
D_{13}	$UrgPtr_0$								$UrgPtr_1$																							

Figure A.1: Packet format for IXP1200 forwarders and extensions.

	31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 09 08 07 06 05 04 03 02 01 00
0	Ethernet Source
1	Ethernet Source | Ethernet Destination
2	Ethernet Destination
3	IP Source (high bits) | IP Destination (low bits)
4	IP Source (low bits) | IP Destination (high bits)
5	Port | Reserved
6	Reserved
7	Reserved

Figure A.2: Cache line layout for the IP-- forwarder.

```
#macro ipmm_match [in_cache_ip2, in_cache_d07, in_d06, in_d07, in_d08]
  .local match temp
  ; Get the values to cache format
    combine_from_2[temp, in_d06, in_d08]
    alu [temp, temp, XOR, in_cache_ip2] ; Compare 1
    sa_debug_msg [temp]
    alu [match, in_cache_d07, XOR, in_d07] ; Compare 2
    sa_debug_msg [match]
    ; Set the condition code if both values matched
    alu [--, match, OR, temp]
  .endlocal ; temp match
#endm

;;; debug_xfer
;;; IN: in_num --- The number of transfer registers.
;;; Prints out the contents of in_num sequentially numbered transfer
;;; registers.
#macro debug_xfer [in_num]
  #define_eval count 0
  #while count < in_num
    sa_debug_msg [$xfer/**/count]
    #define_eval count count + 1
  #endloop
  #undef count
#endm

;

;;; ipmm_dec_ttl
;;; IN/OUT: io_ttl --- Long word 5 in the IP packet, contains TTL
;;; IN/OUT: io_cksum --- Long word 6 in the IP Packet, contain checksum
;;; Decrement the TTL in the given register, assuming the register is long word
;;; 5 of the packet.
#macro ipmm_dec_ttl [io_ttl, io_cksum]
  ; Decrement TTL in place
  alu_shf [io_ttl, io_ttl, -, 1, <<IP_TTL_OFFSET]
  .local lsum
    ntohl [lsum, io_cksum] ; Move the little endian

    alu_shf [lsum, lsum, +, 1, <<24]
    .if cout() ; If the cksum overflowed, Only for big endian
      alu_shf [lsum, lsum, +, 1, <<16]; Add the overflow.
    .endif
    ntohl [io_cksum, lsum]
  .endlocal
#endm

;;; ipmm_link_layer
;;; OUT: out_d0 --- Longword 0 of the packet
```

```
;;; OUT: out_d1 --- Longword 1 of the packet
;;; OUT: out_d2 --- Longword 2 of the packet
;;; IN: in_ethsrc0 --- The new contents of longword 0
;;; IN: in_ethsrc1 --- The new contents of longword 1
;;; IN: in_ethsrc2 --- The new contents of longword 2
;;; Rewrite the link layer header of the IP packet from the contents of
;;; in_ethsrc[012].
#macro ipmm_link_layer [out_d0, out_d1, out_d2, in_ethsrc0, in_ethsrc1,
    in_ethsrc2]

  alu [out_d0, --, B, in_ethsrc0]
  alu [out_d1, --, B, in_ethsrc1]
  alu [out_d2, --, B, in_ethsrc2]
#endm

;;;
;;; This is the IP-- function which will get loaded into the VRP
;;; See vrp.uc for most of the parameters.
;;; IN: IPhashValue --- The hashed value of the IP source and
;;; destination addresses.
;;; IN: funcAndData --- Contains the base address of the route cache.
;;;
#macro ipmm [vrpResult, srcPid, pfState, d00, d01, d02, d03, d04, d05, d06, d07
    , d08, d09, d10, d11, d12, d13, d14, d15, IPhashValue, funcAndData]
  sa_debug_msg [0x0000EEEA]

  ; Only forward for the first part of the packet.
  br_bclr [pfState, 0, end_ipmm#], defer[3]

  .local iptype
  ;; ETHType(IP) = 0x0800, IPv4, Hdrlen = 5 (long words == 20 bytes)
    #if BIG_ENDIAN
    set [iptype, 0x08004500]
    #else
    set [iptype, 0x450008]
    #endif
    alu [--, iptype, XOR, d03]
  .endlocal ; iptype

  ;; Condition code 0 set IFF the packet is an IPv4 packet with no
  ;; options.
  br!=0 [end_ipmm#]

  ;;
  ;; Look up the address in the route cache.
  .local $xfer0 $xfer1 $xfer2 $xfer3 $xfer4 $xfer5
    .xfer_order $xfer0 $xfer1 $xfer2 $xfer3 $xfer4 $xfer5
    .local base offset
      sa_debug_msg [IPhashValue]
```

```
      ; Convert the hash value into an offset into the table.
      ld_field_w_clr [offset, 0011, IPhashValue, <<3]
      alu_shf [offset, offset, AND~, 1, <<15]
      ; Get the base address
      get_field [funcAndData, 10, 19, base]
      ; Read the cache entry from SRAM[base + offset].
      sram [read, $xfer0, base, offset, 6], ctx_swap
    .endlocal ; base offset

    debug_xfer [6]

    ; Make sure the cache entry matches
    ipmm_match [$xfer3, $xfer4, d06, d07, d08]

    br!=0 [end_ipmm#]
    ld_field [vrpResult, 1000, $xfer5] ; Load the port

    ipmm_dec_ttl [d05, d06] ; Decrement the TTL

    ; Copy the link layer
    ipmm_link_layer [d00, d01, d02, $xfer0, $xfer1, $xfer2]

    ; Drop the packet if the TTL==0
    br!=byte [d05, IP_TTL_BYTE, 0, end_ipmm#]

  .endlocal ; $xfer0 $xfer1 $xfer2 $xfer3 $xfer4 $xfer5

ipmm_drop#:
  vrp_packet_drop [vrpResult]

end_ipmm#:
  sa_debug_msg [vrpResult]
  sa_debug_msg [0xAEEE0000]
  nop
#endm
```

A.2 Chord

Nodes on the Chord network maintain forwarding information in a table called the finger table. Table A.1 defines the names and values for variables in the finger table. For every key in a Chord network, the key's successor node is responsible for it, so packets in a Chord network are forwarded to the node that is the successor of the identifier in the packet. Figure A.3, reproduced from [58], shows the iterative Chord lookup algorithm. The recursive form of the algorithm forwards the packet every

Notation	Definition
$finger[k].start$	$(n + 2^{k-1}) \bmod 2^m$ for $1 \leq k \leq m$
$finger[k].interval$	$[finger[k].start, finger[k+1].start)$
$finger[k].node$	first node $\geq n.finger[k].start$
successor	$finger[1].node$
predecessor	the previous node on the identifier circle

Table A.1: Definitions of variables for node n, using m-bit identifiers.

time information is required from a node other than n, that is after each iteration in *find_predecessor*.

The Chord forwarding algorithm is not suitable for implementation within the cycle budget of the NP-based Plug Board because of the large number of 128-bit comparisons. Specifically, in the worst case the function *closest_preceding_finger* requires 128 interval comparisons. Since the IXP1200's registers are 32 bits wide, each interval comparison requires four simple comparisons in hardware. Together, these facts imply that the algorithm potentially requires 1024 comparisons, which is beyond the budget of 240 instructions. This section presents an alternative formulation of the algorithm and its implementation in IXP1200 assembly.

A.2.1 Algorithm

To properly formulate an alternative algorithm for Chord forwarding it is necessary to examine the steps taken by a node executing the recursive form of Chord forwarding. In the recursive form, any reference to a variable stored in another node causes the packet to be forwarded to that node. The algorithm executed at each node is:

Require: The local node n, and requested identifier n_r

1: **if** $n_r \in (n, n.successor]$ **then**
2: forward to $n.successor$
3: **else**
4: $n' \leftarrow n$
5: **for** $i \leftarrow 1$ to m **do**
6: **if** $finger[i].node \in (n, n_r)$ **then**

```
//ask node n to find id's successor
n.find_successor(id)
  n' = find_predecessor(id);
  return n'.successor;

//ask node n to find id's predecessor
n.find_predecessor(id)
  n' = n
  while (id ∉ (n', n'.successor])
    n' = n'.closest_preceding_finger(id);
  return n';

//return closest finger preceding id
n.closest_preceding_finger(id)
  for i = m downto 1
    if finger[i].node ∈ (n, id)
      return finger[i].node;
  return n;
```

Figure A.3: The pseudocode to find the successor node of an identifier *id*. Remote procedure calls and variable lookups are preceded by the remote node.

```
7:        n' ← finger[i].node
8:    forward to n'
```

Note that the direction of the loop in line 5 has been reversed from the original to simplify the algorithm's presentation.

In the algorithm, there are at most 129 interval comparisons, one in line 1 and up to 128 in the loop on line 5. Since all these comparisons have the local node as the start of the interval, the interval comparisons can be converted to simple comparisons by subtracting the local node's identifier from the identifier in the packet and from all the identifiers in the finger table. Renormalizing the identifiers in this way reduces the number of comparisons by half.

Renormalizing the identifier space has the effect of rotating the local node's view of the identifier circle so that the local node has the identifier 0. This means that the node identifiers that define the start of the intervals in the finger table take the simplified form 2^{k-1}. The binary representation of these node identifiers contain a single one in position k, for $1 \leq k \leq m$. Therefore, finding the interval on which the requested identifier lies reduces to finding the index of the most significant bit

set in the renormalized identifier. If the identifier of the node associated with this interval is less than the requested identifier, that node is the closest preceding finger and should receive the packet next.

Assuming the entire finger table has been appropriately renormalized, the reformulated algorithm takes the following form:

Require: The local node, n, and requested identifier n_r

$n_{r'} \leftarrow n_r - n$
if $n_r < n.successor$ **then**
  forward to $n.successor$
else
  $n' \leftarrow n$
  $k \leftarrow$ index of MSB($n_{r'}$)
  for $i \leftarrow 1$ to k **do**
    if $finger[i].node < n_{r'}$ **then**
      $n' \leftarrow finger[i].node$
  forward to $n' + n$

This algorithm has no interval comparisons, but it still potentially has 129 simple comparisons. The number of comparisons can be reduced further by introducing another entry in the finger table—the *last* pointer. Define $finger[k].last$ as $finger[j].node$ for the largest j such that $j < k$ and that $finger[j].node \neq finger[k].node$. Intuitively, this node is the node in the finger table that directly precedes $finger[k].node$. The value of the last node can be derived strictly from information already in the finger table, so adding it does not change Chord's protocols. To eliminate the last loop in the algorithm note that for $k = \text{MSB}(n_{r'})$, if $finger[k].node \geq n_{r'}$ then $finger[k].last < n_r$ since n_r is on the interval k while $finger[k].last$ is on a preceding interval by definition. The final algorithm becomes:

Require: The local node, n, and requested identifier n_r

$n_{r'} \leftarrow n_r - n$
if $n_{r'} < n.successor$ **then**
  forward to $n.successor$
else

$n' \leftarrow n$
$k \leftarrow$ index of MSB($n_{r'}$)
if $finger[k].node < n_{r'}$ **then**
  $n' \leftarrow finger[k].node$
else
  $n' \leftarrow finger[k].last$
forward to $n' + n$

The final algorithm has only two simple comparisons which should fit within the NP-based Plug Board's cycle budget. The only complication is the complexity of finding the index of the most significant bit. A binary search would find the index in roughly seven steps. Fortunately, the IXP1200 has a hardware instruction specifically for this purpose which can be leveraged to improve performance even further.

A.2.2 Implementation

This section contains an annotated code listing of the implementation of the Chord forwarding algorithm for the IXP1200. The following are assumed

1. The forwarding algorithm will only be applied to packets that are not destined to the local node.

2. The Chord protocol is encapsulated directly by IP and that the addresses are readily available in the beginning of the packet.

3. The finger table contains the current node's identifier and its additive inverse, followed by 128 pairs of IP addresses.

```
; Include files containing useful macros
#include "libRegs.h"
#include "libVrp.uc"
#include "libStd.uc"

; Load the values of the predefined registers
regs_packet
regs_state
```

```
#macro bit_swizzle [out_dst, in_src]
;; This code is from Microcode Programmer's Reference Manual,
;; pg. 305.  It reverses the order of the bits within each byte of
;; the register in_src
  .local temp mask

    immed_w1 [mask, 0x0101]
    immed_w0 [mask, 0x0101]

    alu [temp, in_src, AND, mask]
    alu_shf [out_dst, out_dst, OR, temp, <<rot7]
    alu_shf [temp, in_src, AND, mask, <<rot1]
    alu_shf [out_dst, out_dst, OR, temp, <<rot5]
    alu_shf [temp, in_src, AND, mask, <<rot2]
    alu_shf [out_dst, out_dst, OR, temp, <<rot3]
    alu_shf [temp, in_src, AND, mask, <<rot3]
    alu_shf [out_dst, out_dst, OR, temp, <<rot1]
    alu_shf [temp, in_src, AND, mask, <<rot4]
    alu_shf [out_dst, out_dst, OR, temp, >>rot1]
    alu_shf [temp, in_src, AND, mask, <<rot5]
    alu_shf [out_dst, out_dst, OR, temp, >>rot3]
    alu_shf [temp, in_src, AND, mask, <<rot6]
    alu_shf [out_dst, out_dst, OR, temp, >>rot5]
    alu_shf [temp, in_src, AND, mask, <<rot7]
    alu_shf [out_dst, out_dst, OR, temp, >>rot7]
  .endlocal ; temp mask
#endm

;; The chord forwarding code begins here.
.local $xfer0 $xfer1 $xfer2 $xfer3
  .xfer_order_rd $xfer0 $xfer1 $xfer2 $xfer3

  ; Load the additive inverse of the local node's 128-bit id.
  .local data_addr
    alu_shf [data_addr, --, B, funcAndData, >>10]
    sram [read, $xfer0, data_addr, 4, 4], ctx_swap
  .endlocal

  ;; Assume the packet lists the destination id aligned properly
  ;; in d09, d10, d11, and d12.  Convert to host representation.
  ntohl [d12, d12]
  ntohl [d11, d11]
  ntohl [d10, d10]
  ntohl [d09, d08]

  ;; Renormalize the address in the packet
  alu [d12, d12, +, $xfer3]
  alu [d11, d11, +carry, $xfer2]
```

```
  alu [d10, d10, +carry, $xfer1]
  alu [d09, d09, +carry, $xfer0]

.endlocal ; $xfer0 $xfer1 $xfer2 $xfer3
.local bit_index
;; Find the index of the first bit in the address
  .local difference
  ;; First find the correct 32-bit word
    alu [difference, --, B, d09]
    br!=0 [found_word#], defer[1]
    set [bit_index, 0]

    alu [difference, --, B, d10]
    br!=0 [found_word#], defer[1]
    alu_shf [bit_index, bit_index, +, 1, <<5]

    alu [difference, --, B, d11]
    br!=0 [found_word#], defer[1]
    alu_shf [bit_index, bit_index, +, 1, <<5]

    ;; This word must contain the difference, otherwise the
    ;; packet would not be forwarded by this code
    alu [difference, --, B, d12]
    alu_shf [bit_index, bit_index, +, 1, <<5]

found_word#:
    ;; Now find the bit in the word.  First make the packet
    ;; little endian byte order
    #if BIG_ENDIAN
      byte_swap [difference, difference]
    #endif

    ;; Next put the bits within each byte in little endian order
    .local altered_difference
      bit_swizzle [altered_difference, difference]
      alu [difference, --, B, altered_difference]
    .endlocal

    ;; Use the hardware bit-set finder to find the first bit,
    ;; from the little end that is set.
    find_bset [difference]
    find_bset [difference, >>16]
  .endlocal ; difference
  nop
  nop
  nop
  .local bit_index_lower
    load_bset_result1 [bit_index_lower], clr_results
```

```
    ;; Mask away the valid bit that, by assumption, must be
    ;; true.
    alu [bit_index_lower, bit_index_lower, +8, 0]
    alu [bit_index, bit_index, +, bit_index_lower]
  .endlocal

  ;; Convert the index of the first bit into an offset into
  ;; memory
  alu_shf [bit_index, 8, +, bit_index, <<3]

  ; Load the node ID and the IP addreses
  .local $xfer0 $xfer1 $xfer2 $xfer3 $xfer4 $xfer5
    .xfer_order_rd $xfer0 $xfer1 $xfer2 $xfer3 $xfer4 $xfer5

    .local data_addr
      alu_shf [data_addr, --, B, funcAndData, >>10]
      sram [read, $xfer0, data_addr, bit_index, 6], ctx_swap
    .endlocal

    ;; If finger[k].node < n_r
    alu [--, $xfer0, -, d09]
    br<0 [use_ip_dst#]
    br>0 [use_ip_alt#]

    alu [--, $xfer1, -, d10]
    br<0 [use_ip_dst#]
    br>0 [use_ip_alt#]

    alu [--, $xfer2, -, d11]
    br<0 [use_ip_dst#]
    br>0 [use_ip_alt#]

    alu [--, $xfer3, -, d12]
    br<0 [use_ip_dst#]
    br [use_ip_alt#]

use_ip_dst#:
    ;; ...then use finger[k].node
    replace_ip_address [d07, d08, d06, $xfer4]
    br [chord_done#]
use_ip_alt#:
    ;; ...else use n_r
    replace_ip_address [d07, d08, d06, $xfer5]

  .endlocal ; $xfer0 $xfer1 $xfer2 $xfer3 $xfer4 $xfer5
.endlocal

chord_done#:
;; Undo the rescaling.  Load the local node's id
```

```
.local $xfer0 $xfer1 $xfer2 $xfer3
  .xfer_order_rd $xfer0 $xfer1 $xfer2 $xfer3

  .local data_addr
    alu_shf [data_addr, --, B, funcAndData, >>10]
    sram [read, $xfer0, data_addr, 0, 4], ctx_swap
  .endlocal

  ;; Add the 128-bit values.
  alu [d12, d12, +, $xfer3]
  alu [d11, d11, +carry, $xfer2]
  alu [d10, d10, +carry, $xfer1]
  alu [d09, d09, +carry, $xfer0]

  ntohl [d12, d12]
  ntohl [d11, d11]
  ntohl [d10, d10]
  ntohl [d09, d09]

.endlocal ; $xfer0 $xfer1 $xfer2 $xfer3
```

A.3 Pastry

The forwarder for the Pastry network is easy to implement within the cycle budget of the NP-based Plug Board. The code in the listing makes the following assumptions:

1. On this Pastry network $b = 8$. That is, the Pastry addresses are interpreted as 16-digit numbers in base 256.

2. The code will only be applied to packets that are not destined to the local node. This restriction is enforceable by the classifier via two interval comparisons.

3. The Pastry protocol is encapsulated directly by IP and the addresses are readily available in the beginning of the packet. Unlike the Pastry implementation described before, the destination address appears before the source address.

```
#include "libRegs.h"
#include "libStd.uc"
#include "libVrp.uc"

#macro shift_column_index [io_columnidx]
```

```
#if BIG_ENDIAN
alu_shf [io_columnidx, --, B, io_columnidx, <<8]
#else
alu_shf [io_columnidx, --, B, io_columnidx, >>8]
#endif
#endm

regs_packet
regs_state

.local value
  .local test_register byteCount
    .local $xfer0 $xfer1 $xfer2 $xfer3
      .xfer_order_rd $xfer0 $xfer1 $xfer2 $xfer3

      ; Read the node's ID from memory.
      .local data_addr
        alu_shf [data_addr, --, B, funcAndData, >>10]
        sram [read, $xfer0, data_addr, 0, 4], ctx_swap
      .endlocal

      ;; Assume pastry packet lists destination first aligned
      ;; properly in d09, d10, d11, and d12.
      ;; Find the first bit in which the node's ID differs from
      ;; the packet's ID.

      alu [test_register, d09, XOR, $xfer0]
      br!=0 [found_first_set_bit#], defer[1]
      alu [value, --, B, $xfer0]

      alu [test_register, d10, XOR, $xfer1]
      br!=0 [found_first_set_bit#], defer[2]
      alu [byteCount, byteCount, +, 4]
      alu [value, --, B, $xfer1]

      alu [test_register, d11, XOR, $xfer2]
      br!=0 [found_first_set_bit#], defer[2]
      alu [byteCount, byteCount, +, 4]
      alu [value, --, B, $xfer2]

      alu [test_register, d12, XOR, $xfer3]
      br=0 [pastry_done#], defer[2]
      alu [byteCount, byteCount, +, 4]
      alu [value, --, B, $xfer3]
    .endlocal ; $xfer0--3

found_first_set_bit#:
    ; Find the byte in which this bit is set.
    ld_field_w_clr [--, BYTEZERO, test_register], load_cc
```

```
    br!=0 [found_first_set_byte#]

    ld_field_w_clr [--, BYTEONE, test_register], load_cc
    br!=0 [found_first_set_byte#], defer[2]
    shift_column_index [value]
    alu [byteCount, byteCount, +, 1]

    ld_field_w_clr [--, BYTETWO, test_register], load_cc
    br!=0 [found_first_set_byte#], defer[2]
    shift_column_index [value]
    alu [byteCount, byteCount, +, 1]

    shift_column_index [value]
    alu [byteCount, byteCount, +, 1]

found_first_set_byte#:
    ;; Each entry in the table has only one value:  the
    ;; destination address.
    #if BIG_ENDIAN
      alu_shf [value, 4, +, value, >>24]
    #else
      alu [value, 4, +8, value]
    #endif

    alu_shf [value, value, +, byteCount, <<8]
  .endlocal ; test_register byteCount

  .local $xfer0
    .local data_addr
      alu_shf [data_addr, --, B, funcAndData, >>10]
      sram [read, $xfer0, data_addr, value, 1]
    .endlocal ; data_addr

    replace_ip_address [d07, d08, d06, $xfer0]

  .endlocal ; $xfer0

.endlocal ; value

pastry_done#:
```

A.4 TCP Splicing

Splicing two TCP connections together requires replacing the IP addresses and TCP port numbers of a packet, adjusting the sequence and acknowledgment numbers by

	3 3 2 2 2 2 2 2 2 2 2 2 1 1 1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0
0	Difference of IP checksum / IP source address (high bits)
1	IP source address (low bits) / IP destination address (high bits)
2	IP destination address (low bits) / TCP source port
3	TCP destination port / Difference of TCP checksum
4	Sequence number difference
5	Acknowledgment number difference

Figure A.4: Data format for TCP splicer.

a known difference and recomputing the IP and TCP checksums to match the other changes. The TCP splicer keeps a set of values for this purpose whose format is given in Figure A.4.

```
#include "libRegs.h"
#include "libStd.uc"
#include "libVrp.uc"

;;;

; Recomputes the IP checksum for spliced packets.  Must be called
; before changing the IP addresses in the packets.  Uses incremental
; update and a precomuputed value.

#macro tcp_splice_ipcksum [d06, xfer0]
.local cksum diff
  ; Get the IP Checksum of the packet
  load_upper_half [cksum, d06]

  ; The precomputed difference between the new and old check sums
  load_upper_half [diff, xfer0]

  ; Recompute the checksum
  alu [cksum, --, ~B, cksum]
  alu [cksum, cksum, +, diff]
  alu [cksum, cksum, +carry, 0]

  ; Place the checksum in the packet.  This will overwrite half
  ; the IP address in the packet, but that will be overwritten
  ; anyway, and using alu or alu_shf saves a cycle for the
  ; inversion.
  #if BIG_ENDIAN
    alu_shf [d06, --, ~B, cksum, <<16]
  #else
    alu [d06, --, ~B, cksum]
```

```
  #endif
.endlocal
#endm

; out = in + diff.  Increment in and place the result in out.  In is in host
; byte order, diff is in network byte order.
#macro tcp_splice_ninc [out, in, diff]
.local rev
  ntohl [rev, in]
  alu [rev, rev, +, diff]
  ntohl [out, rev]
.endlocal
#endm

; Increment the ACK and SEQ fields of the TCP header.
#macro tcp_splice_seqack [seqout, seqin, seqdiff, ackout, ackin, ackdiff]
tcp_splice_ninc [seqout, seqin, seqdiff]
tcp_splice_ninc [ackout, ackin, ackdiff]
#endm

; Subtract in_num from inout_cksum using two's complement subtraction
#macro tcp_splice_add_inv [inout_cksum, in_num]
.local inv
  alu [inv, --, ~B, in_num]
  alu [cksum, cksum, +, inv]
  alu [inout_cksum, inout_cksum, +carry, 0]
.endlocal
#endm

; Recompute the TCP checksum
#macro tcp_splice_tcpcksum [d12, seq, newseq, ack, newack, xfer3]
.local cksum

  tcp_splice_add_inv [cksum, seq]
  tcp_splice_add_inv [cksum, ack]

  load_lower_half [cksum, d12] ; Get the original checksum

  .local diff
    load_lower_half [diff, xfer3] ; Get the precomputed diffenence
    ;
    ; Replace the old bits with the new ones.
    ;
    alu [cksum, cksum, +carry, diff]
  .endlocal

  alu [cksum, cksum, +carry, newseq]
  alu [cksum, cksum, +carry, newack]
```

```
  ; Add the upper half of cksum to the lower half.
  .local upper
    ld_field_w_clr [upper, 0011, cksum]
    alu_shf [upper, upper, +carry, cksum, >>16]
    ; We need to get the top bit of upper
    ld_field_w_clr [cksum, 0011, upper]
    alu_shf [cksum, cksum, +16, upper, >>16]
  .endlocal

  store_lower_half [d12, cksum]; Store the checksum where it belongs
.endlocal
#endm

;; The splicing Code begins here
regs_state
regs_packet

.local $xfer0 $xfer1 $xfer2 $xfer3 $xfer4 $xfer5
  .xfer_order $xfer0 $xfer1 $xfer2 $xfer3 $xfer4 $xfer5

  .local addr
    alu_shf [addr, --, B, funcAndData, >>10]
    ;; Defer the first instruction of ipcksum, it doesn't rely on xfer
    sram [read, $xfer0, addr, 0, 6], ctx_swap, defer[1]
  .endlocal

  ;; First Recompute the IP checksum
  tcp_splice_ipcksum [d06, $xfer0]

  ;; Next overwrite the values we need to change
  ld_field [d06, LOWER_HALF, $xfer0]
  ld_field [d07, ALL_BITS, $xfer1]
  ld_field [d08, ALL_BITS, $xfer2]
  ld_field [d09, UPPER_HALF, $xfer3]

  ;; Now the TCP checksum
  .local seq ack newseq newack

    combine_from_2 [seq, d09, d10]
    combine_from_2_same_bank [ack, d10, d11]

    tcp_splice_seqack [newseq, seq, $xfer4, newack, ack, $xfer5]
    tcp_splice_tcpcksum [d12, seq, newseq, ack, newack, $xfer3]

    ; Use temporary register to permit automatic allocation
    .local tmp
      set [tmp, newseq]
      ld_field [d09, UPPER_HALF, tmp, <<rot16]
      ld_field [d10, LOWER_HALF, newseq, <<rot16]
```

```
    .endlocal

    split_across_2 [d10, d11, newack]
  .endlocal

.endlocal
```

A.5 Wavelet Dropper

The following extension implements part of the WaveVideo [20] system for dropping video packets. It transforms the tag in the packet and uses the transformed tag as an index to a table that contains boolean values indicating whether the packet should be dropped. It is assumed that the packets are encapsulated in UDP and that the tag is available immediately after the UDP header. The listing is based on a September 2000 version of the WaveVideo system.

```
#include "libRegs.h"
#include "libVrp.uc"
#include "libStd.uc"

regs_packet
regs_state

;;; Places drop/no drop in the state variable then jumps to return addr.

.local tag tag8bit

  ; Load the value of the tag in the packet
  ;
  load_lower_half[tag, d10]
  store_upper_half[tag, d11]
  ntohl [tag, tag]

  ;; The next 3 instructions execute regardless of the branch.  This
  ;; causes no harm and improves performance.

  ; Branch if no scaling to be done.
  br_bclr [tag, 23, end_wavelet_dropper#], defer[3]
  ;; Otherwise, transform the tag into an 8-bit tag (see Tag.c in
  ;; WaveVideo release.

  ; tag & SEQUENCENUMBER
  ld_field [tag8bit, 0001, tag]
```

```
; tag8bit |= ((tag & IFRAME) >> 16)
.local tmp
  alu_shf [tmp, tag, AND, 1, <<31]
  alu_shf [tag8bit, tag8bit, OR, tmp, >>16]
.endlocal ; tmp


; tag8bit |= ((tag & COLORMASK) << 13);
.local tmp
  alu_shf [tmp, tag, AND, 3, <<8]
  alu_shf [tag8bit, tag, OR, tmp, <<13]
.endlocal ; tmp

; tag8bit |= (((tag & DEPTH) >> 9) - ((tag & SIZE) >> 6));
.local tmp1 tmp2
  alu_shf [tmp1, tag, AND, 7, <<27]
  alu_shf [tmp1, --, B, tmp1, >>9]

  alu_shf [tmp2, tag, AND, 7, <<24]

  alu_shf [tmp2, tmp1, -, tmp2, >>6]
  alu [tag8bit, tag8bit, OR, tmp2]
.endlocal ; tmp1 tmp2

; tag8bit |= ( ( ( (tag >> ( ((tag & DEPTH) >> 27) << 1 ) )
; >> 8)
; & DIRECTION)
; << 16)
.local tmp
  alu_shf [tmp, 0xE, AND, tag, >>26]
  alu [--, tmp, B, 0]
  alu_shf [tmp, --, B, tmp, >>indirect]
  alu_shf [tmp, 0x3, AND, tmp, >>8]
  alu_shf [tag8bit, tag8bit, OR, tmp, <<16]
.endlocal ; tmp

; tag8bit = (tag8bit >> 16) & 0x7f;
.local tmp
  alu_shf [tmp, --, B, tag8bit, <<9]
  alu_shf [tag8bit, --, B, tmp, >>25]
.endlocal ; tmp

;; Now that the tag has been transformed.  Use the upper 6 bits
;; of the 8-bit tag as an index into a table.
.local $xfer tmp addr
  alu_shf [tmp, --, B, tag8bit, >>2]
  alu_shf [addr, --, B, funcAndData, >>10]
  sram [read, $xfer, addr, tmp, 1], ctx_swap, defer[1]
```

```
    ;; Use the last two bits of the 8 bit tag as an index into the
    ;; long word.  The long word has 4, 8-bit fields.  If the drop
    ;; if the field is 1.
    alu_shf [tmp, 0x1F, AND, tag8bit, <<3]
    alu [--, tmp, B, 0]
    alu_shf [--, 0xF, AND, $xfer, >>indirect]

    br=0 [end_wavelet_dropper#]
    vrp_packet_drop [vrpResult]

  .endlocal ; $xfer tmp

.endlocal ; tag tag8bit
end_wavelet_dropper#:
nop
```

A.6 ACK Monitor

The ACK Monitor counts repeated TCP acknowledgments seen on a connection. It keeps a list of the last sixteen unique acknowledgment numbers and a count of how many times each was seen.

```
#include "libRegs.h"
#include "libStd.uc"
#include "libVrp.uc"

regs_packet
regs_state

; The position of the ACK flag depends on byte order.
#if LITTLE_ENDIAN
#define TCP_ACK_OFFSET 28
#else
#define TCP_ACK_OFFSET 4
#endif

;; Skip processing if the packet is not a TCP packet or doesn't
;; contain an acknowledgment
br!=byte [d5, IP_PROTO_BYTE, 6, end_ack_monitor#]
br_bclr [d11, TCP_ACK_OFFSET, end_ack_monitor#]

.local index acknum count tmp addr
  .local $xfer0 $xfer1
```

```
    .xfer_order $xfer0 $xfer1
    alu_shf [addr, --, B, funcAndData, >>10]
    sram [read, $xfer0, addr, 0, 1], ctx_swap, defer[1]

    ; Load half the current ACK number into tmp during the branch
    ; shadow
    load_lower_half [tmp, d10]

    alu [index, --, B, $xfer0]

    sram [read, $xfer0, addr, index, 2], ctx_swap, defer[1]

    ; Load the second half of the ACK number during another branch
    ; shadow.
    store_lower_half [tmp, d11]; Do useful work

    alu [acknum, --, B, $xfer0]
    alu [count, --, B, $xfer1]
  .endlocal

  ;; tmp now contains the ack number in the packet.
  ;; If ACK is repeated, branch to update
  alu [acknum, tmp, XOR, acknum]
  br=0 [ack_mon_update#], defer[1]

  ; set acknum to tmp during branch shadow
  alu [acknum, --, B, tmp]
  ; update the index to count repeats of different ACK
  alu [index, index, +4, 2]
  alu [count, --, B, 0]

ack_mon_update#:

  .local $xfer0 $xfer1
    .xfer_order $xfer0 $xfer1
    ; Increment ACK count
    alu [$xfer1, count, +, 1]
    alu [$xfer0, --, B, acknum]
    sram [write, $xfer0, addr, index, 2], ctx_swap

    ; Increment index of current ACK
    alu [$xfer0, --, B, index]
    sram [write, $xfer0, addr, 0, 1], ctx_swap

  .endlocal

.endlocal
end_ack_monitor#:
nop
```

A.7 SYN Monitor

The SYN counter counts the number of TCP packets with the SYN bit set.

```
#include "libRegs.h"
#include "libVrp.uc"

regs_packet
regs_state

;;; A SYN packet is identified by the SYN bit set in a TCP packet.
#if BIG_ENDIAN
#define TCP_SYN_OFFSET 1
#else
#define TCP_SYN_OFFSET 25
#endif

; Check the protocol number field in the IP packet
br!=byte [d5, IP_PROTO_BYTE, 6, end_syn_monitor#]

; Check for the SYN bit
br_bclr [d11, TCP_SYN_OFFSET, end_syn_monitor#]

; Read the counter from the memory.  Increment it.  Write the new
; value to the same location.
.local $count addr
  alu_shf [addr, --, B, funcAndData, >>10]
  sram [read, $count, state, 0, 1], ctx_swap
  alu [$count, $count, + , 1]
  sram [write, $count, funcAndData, 0, 1], ctx_swap
.endlocal

end_syn_monitor#:
nop
```

A.8 Port Filter

The port filter drops packets bound for a port in one of the five ranges indicated. Note that the loop in the code is unrolled by the preprocessor so the compiled code contains no backward branches.

```
#include "libRegs.h"
#include "libVrp.uc"
```

```
#include "libStd.uc"

regs_packet
regs_state

; Filter packets bound to the ports in the ranges indicated.
.local packet large tmp1 tmp2
  .local $xfer0 $xfer1 $xfer2 $xfer3 $xfer4
    .xfer_order $xfer0 $xfer1 $xfer2 $xfer3 $xfer4

    ; Load the restricted ranges.
    alu_shf [tmp1, --, B, funcAndData, >>10]
    sram [read, $xfer0, tmp1, 0, 5], ctx_swap, defer[1]

    ; Load the destination port in host order
    load_upper_half_ntoh [packet, d09]

    immed [large, 0xFFFF]

    ; For each of the 5 port ranges...
    #define_eval __pair_num 0
    #while (__pair_num < 5)
      ; If port > lower bound...
      alu [tmp1, large, -, packet]
      alu_shf [tmp1, tmp1, + , $xfer/**/__pair_num, >>16]

      br_bclr [tmp1, 16, port_filter_drop#], defer[3]

      ; and port < upper bound.  Use a local temporary variable to
      ; allowautomatic register allocation.
      .local tmp
        ld_field_w_clr [tmp2, 0011, $xfer/**/__pair_num]
        alu [tmp, large, -, tmp2]
        alu [tmp2, tmp, +, packet]
      .endlocal
        br_bclr [tmp2, 16, port_filter_drop#], defer[2]

      #define_eval __pair_num __pair_num + 1
    #endloop
    #undef __pair_num
  .endlocal
.endlocal
; For the last two defered instructions
nop
nop
br [port_filter_end#]

port_filter_drop#:
vrp_packet_drop [vrpResult]
```

```
port_filter_end#:
nop
```

Bibliography

[1] Internet Architecture Board. Internet Standard 9. File Transfer Protocol. Is also RFC0959 [45].

[2] Internet Architecture Board. Internet Standard 7. Transmission Control Protocol. Is also RFC0793 [44].

[3] Akamai Home Page. <URL:http://www.akamai.com/>, 2004.

[4] Active Network NodeOS Working Group. NodeOS Interface Specification. Available as <URL:http://www.cs.princeton.edu/nsg/papers/nodeos.ps>, Jan. 2000.

[5] D. S. Alexander, W. A. Arbaugh, M. W. Hicks, P. Kakkar, A. D. Keromytis, J. T. Moore, C. A. Gunter, S. M. Nettles, and J. M. Smith. The SwitchWare Active Network Architecture. *IEEE Network*, 12(3):29–36, May/June 1998.

[6] S. Athuraliya, V. H. Li, S. H. Low, and Q. Yin. REM: Active Queue Management. *IEEE Network*, May/June 2001.

[7] J. Aweya, M. Ouellette, D. Y. Montuno, and A. Chapman. A Control Theoretic Approach to Active Queue Management. *Computer Networks*, 36, 2001.

[8] M. L. Bailey, B. Gopal, M. A. Pagels, L. L. Peterson, and P. Sarkar. PATHFINDER: A Pattern-Based Packet Classifier. In *Proceedings of the First USENIX Symposium on Operating System Design and Implementation (OSDI)*, pages 115–123, Monterey, CA USA, Nov. 1994. USENIX Association.

[9] A. Bavier, T. Voigt, M. Wawrzoniak, L. Peterson, and P. Gunningberg. SILK: Scout Paths in the Linux Kernel. Technical Report 2002-009, Uppsala University, Feb. 2002.

[10] A. Begel, S. McCanne, and S. L. Graham. BPF+: Exploiting Global Data-flow Optimization in a Generalized Packet Filter Architecture. In *Proceedings of the Conference on Applications, Technologies, Architectures,*

and Protocols for Computer Communication, Cambridge, MA USA, Aug. 1999. ACM SIGCOMM.

[11] B. Bershad, S. Savage, P. Pardyak, E. G. Sirer, D. Becker, M. Fiuczynski, C. Chambers, and S. Eggers. Extensibility, Safety, and Performance in the SPIN Operating System. In *Proceedings of the Fifteenth ACM Symposium on Operating System Principles*, pages 267–284, Copper Mountain, CO USA, Dec. 1995.

[12] P. Bhagwat, D. A. Maltz, and A. Segall. MSOCKS+: An Architecture for Transport Layer Mobility. *Computer Networks*, 39(4):385–403, July 2002.

[13] D. D. Clark and W. Fang. Explicit Allocation of Best Effort Packet Delivery Service. *IEEE/ACM Transactions on Networking*, Aug. 1998.

[14] B. Davie and Y. Rekhter. *MPLS: Technology and Applications*. Morgan Kaufmann Publishers, Inc., 2000.

[15] D. Decasper, Z. Dittia, G. Parulkar, and B. Plattner. Router Plugins: A Software Architecture for Next Generation Routers. *IEEE/ACM Transactions on Networking*, 8(1):2–15, Feb. 2000.

[16] S. Deering. Host extensions for IP multicasting. Request for Comments 1112, Network Working Group, Aug. 1989.

[17] A. Demers, S. Keshav, and S. Shenker. Analysis and Simulation of a Fair Queueing Algorithm. *Journal of Internetworking Research and Experience*, 1(1):3–26, Jan. 1990.

[18] P. Druschel and A. Rowstron. PAST: A Persistent and Anonymous Store. In *Proceedings of the Eighth Workshop on Hot Topics in Operating Systems (HotOS–VIII)*. USENIX Association, May 2001.

[19] D. R. Engler, M. F. Kaashoek, and J. O'Toole, Jr. Exokernel: An Operating System Architecture for Application-Level Resource Management. In *Proceedings of the Fifteenth ACM Symposium on Operating System Principles*, pages 251–266, Copper Mountain, CO USA, Dec. 1995.

[20] G. Fankhauser, M. Dasen, N. Weiler, B. Plattner, and B. Stiller. The WaveVideo System and Network Architecture: Design and Implementation. Technical Report 44, Institute TIK, Gloriastrasse 35, 8092 Zürich, Switzerland, June 1998.

[21] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Hypertext Transfer Protocol—HTTP/1.1. Request for Comments 2616, Network Working Group, June 1999.

[22] S. Floyd and V. Jacobson. Random Early Detection Gateways for Congestion Avoidance. *IEEE/ACM Transactions on Networking*, 1(4):397–413, Aug. 1993.

[23] S. J. Friedman and K. J. Supowit. Finding the Optimal Variable Ordering for Binary Decision Diagrams. In *24th ACM/IEEE Conference Proceedings on Design Automation Conference*, pages 348–356, Miami Beach, FL USA, June 1987.

[24] Y. Gottlieb and L. Peterson. A Comparative Study of Extensible Routers. In *2002 IEEE Open Architectures and Network Programming Proceedings*, pages 51–62, New York, NY USA, June 2002.

[25] M. Hicks, P. Kakkar, J. T. Moore, C. A. Gunter, and S. Nettles. PLAN: A Packet Language for Active Networks. In *Proceedings of the Third International Conference on Functional Programming Languages*, pages 86–93, San Diego, CA USA, Sept. 1998. ACM SIGPLAN.

[26] G. Hjálmtýsson. The Pronto Platform: A Flexible Toolkit for Programming Networks using a Commodity Operating System. In *2000 IEEE Third Conference on Open Architectures and Network Programming Proceedings*, pages 98–107, Tel Aviv, Israel, Mar. 2000. IEEE Communications Society.

[27] IBM Microelectronics Division. *IBM PowerNP NP4GS3 Network Processor Solutions Product Overview*, Apr. 2001.

[28] Intel Corporation. *IXP1200 Network Processor Datasheet*, Sept. 2000.

[29] S. C. Karlin. *Embedded Computational Elements in Extensible Routers*. PhD thesis, Princeton University, Princeton, NJ USA, Jan. 2003.

[30] D. Katabi, M. Handley, and C. Rohrs. Congestion Control for High Bandwidth-Delay Product Networks. In *Proceedings of the ACM SIGCOMM 2002 Conference Applications, Technologies, Architectures, and Protocols for Computer Communication*, pages 89–102, Pittsburgh, PA USA, Aug. 2002.

[31] T. Klingberg and R. Manfredi. Gnutella 0.6. `<URL:http://groups.yahoo.com/group/the_gdf/files/Development/GnutellaProtoco%l-v0.6-200206draft.txt>`.

[32] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek. The Click Modular Router. *ACM Transactions on Computer Systems*, 18(3):263–297, Aug. 2000.

[33] T. Lavian and P. Y. Wang. Active Networking on a Programmable Network Platform. In *Hot Interconnects 7: A Symposium on High Performance Interconnects*, Stanford, CA USA, Aug. 1999. IEEE Computer Society.

[34] K. McCloghrie and M. Rose. Management Information Base for network management of TCP/IP-based internets. Request for Comments 1156, Network Working Group, May 1990.

[35] D. Mosberger and L. L. Peterson. Making Paths Explicit in the Scout Operating System. In *Proceedings of the Second USENIX Symposium on Operating System Design and Implementation (OSDI)*, pages 153–167, Oct. 1996.

[36] J. Moy. OSPF Version 2. Request for Comments 2328, Network Working Group, Apr. 1998.

[37] B. Mukherjee, L. T. Heberlein, and K. N. Levitt. Network Intrusion Detection. *IEEE Network*, 8(3):26–41, May/June 1994.

[38] K. Nichols and B. Carpenter. Definition of Differentiated Services Per Domain Behaviors and Rules for their Specification. Request for Comments 3086, Network Working Group, Apr. 2001.

[39] C. Partridge, T. Mendez, and W. Milliken. Host Anycasting Service. Request for Comments 1546, Network Working Group, Nov. 1993.

[40] C. Perkins. IP Mobility Support. Request for Comments 2002, Network Working Group, Oct. 1996.

[41] L. Peterson, T. Anderson, D. Culler, and T. Roscoe. A Blueprint for Introducing Disruptive Technology into the Internet. In *Proceedings of the First ACM Workshop on Hot Topics in Networks (HotNets-I)*, pages 59–64, Oct. 2002.

[42] J. Postel. User Datagram Protocol. Request for Comments 768, Network Working Group, Aug. 1980.

[43] J. Postel. Internet Protocol. Request for Comments 791, Network Working Group, Sept. 1981.

[44] J. Postel. Transmission Control Protocol. Request for Comments 793, Network Working Group, Sept. 1981.

[45] J. Postel and J. Reynolds. File Transfer Protocol. Request for Comments 959, Network Working Group, Oct. 1985.

[46] X. Qie, A. Bavier, L. Peterson, and S. Karlin. Scheduling Computations on a Software-Based Router. In *Proceedings of the ACM SIGMETRICS 2001 Conference*, pages 13–24, June 2001.

[47] M. J. Ranum, K. Landfield, M. Stolarchuk, M. Sienkiewicz, A. Lambeth, and E. Wall. Implementing a Generalized Tool For Network Monitoring. In *Proceedings of the Eleventh Systems Administration Conference (LISA '97)*, pages 1–8, San Diego, CA, Oct. 1997. USENIX Association.

[48] Y. Rekhter and T. Li. A Border Gateway Protocol 4 (BGP-4). Request for Comments 1771, Network Working Group, Mar. 1995.

[49] A. Rowstron and P. Druschel. Pastry: Scalable, Distributed Object Location and Routing for Large-scale Peer-to-peer Systems. In *IFIP/ACM International Conference on Distributed Systems Platforms (Middleware)*, pages 329–350, Nov. 2001.

[50] A. Rowstron and P. Druschel. Storage Management and Caching in PAST, a Large-scale, Persistent Peer-to-Peer Storage Utility. In *Proceedings of the Eighteenth ACM Symposium on Operating System Principles*, pages 188–201, Chateau Lake Louise, Banff, Canada, Oct. 2001.

[51] A. Rowstron, A.-M. Kermarrec, M. Castro, and P. Druschel. Scribe: The Design of a Large-scale Event Notification Infrastructure. In J. Crowcroft and M. Hofmann, editors, *Networked Group Communication, Third International COST264 Workshop (NGC'2001)*, volume 2233 of *Lecture Notes in Computer Science*, pages 30–43, Nov. 2001.

[52] S. Savage, D. Wetherall, A. Karlin, and T. Anderson. Practical Network Support for IP Traceback. In *Proceedings of the ACM SIGCOMM 2000 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, pages 295–306, Stockholm, Sweden, Aug. 2000.

[53] N. Shalaby, Y. Gottlieb, L. Peterson, and M. Wawrzoniak. Snow on Silk: A NodeOS in the Linux Kernel. In *Active Networks: IFIP-TC6 4th International Working Conference, IWAN 2002, Zurich, Switzerland, December 4–6, 2002, Proceedings*, Lecture Notes in Computer Science, pages 1–19, Zürich, Switzerland, Dec. 2002. Springer.

[54] M. Shreedar and G. Varghese. Efficient Fair Queueing Using Deficit Round-Robin. *IEEE/ACM Transactions on Networking*, 4(3):375–385, June 1996.

[55] T. Spalink, S. Karlin, L. Peterson, and Y. Gottlieb. Building a Robust Software-Based Router Using Network Processors. In *Proceedings of the Eighteenth ACM Symposium on Operating System Principles*, pages 216–229, Chateau Lake Louise, Banff, Alberta, Canada, Oct. 2001.

[56] O. Spatscheck, J. Hansen, J. Hartman, and L. Peterson. Optimizing TCP Forwarder Performance. Technical Report 89-6, Department of Computer Science, University of Arizona, Feb. 1998.

[57] P. Srisuresh and M. Holdrege. IP Network Address Translator (NAT) Terminology and Considerations. Request for Comments 2663, Network Working Group, Aug. 1999.

[58] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications. In *Proceedings of the 2001 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, pages 149–160, San Diego, CA USA, Aug. 2001. ACM SIGCOMM.

[59] Sun Microsystems. RPC: Remote Procedure Call Protocol specification: Version 2. Request for Comments 1057, Network Working Group, June 1988.

[60] J. Touch. Dynamic Internet Overlay Deployment and Management Using the X-Bone. *Computer Networks*, 36(2):117–135, July 2001.

[61] Vitesse Semiconductor Corporation. *IQ2000 Network Processor Product Brief*, 2000.

[62] M. Wawrzoniak, N. Shalaby, and L. Peterson. Intelligent Devices as Symmetric Partners for End-to-end Data Flows. Technical Report TR–642–02, Department of Computer Science, Princeton University, July 2002.

[63] P. Weis and X. Leroy. *Le langage Caml*. Dunod, 1999.

[64] D. Wetherall, J. Guttag, and D. Tennenhouse. ANTS: A Toolkit for Building and Dynamically Deploying Network Protocols. In *1998 IEEE Open Architectures and Network Programming*, pages 117–129, San Francisco, CA USA, Apr. 1998.

[65] S. Yadav, S. Bakshi, D. Putzolu, and R. Yavatkar. The Phoenix Framework: A Practical Architecture for Programmable Networks. *Intel Technology Journal*, 3(3):1–7, Q3 1999.

[66] L. Zhang, S. Deering, D. Estrin, S. Shenker, and D. Zappala. RSVP: A New Resource ReSerVation Protocol. *IEEE Network*, 7(5):8–18, Sept. 1993.