Report Number: WUCS-98-08

1998-01-01

Router Plugins: A Modular and Extensible Software Framework for Modern High Performance Integrated Services Routers

Dan Decasper, Zubin Dittia, Guru Parulkar, and Bernhard Plattner

Present day routers typically employ monolithic operating systems which are not easily upgraded and extensible. WIth the rapid rate of protocol development it is becoming increasingly important to dynamically upgrade router software in an incremental fashion. We have designed and implemented a high performance, modular, extended integrated services router software architecture in the NetBSD operating system kernel. This architecture allows code modules, called plugins, to be dynamically added and configured at run time. One of the novel features of our design is the ability to bind different plugins to individual flows; this allows for distinct plugin implementations to seamlessly... **Read complete abstract on page 2.**

Router Plugins: A Modular and Extensible Software Framework for Modern High Performance Integrated Services Routers

Dan Decasper, Zubin Dittia, Guru Parulkar, and Bernhard Plattner

Complete Abstract:

Present day routers typically employ monolithic operating systems which are not easily upgraded and extensible. WIth the rapid rate of protocol development it is becoming increasingly important to dynamically upgrade router software in an incremental fashion. We have designed and implemented a high performance, modular, extended integrated services router software architecture in the NetBSD operating system kernel. This architecture allows code modules, called plugins, to be dynamically added and configured at run time. One of the novel features of our design is the ability to bind different plugins to individual flows; this allows for distinct plugin implementations to seamlessly coexist in the same runtime environment. High performance is achieved through a carefully designed modular architecture; an innovative packet classification algorithm that is both pwerful and highly efficient; and by caching that exploits the flow-like characteristics of Internet traffic. Compared to a monolithic best-effort kernel, our implementation requires an average increase in packet processing overhead of only 8%, or 500 cycles/2.1microsecond per packet when running on a P6/233.

Router Plugins: A Modular and Extensible Software Framework for Modern High Performance Integrated Services Routers

Dan Decasper, Zubin Dittia, Guru Parulkar and Bernhard Plattner

WUCS-98-08

February 1998

Department of Computer Science
Washington University
Campus Box 1045
One Brookings Drive
St. Louis MO 63130

Router Plugins:
A Modular and Extensible Software Framework for Modern High Performance Integrated Services Routers

Dan Decasper[1], Zubin Dittia[2], Guru Parulkar[2], Bernhard Plattner[1]
[dan|plattner]@tik.ee.ethz.ch
[zubin|guru]@arl.wustl.edu
[1]Computer Engineering and Networks Laboratory, ETH Zurich, Switzerland
Phone: +41-1-632 7019 Fax: +41-1-632 1035
[2]Applied Research Laboratory, Washington University, St. Louis, USA
Phone: +1-314-935 4586 Fax: +1-314-935 7302

Abstract

Present day routers typically employ monolithic operating systems which are not easily upgradable and extensible. With the rapid rate of protocol development it is becoming increasingly important to dynamically upgrade router software in an incremental fashion. We have designed and implemented a high performance, modular, extended integrated services router software architecture in the NetBSD operating system kernel. This architecture allows code modules, called plugins, to be dynamically added and configured at run time. One of the novel features of our design is the ability to bind different plugins to individual flows; this allows for distinct plugin implementations to seamlessly coexist in the same runtime environment. High performance is achieved through a carefully designed modular architecture; an innovative packet classification algorithm that is both powerful and highly efficient; and by caching that exploits the flow-like characteristics of Internet traffic. Compared to a monolithic best-effort kernel, our implementation requires an average increase in packet processing overhead of only 8%, or 500 cycles/2.1µs per packet when running on a P6/233.

1 Introduction

New network protocols and extensions to existing protocols are being deployed on the Internet. New functionality is being added to modern IP routers at an increasingly rapid pace. In the past, the main task of a router was to simply forward packets based on a destination address lookup. Modern routers, however, incorporate several new services:

- Integrated Services
- Enhanced routing functionality (level 3 and level 4 routing and switching techniques)
- Security algorithms (e.g. to implement virtual private networks (VPN))
- Enhancements to existing protocols (e.g. Random Early Detection (RED))
- New core protocols (e.g. IPv6 [7])

Figure 1 contrasts the software architecture of our proposed Extended Integrated Services Router (EISR) with that of a conventional best-effort router. A typical EISR kernel features the following important additional components: a packet scheduler, a packet classifier, security mechanisms, and QoS-based routing/Level 4 switching. Various algorithms and implementations of each component offer specific advantages in terms of performance, feature sets, and cost. Most of these algorithms undergo a constant evolution and are replaced and upgraded frequently. Such networking subsystem components are characterized by a relatively "fluid" implementation, and should be distinguished from the small part of the network subsystem code that remains relatively stable. The stable part (called the core) is mainly responsible for interacting with the network hardware and for demultiplexing packets to specific modules. Different implementations of the EISR components outside of the core often need to coexist. For example, we might want to use one kind of packet scheduling on one interface, and a different kind on another.

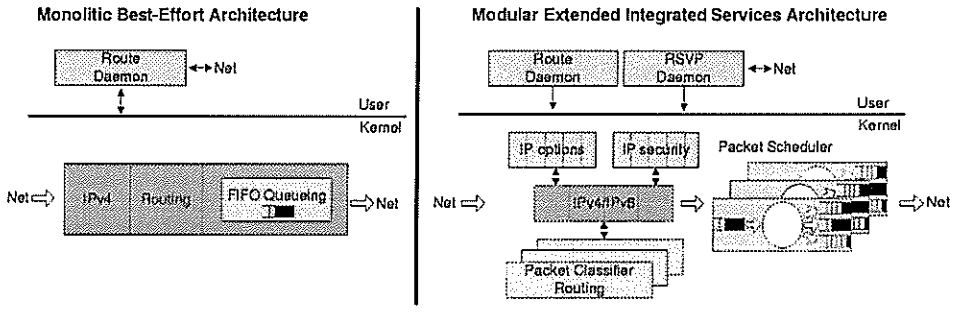

Figure 1: Best Effort Router vs Extended Integrated Services Router (EISR)

In this paper, we propose a software framework and present an implementation which addresses these requirements. The specific goals of our framework are:

- **Modularity**: Implementation of specific algorithms come in the form of modules called *plugins*[†].
- **Extensibility**: New plugins can be dynamically loaded at run time.
- **Flexibility**: Instances of plugins can be created, configured, and *bound to specific flows*.
- **Performance**: The system should provide for a very efficient data path, with no data copying, no context switching, and no additional interrupt processing. The overhead of modularity should not seriously impact performance.

[†] A note on our use of the word 'plugin' (instead of 'module') is in order. In the web browser world, a plugin is a software module that is dynamically linked with the browser and is responsible for processing certain types of application streams (or flows). In a similar fashion, our router plugins are kernel software modules that are dynamically loaded into the kernel and are responsible for performing certain specific functions on specified network flows.

Our proposed framework has been implemented in the NetBSD UNIX kernel. This platform was selected because of its portability (all major hardware platforms are supported), efficiency, and extensive documentation. In addition, we found state-of-the-art implementations on this platform for IPv6 [12] and packet schedulers [25, 5] that could be integrated into our framework.

We envision several applications for our framework. First, our architecture fits very well into the operating system of small and mid-sized routers. It is particularly well suited to the implementation of modern edge routers that are responsible for doing flow classification, and for enforcing the configured profiles of differential service flows. This kind of enforcement can be done either on a per-application flow basis, or on a generalized class-based approach (e.g. CBQ [10]). Our implementation supports both models efficiently.

Our framework is also very well suited to Application Layer Gateways (ALGs), and to security devices like Firewalls. In both situations, it is very important to be able to quickly and efficiently classify packets into flows, and to apply different policies to different flows: these are both things that our architecture excels at doing.

Yet another application of our framework is for network management applications, which typically need to monitor transit traffic at routers in the network, and to gather and report various statistics thereof. For such applications, it is important to be able to quickly and easily change the kinds of statistics being collected, and to do this without incurring significant overhead on the data path.

Finally, while our proposed framework is very useful in real-world implementations, its modularity and extensibility also make it an invaluable tool for researchers. We plan to release all of our code in the public domain and we will attempt to incorporate several core portions into the standard NetBSD distribution tree.

The main contributions of our work are:

- An innovative, modular, extensible, and flexible EISR networking subsystem architecture and implementation that introduces only 8% more overhead than a best-effort kernel.
- A very fast packet classifier algorithm which provides highly competitive upper bounds for classification times. With a very large number of filters (in the order of 50000), it classifies IPv6 packets in 24 memory accesses, and is much faster for smaller numbers of filters.
- Implementations of plugins for two state-of-the-art packet schedulers: Deficit Round Robin (DRR, [22]) for fair queuing, and the Hierarchical Fair Service Curves (H-FSC, [25]) scheduler for class-based packet scheduling.

There are a few commercial attempts that we are aware of which follow similar lines. The latest versions of Cisco's Internet OS (IOS, [6]) claims to fulfill some of the requirements, but since it's a commercial operating system, there is no easy access for the research community and these claims are not verifiable. Microsoft's Routing and Remote Access Service for Windows NT (RRAS, previously referred to as "Steelhead" [17, 18]) is an attempt to implement router functionality under Windows NT. RRAS exports an API and allows third party modules to implement routing protocols like OSPF and SNMP agents in user space. The API does not provide an interface to the routing and forwarding engines, and the platform offers no integrated services components. A few research projects attempt to achieve some of the goals mentioned above [11, 19, 20]. Most of them are focused on the implementation of modular **end-system** networking subsystems instead of routing architectures. *Scout* from the University of Arizona is a particularly interesting project based on the x-kernel that implements an operating system targeted at network appliances (including routers). It comes with router components implementing simple QoS support. Since the whole operating system is implemented from scratch, most of the provided functionality is over-simplified and does not provide the large feature set that is found in mature implementations. Later in this paper, we shall look at some of these commercial and research projects in more detail.

In Section 2, we describe our architecture and explain how it achieves modularity, extensibility, and flexibility while maintaining high-performance. In Section 3, we describe the implementation of a module called the Plugin Control Unit (PCU), which is responsible for all control path interactions with plugins. Section 4 outlines the implementation of the Association Identification Unit (AIU), which is used by almost all other components in our design. The AIU implements an innovative algorithm for packet classification which efficiently maps packets to code modules (plugins). In Section 5, we elaborate on two example plugins (packet schedulers) which we implemented or adapted for our environment. Section 6 presents performance results from our implementation. Section 7 relates our work to that of others, and Section 8 summarizes our ideas.

2 Overall Architecture

The primary goal of our proposed architecture was to build a modular and extensible networking subsystem that supported the concept of flows, and the ability to select implementations of components based upon flows (in addition to simple static configurations). Because the deployment of multimedia data sources and applications (e.g. real-time audio/video) will produce longer lived packet streams with more packets per session than is common in today's environment, an integrated services router architecture should support the notion of flows and build upon it. In particular, the locality properties of flows should be effectively exploited to provide for a highly efficient data path. Our plugin framework features:

- Dynamic loading and unloading of plugins at run time into the networking subsystem. Plugins are code modules which implement a specific EISR functionality (e.g. packet scheduling). NetBSD offers a simple yet powerful mechanism which allows modules to be loaded into the kernel. This mechanism is used to load our plugins into the kernel. Once a plugin is loaded, it is no different from any other kernel code. What is required for our system is a component which glues the individual plugins to the networking subsystem, and which provides a control-path interface used by other kernel components (possibly also other plugins) and user space daemons to talk to the plugin. In our system, this component is called the Plugin Control Unit (PCU). The PCU hides most of the implementation specific details from the individual plugins and allows them to access the system in a simple yet flexible fashion.
- Creation of individual instances of plugins for maximal flexibility. An instance is a specific run-time configuration of an individual plugin. It is often very desirable to have multiple instances of one and the same plugin concurrently in the kernel. For example, consider packet scheduling. A packet scheduler can work with different configurations on different network interfaces. State-of-the-art packet schedulers are usually hierarchical, with possibly different modules working on different levels of the scheduling hierarchy. Among the nodes of the same level, modules are specifically configured, which means that they coexist in our framework as plugin instances. In order to provide a simple and unified interface for the allocation of multiple instances of one and the same plugin, the plugins must respond to a set of standardized messages. By standardizing this message set and implementing it in all plugins, we guarantee interoperability among different plugins and provide a simple configuration interface.
- Efficient mapping of individual data packets to flows, and the ability to bind flows to plugin instances. Sets of flows are specified using *filters*. For example, a filter might match all TCP traffic from the network 129.0.0.0 to the host 192.94.233.10. Filters can also match individual end-to-end application flows. Filters are specified as six-tuples:

  <source address, destination address, protocol, source port, destination port, incoming interface>

  Any of the fields in the six tuple may be wildcarded. Additionally, for network addresses, a prefix mask may be used to partially wildcard the corresponding field. For instance, for the above example, the filter specification would read:

  <129..*.*, 192.94.233.10, TCP, *, *, *>*

  Clearly, the filter for an end-to-end application flow would have all fields (except perhaps the incoming interface) fully specified. We will see later in this section that a packet matching a particular filter will be passed to the plugin instance that has been bound to that filter. This will be shown to happen whenever the packet reaches a "gate" in the IP stack; a gate can be thought of as the entry point for a plugin.

- Overall high performance. High performance is guaranteed only in part through a fully kernel space implementation which prevents costly context switches. We identified two other critical properties which, when combined, guarantee high performance even in a highly modular environment: the flow-like nature of most internet traffic, and the ability to classify packets into flows quickly and efficiently. As we show below, the filter lookup to determine the right plugin instance to which a packet should be passed happens only for the first packet of a burst. Subsequent packets get this information from a fast flow cache which temporarily stores the information gathered by processing the first packet. The filter lookup itself is efficiently implemented using a Directed Acyclic Graph (DAG). We elaborate on these techniques later in this section, and also in section 4

In order to describe our framework, we first look at the different components and how they interact in the control path. In the Section 2.2, we will look at the data path, and how individual packets are processed by our architecture.

2.1 The Control Path

Figure 2 shows the architecture of our system and the control communication between different components. A description of the different components follows:

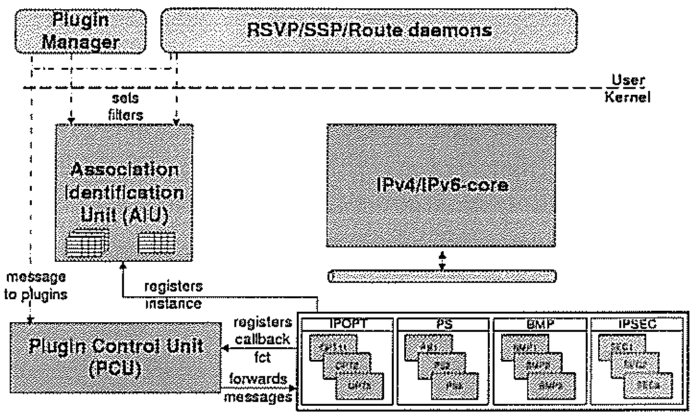

Figure 2: System Architecture and Control Path

- **IPv4/IPv6 core:** The IPv4/IPv6 core consists of a stream-lined IPv4/IPv6 implementation which contains the (few) components required for packet processing which do not come in the form of dynamically loadable modules. These are mainly functions that interact with network devices. The core is also responsible for demultiplexing individual packets to plugins as we will show in the next section. There are no plugin related control path interactions with the IP core.
- **Plugins:** Figure 2 shows four different types of plugins — plugins implementing IPv6 options, plugins for packet scheduling, plugins to calculate the best-matching prefix (BMP, used for packet classification and routing), and plugins for IP security. Other plugin types are also possible: e.g., a routing plugin, a statistics gathering plugin for network management applications, a plugin for congestion control (RED), a firewall plugin. Note that all plugins come in the form of dynamically loadable kernel modules.
- **Plugin Control Unit (PCU):** The PCU manages plugins, and is responsible for forwarding messages to individual plugins from other kernel components, as well as from user space programs (using library calls).
- **Association Identification Unit:** The Association Identification Unit (AIU) implements a packet classifier and builds the glue between the flows and plugin instances. The operation of the AIU will become clear when we describe the data path in the next subsection.
- **Plugin Manager:** The Plugin Manager is a user space utility used to configure the system. It

is a simple application which takes arguments from the command line and translates them into calls to the user-space *Router Plugin Library* which we provide with our system. This library implements the function calls needed to configure all kernel level components. In most cases, the plugin manager is invoked from a configuration script during system initialization, but it can also be used to manually issue commands to various plugins. We show an example of how the Plugin Manager is used in Section 5.

- **Daemons:** The RSVP [29], SSP [1] (a simplified version of RSVP), and route daemon are linked against the Router Plugin Library to perform their respective tasks. We implemented an SSP daemon for our system, and are currently in the process of porting an RSVP implementation.

After a reboot, the system has to be configured before it is ready to receive and forward data packets. Configuration involves the selection of a set of plugins. Since a selection does not necessarily apply to all packets traversing the router, a definition of the set of packets which should be processed by each individual plugin instance is required. This configuration can be done either by a system administrator, or by executing a script. Configuration involves the following steps:

- **Loading a plugin:** Using the *modload* command, which is part of the NetBSD distribution, plugins are loaded into the kernel. On loading, they register themselves with the PCU by providing a callback function. This function is used to send messages to the plugin. There are messages for creating and freeing instances of the plugin and for binding plugin instances to flows. Also, plugin developers can define an arbitrary number of plugin specific messages. Once the callback function for a plugin has been registered, the PCU can forward these configuration messages to the plugin.
- **Creating an instance of a plugin:** Using the Plugin Manager application, configuration messages can be sent to specified plugins. Typically, these messages ask the plugin to create an instance of itself. In case of a packet scheduling plugin for example, the configuration information could include the network interface the plugin should work on.
- **Creating filters:** Once a plugin has been configured and an instance has been created, it is ready to be used. What has to be defined next is the set of datagrams which should be passed to the instance for processing. This is done by binding one or more flows to the plugin instance. To specify the set of flows that are supposed to be handled by a particular plugin instance, the Plugin Manager or one of the user space daemons (RSVP or SSP) can create filters through calls to the AIU. Recall (from earlier in this section) that a filter is a specification for the set of flows it matches.
- **Binding flows to instances:** Next, the binding between filters and plugin instances must be established. Each filter in the AIU is associated with a pointer to a plugin instance; this pointer is set by making another call to the AIU to do the binding.

Now the system is ready to process data packets. We will show in the next subsection how data packets are matched against filters and how they get passed to the appropriate instances.

2.2 The Data Path

Data packets in our system are passed to instances of plugins which implement the specific functions for processing the packets. Since data path mechanisms are applied to every single packet, it is very important to optimize their performance. Given a packet, our architecture should be able to quickly and efficiently discover the set of instances that will act on the packet.

The data path interactions are shown in Figure 3. Before we can explain the sequence of actions, we have to introduce the notion of a gate. A ***gate*** is a point in the IP core where the flow of execution branches off to an instance of a plugin. From an implementation point of view, gates are simple macros which encapsulate function calls to the AIU that will return the correct plugin instance which is to be used for processing the packet. In many cases, these macros can avoid a function call to the AIU altogether, thereby permitting a more efficient implementation. Gates are placed wherever interactions with plugins need to take place. For example, sometimes after a packet is received by the hardware, option processing has to be done if the packet contains IP options. In our system, IP option processing functions are modularized and come in the form of plugins. A gate is inserted into the IP core code in place of the traditional call to the kernel function responsible for IP option processing. In our current implementation, we use gates for IPv6 option processing, IP security, packet scheduling, and for the packet filter's best-matching prefix algorithm.

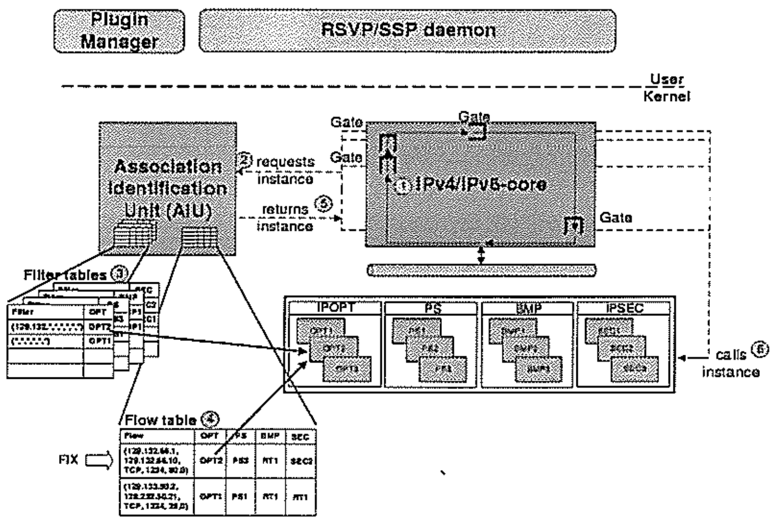

Figure 3: System Architecture and Data Path

To follow the various data path interactions, it is important to get a basic understanding of the operation of the AIU. The AIU is responsible for maintaining the binding between flows and plugin instances. It makes use of a special data structure called a **flow table** to cache flows. Flow tables allow for very fast lookup times for arriving packets that belong to cached flows.

In the AIU, all flows start out being uncached (i.e., they do not have an entry in the flow table). If an incoming packet belongs to an uncached flow, its lookup in the flow table data structure will fail (i.e., there is a cache miss). In this case, the packet needs to be looked up in a different data structure that we call a **filter table**. Filter tables store the bindings between filters and plugins for each gate. The filter table lookup algorithm finds the most specific matching filter (described later) that has been installed in the table, and returns the corresponding plugin instance. Usually, filter table lookups are much slower than flow table lookups. An entry for a flow in the flow table serves as a fast cache for future lookups of packets belonging to that flow. Each flow table entry stores pointers to the appropriate plugins for all gates that can be encountered by packets belonging to the corresponding flow. The processing of the first packet of a new flow with n gates involves n filter table lookups to create a single entry in the flow table for the new flow.

If a cached flow remains idle (i.e., no new packets are received) for an extended period, its cached entry in the flow table data structure may be removed (or replaced by a different flow). In this case, if the flow becomes active again, the first packet that is received would again result in a cache miss, which would again cause a new cache entry to be created in the flow table so that subsequent packets can benefit from faster lookup times.

Section 4.1 describes a very fast filter table lookup implementation based on directed acyclic graphs (DAGs). Section 4.2 describes our flow table implementation, which is based on hashing.

As an example, consider the steps involved in processing an IPv6 packet (see numbers 1-6 in Figure 3). Uncached flow processing involves the following sequence of events and actions:

0. **Packet arrival:** When a packet arrives, it gets passed to the IP core by the network hardware. As it makes its way through the core, it may encounter multiple gates.
1. **Encountering a gate:** Assume that the packet has reached the gate where IPv6 option processing will be handled. The task of this gate is to find the plugin instance which is responsible for processing the IPv6 options contained in the packet.
2. **Discovering the right instance:** The gate makes a call to the AIU. The parameters of the call are a pointer to the packet and an identification of the gate issuing the call. In our case, we would identify the IP options gate as the caller.
3. **Packet classification:** The AIU first does a lookup in the flow table, and finds that there is no cached entry available for the flow. Consequently, it performs a lookup in the filter table corresponding to the IPv6 options gate. The resulting plugin instance pointer is returned to the calling gate ("OPT2" in Figure 3). Note that since this packet classification step performed by the AIU is the most expensive step in the whole cycle, an efficient packet classification scheme and implementation is important.
4. **Caching of the instance pointer:** Before the AIU returns the instance pointer to the gate, it stores the pointer in the flow table. Note that entries in the flow table are identified by the same six tuple used to specify filters, but without masks or wildcards (all fields have fully specified values). In other words, a flow table entry unambiguously identifies a particular flow. In our example, the pointer to the OPT2 plugin is stored in the row of the flow table which corresponds to our packet's flow.
5. **Returning the instance pointer:** The instance pointer found is returned to the gate.
6. **Calling the instance:** The gate calls the plugin instance, passing the packet as an argument.
7. **Repeating the cycle:** When the call returns, the IP stack continues processing the packet, until it encounters another gate, in which case the same cycle repeats.

This cycle is executed only for the first packet arriving on an uncached flow. Subsequent packets follow a faster path because of the cached entry in the flow table. Note that in our system, we have created optimized implementations of both the flow and filter tables, allowing for high performance on both the cached and uncached paths. These implementations are described in Section 4.

Cached flow processing involves the following sequence:

- **Processing at the first gate:** When a packet from a cached flow encounters the **first** gate, the AIU is called to request the plugin instance. This time, the pointer to the instance requested is already in the flow table. The flow table is looked up efficiently, and the plugin instance pointer corresponding to the calling gate is returned. No filter table lookups are required.
- **Associating the packet with a flow index:** Together with the instance requested, the AIU returns a pointer to the row in the flow table where the information associated with the flow is stored. This pointer is called the flow index (FIX), and is stored in the packet's mbuf[1]. The instance is then called to process the packet, following which the IP stack passes the packet on

[1] The mbuf is a data structure that is used to store packets and packet related information efficiently in BSD derived operating system kernels.

to the next gate.

- **Processing at subsequent gates:** Once the packet has made its way past the first gate, the AIU does not have to be called upon to classify the packets at the remaining gates. Macros implementing a gate can retrieve the instance pointers cached in the flow table by accessing the FIX stored in the packet. This allows us to pass packets to the appropriate instances in a very efficient manner using an indirect function call instead of a "hardwired" function call. We show in section 6 that this does not imply significant performance penalties.

Our architecture implements a highly modular system with minimal performance overhead. Our architecture is scalable to a very large number of gates since the number of gates matters only for the first packet arriving on a (uncached) flow. But even for the first packet, fast retrieval of the instance is possible with the DAG based packet classification algorithm that is used to implement the filter tables in our system (see Section 4).

3 Plugins and the Plugin Control Unit (PCU)

Depending on the type of network software component that is implemented by a plugin, it can be very simple (e.g., a dozen lines of code for an IP option plugin) or very complex (e.g., a state-of-the-art packet scheduler). Each plugin in our framework is identified by a name and a type. The type of a plugin refers to the specific network software component it implements; thus, there is a direct correspondence between a gate in our architecture and the plugin type. Whenever a packet enters a gate, it will be passed to a registered plugin of the appropriate type. There can potentially be multiple plugins of the same type that have been registered; in this case, flow filters that have been installed for the corresponding plugin type are used to pick the right plugin to which the packet should be passed.

Our implementation currently supports four types of plugins, corresponding to different network functions: IP options, IP security, Packet Scheduling, and Longest-prefix Matching (used as part of the packet classifier that is present in the AIU). In the future, we plan to also add support for a Routing plugin, which would allow routing table lookups to be based on the flow classification that is performed by the AIU. Other plugins that are envisioned include a plugin for statistics gathering (useful for network monitoring/management), a plugin for congestion control mechanisms (e.g., RED), and a plugin for firewall functions. Doubtless, additional plugin types will be introduced by third parties once we have released our code into the public domain. We will discuss the implementation of two example plugins in section 5.

Plugins must fulfill two important requirements: they have to register a callback function with the PCU when they are loaded into the kernel, and that callback function must reply to a set of messages. As mentioned earlier, these messages fall into two categories: standardized messages, and plugin-specific messages. The set of standardized messages include:

- **create_instance**: Creates an instance of a plugin. This results in the allocation of a data structure that will be used to store configuration and run-time information for that instance. A function to handle a data packet (the main packet processing function which is called at the gate) must be specified and functions which are called by the AIU on removal of an entry in the flow or filter table can optionally be specified.
- **free_instance**: Removes all instance specific data structures. A freed instance can no longer be used by the kernel and all references to it are removed from the flow table and the filter table.
- **register_instance**: Registers a plugin instance with the AIU, and binds that instance to a filter that has to be supplied as a parameter. The same instance may be registered multiple times with the AIU with different filter specifications. This message would result in a call to a registration function that is published by the AIU.
- **deregister_instance**: Removes the binding between a specified filter in the AIU and the plugin instance.

Creating and freeing instances is a highly plugin specific task. Registering and deregistering instances with the AIU and a supplied filter is a relatively simple task; in most cases, this merely results in a call to the corresponding AIU function.

The PCU itself is a very simple component (200 lines of C code) managing a table for each plugin type to store the plugin's names and callback functions. Once loaded into the kernel, plugins register their callback function through a function call to the PCU. All control path communication to the plugins goes through the PCU. Usually, such messages come from user space, either from the Plugin Manager or from one of the daemons using a library call. The PCU is responsible for dispatching these messages to the target plugin, and for handling exceptions. We implemented a dedicated socket type for all plugin related user space communication with the kernel, which is similar to the routing socket that is used by `routed` to communicate with the routing engine in a BSD-based kernel.

4 The Association Identification Unit (AIU)

The Association Identification Unit (AIU) is the most important component in our proposed framework. It implements a packet classifier, fast flow detection, and provides the binding between plugin instances and filters. To do so, it manages two main data structures: filter tables and a flow table. In Section 2.2, we described how flow and filter tables are used; in this section, we will describe their implementations.

4.1 Filter Table Implementation Using DAGs

Filter tables are used to classify packets belonging to uncached flows. They are usually invoked only for the first packet of a flow. Nonetheless, many flows may be very short-lived (just one or a few packets), so it is important to have an efficient filter table implementation.

Several generic packet filtering algorithms have been proposed in the literature [2, 9, 19]. These algorithms are very powerful and flexible when they are used to look into arbitrary packet fields. They usually come with a 'language' which allows for the specification of filters in terms of individual bytes in the packet header, and the values they should be checked against. They are complex both in terms of theoretical background as well as in terms of code size (typically several 1000 lines of C code). To specify a simple filter to match a given TCP connection, half a page of filter specification written in the filter's language might be required (see [2] for an example of a TCP filter specification). Besides complexity, all except DPF [9] typically provide performance which is worse than that of tailor-made packet classifiers optimized for a certain fixed pattern of packet header.

Furthermore, these existing packet filtering algorithms either do not support or cannot efficiently match on partially (arbitrary number of bits) wildcarded fields, and therefore cannot be used for efficient detection of best matching prefixes on addresses. This was an important requirement in our EISR framework.

Unlike generic packet filters that are optimized to search based on arbitrary bytes (specified by the user) in a packet, our filter table implementation targets only the Internet protocol stack, and requires packets to be classified based upon the same five packet header fields and the interface on which the packet was received. Our goal was therefore to find a fast lookup algorithm for matching the six-tuple <*source address, destination address, protocol, source port, destination port, incoming interface*> in a packet against a possibly large set of filters (several of which may include address fields that are partially wildcarded, requiring a longest prefix match).

Note that since there is one filter table for every gate in our system, usually multiple lookups (in different filter tables) are necessary for each packet that is received on an uncached flow. Why is it that we don't have a single filter table that applies for all network functions? The answer is that the router administrator may have very different sets of policies for different networking components. For example, the set of filters that are specified for one function (e.g. packet scheduling for QoS) will usually be quite different from the set of filters that are installed for security applications (e.g., firewalls). While it is theoretically possible to merge all filter tables into a single global filter table (by merging the different filter specifications and creating new filters whenever there is an overlap), such an implementation is practically infeasible because the space requirements for the global table can, even with very few installed filters, increase very quickly (exponentially) to unacceptable levels.

Note that the property of requiring multiple packet classification steps (filter table lookups) is not unique to our system. Every common integrated services router does at least two filter lookups: one for packet scheduling, and one for routing. Routing in that sense is packet classification with only one field (destination address) in the six-tuple for a filter specified, and all the other fields set to wildcards. A more generalized approach to routing would involve looking not just at the destination address, but also at other fields in the packet; this kind of extended routing functionality has come to be known as L4 switching.

4.1.1 Directed Acyclic Graph (DAG) Implementation

Our implementation of filter tables makes use of a directed acyclic graph (DAG) to find the best matching filter. The easiest way to explain the algorithm is to use an example. For simplicity, our example assumes filters with only three header fields in place of six. It should be noted that this scheme can work with an arbitrary (but constant) number of filter fields.

#	Source Address	Destination Address	Protoco l
1	129.*	192.94.233.10	TCP
2	128.252.153.1	128.252.153.7	UDP
3	128.252.153.1	128.252.153.7	TCP
4	128.252.153.*	*	UDP

Table 1: Sample Filters

We consider a filter table containing four filters (see Table 1); the first field in each filter corresponds to the source address, the second field to the destination address, and the third field to the protocol. The first filter matches all TCP traffic from the network 129.0.0.0 to the host 192.94.233.10. The second and the third filters match all UDP/TCP traffic from host 128.252.153.1 to host 128.252.153.7. And the fourth filter matches all UDP traffic from network 128.252.153.0. It is easy to see that filter 2 is a proper subset of filter 4; we say that filter 2 is *more specific* than filter 4. Also note that filters 1 and 4 are *disjoint.*

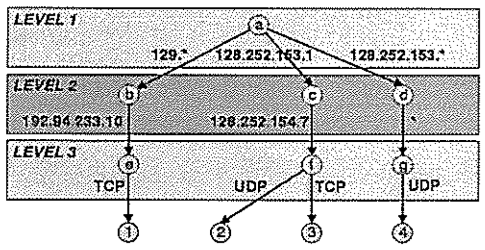

Figure 4: DAG without Soft Edges

Figure 4 shows the corresponding DAG. To match a triple <128.252.153.1, 128.252.154.7, UDP> corresponding to an incoming packet, the triple's first field, the source address of the packet (128.252.153.1) is subjected to a longest prefix match against the three prefixes present at level 1 of the DAG (i.e., 129.*, 128.252.153.1, and 128.252.153.*). The most specific match is clearly (128.252.153.1) and therefore the edge to node 'c' of the DAG is followed. Next, the second field, the packet's destination address, undergoes a similar longest prefix match against prefixes present at level 2 of the DAG on edges leading out of node 'c'. Since there is only one such prefix (128.252.154.7), and it matches our input value, the search continues to node 'f'. On the next level, the match function is a simple equality check on the protocol field from the packet. Since there is a matching outgoing edge for 'UDP', the filter lookup procedure terminates, returning filter 2 as the best matching filter.

Note that the matching function used at each level of the DAG can be different, and is based on the desired lookup method for the corresponding field type. For example, for IP address fields, a match based on the longest prefix match is appropriate. For port numbers, matching can be done on ranges, with the possibility of having the single wildcard '*'. For the protocol and incoming interface fields, an appropriate matching function would be a simple exact match (equality) with the possibility of a wildcard match ('*'). The matching function itself can be independently configured for each level of the DAG, and is implemented as a special plugin in our framework (it is special because, unlike other plugins, it cannot be bound to flows). For IP address matching, we implemented two such plugins: one is based on the slower but freely available PATRICIA algorithm, and the second is based on the patented binary search on prefix length [28] algorithm. For the other levels, we use a default plugin provided as part of our kernel, which performs the simple equality checks mentioned above.

Note that the leaf nodes of a DAG correspond to the installed filters, and therefore contain all information associated with filters. These filter records contain, in addition to a pointer to the correct plugin instance, an opaque pointer that can be filled in by the plugin to point to some private data. This can be used by plugins to store plugin specific (hard) state that is associated with installed filters.

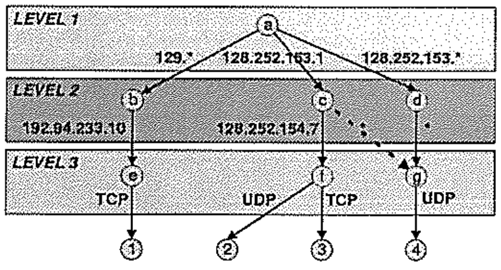

Figure 5: DAG with Soft Edges

There is one difficulty we have to deal with when building the DAG. Consider the triple <128.252.153.1, 129.0.1.1, UDP>, corresponding to an incoming packet. Following the approach outlined in the example above, the first match would lead to node 'c'. The second match operation would fail because 129.0.1.1 is neither equal to nor a subset of 128.252.154.7. However, clearly filter 4 should have matched this triple. To deal with such cases, a check must be made when building the graph for subsets; in our example, since 128.252.153.1 is a pattern matching 128.252.153.*, there must be an edge from node 'c' to node 'g' with the wildcard value '*' (the dashed line in Figure 5). In other words, whenever there is a subset relationship between filter fields corresponding to two or more edges leading out of a node, there must be edges in the graph that connect the corresponding branches on the next lower level. We call these edges 'soft' edges because they might not be permanent. It is these soft edges that result in the graph being a DAG rather than a tree. If in our example the filter <128.252.153.1, *, UDP> were to be added, the soft edge would have to be replaced by a regular (hard) edge pointing to a new node in the graph. If on the other hand filter 4 were to be removed from the filter table, the soft edge between 'c' and 'g' would have to be removed as well. Note that adding one soft edge to the graph at level *l* of the DAG may cause one or multiple adds of soft edges at level *l*+1. Our implementation efficiently finds, inserts, and removes the necessary soft edges. Due to space limitations, details of our implementation are beyond the scope of this document and will be published in a separate paper.

Although our example does not demonstrate it, each DAG will usually have a default (completely wildcarded) filter (i.e., <*, *, *, ...>) installed, which gets used if none of the other filters match. In this case, the default branches from many nodes at different levels in the DAG will usually be soft edges leading to nodes along the default path (i.e., the path with all '*'s) through the DAG.

4.1.2 Optimizing for Lookups in Multiple DAGs

So far, we showed only one DAG, which implements a single filter table. As mentioned earlier, several filter table lookups may be necessary for each packet, one at each gate that is encountered by the packet along its data path. Often, it may be the case that the same or similar filters are installed in two or more filter tables. In such cases, it should be possible to exploit the information that has been gleaned from a lookup in one filter table to speed up the lookup for the same packet in the next and subsequent filter tables.

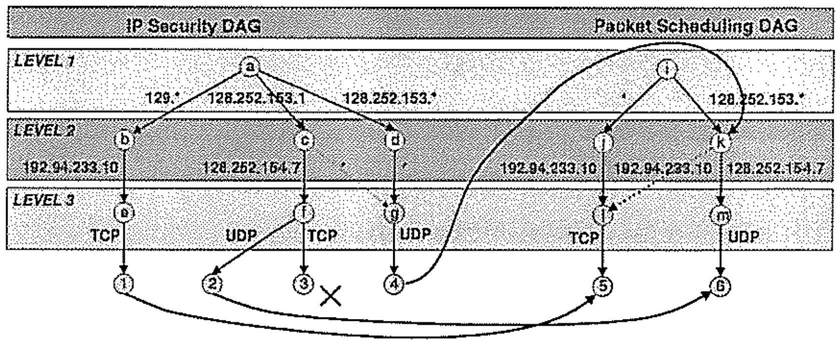

Figure 6: Two DAGs Connected

In Figure 6, we show the DAGs corresponding to two gates along the data path. The first gate corresponds to IP security; we can refer to the corresponding DAG as the *IP security DAG*. The second gate corresponds to packet scheduling; it has a corresponding *packet scheduling DAG*. Note the lines that point from leaf nodes in the first DAG to nodes in the second DAG. These lines, which are calculated whenever a filter is added or removed from either of the DAGs, are used to show where the search in the second DAG can start assuming that the result of the lookup in the first DAG is known. In the example from the figure, the result of the lookup in the second DAG is known without having to traverse that DAG if the search in the first DAG terminated at leaf nodes 1 or 2 (the result is leaf nodes 5 or 6 respectively, as indicated by the lines connecting the two DAGs). Node 3 represents a leaf node in the first DAG which does not have a corresponding node in the second, so that the search would have to begin at the root of the second DAG. Leaf node 4 in the first DAG cannot be mapped to a leaf node in the second DAG. However, the lookup in the second DAG can commence from intermediate node 'k', so fewer levels will have to be traversed. It is easy to see that by computing such interconnections between DAGs implementing the different filter tables, significant performance benefits can be realized. Note however that these performance benefits depend on the actual set of filters that have been installed in the various filter tables. The more alike the filters installed in different filter tables, the greater the gains. In the unlikely case where completely different sets of filters are installed in different filter tables, there will be no performance gains at all.

4.1.3 Other Optimizations

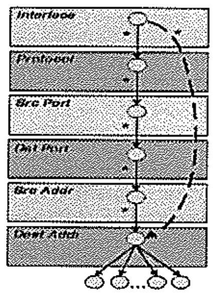

Figure 7: Collapsing Levels

Our DAG approach affords some simple optimizations. If multiple wildcarded edges succeed each other without any branching at intermediate nodes (see Figure 7[2]), then an obvious optimization is to collapse multiple nodes along such a path into a single *supernode*. If a supernode is encountered when traversing a DAG, the lookup procedure can skip over several fields in the packet, corresponding to the levels that have been collapsed. This becomes very important for DAGs that implement a Routing plugin, where most routes will be based on the destination address only; in this case, it is possible to skip over most of the fields and do a best-matching prefix on just the destination address, making the lookup as fast as can be supported with a traditional routing table approach. Of course, if there are several filters that are specified with routing based on fields other than the destination address (L4 switching), then this optimization may not provide as much benefit.

4.1.4 Ambiguous Filters

There is one important property of our algorithm that we have not mentioned so far: the filters specified in a DAG may be *ambiguous*. A pair of filters are considered to be ambiguous if an incoming packet matches both filters, but neither filter can be considered to be a better matching filter. Consider for example the two filters that have been installed for packet scheduling: <*A*, *, *, *, *, *> and <*, *B*, *, *, *, *>. The first filter matches all packets from source address *A*; let us say that packets matching this filter are given 80% of the available bandwidth. The second matches all packets with destination address *B*; assume its bandwidth allocation is 20%. Now, suppose we receive packets with source address *A* and destination address *B*. Such packets match both filters. Further, neither filter can be considered a better match, because neither filter is a subset of the other. So the question arises: what percentage of the bandwidth should the corresponding flow receive? Clearly, the answer cannot be determined without more information. One way to get this extra information would be to require the user to add another filter <*A*, *B*, *, *, *, *> with a specified bandwidth, say 80%. In this case, the packet would match all three filters, but the most specific matching filter would be the new one we just added, so there is no ambiguity. Such filters which are added to remove ambiguities are called *disambiguating filters*.

Resorting to set theory, and thinking in terms of sets in the filter tuple space, a pair of ambiguous filters correspond to a pair of sets which have a non-zero intersection, where neither set of the pair is a proper subset of the other. In this case it is not possible, given a data point that lies within the intersection of the two sets, to decide on which of the sets it belongs to. Note that if one set is completely contained within the other (i.e., a proper subset), then there is no ambiguity — we would match on the smaller set, which would correspond to the best match. If ambiguous filters are inserted in a DAG, then the result from a lookup may depend on the ordering of fields in the DAG (i.e., which field is processed at what level in the DAG). One approach would be to arrange the fields (and levels) in some known order, which would result in an implicit priority among fields

[2] The order of fields corresponding to levels in the DAG which is shown in the figure has changed from previous examples; the destination address field is now at the lowest level. This has been done to emphasize the fact that the ordering of fields is not important for the algorithm to function correctly. As we will see later however, it does impact the memory space requirements; the organization shown in the figure has the minimum space requirement if the filters are all based on the destination address alone.

that can be used to decide the outcome of an ambiguous lookup. For example, in Figure 5, the protocol field has priority over the address fields since it is at the lowest level. The default behavior that is implicit from this priority ordering can always be overridden by the user (the entity installing the filters) if necessary, by installing one or more disambiguating filters. This kind of resolution progresses by finding filters corresponding to intersections of ambiguous filters, and adding those filters to the DAG, until the resulting collection of filters are either completely disjoint, or are proper subsets of one another. In other words, there would be no overlapping filters.

Often however, it might make sense to choose an ordering of fields in the DAG so that it results in the minimum memory space usage. With large numbers of filters, the memory needs of a DAG can be quite significant, so this is an important optimization. In such cases, the ordering of fields may not be known in advance, but may be dynamically determined based on the current set of installed filters. Thus, the outcome of a lookup would be one of the matching ambiguous filters, but it may not be possible to know in advance which one. In many cases, this kind of behavior may be quite acceptable. In cases where it isn't, it would be necessary to resort to the trick of adding disambiguating filters, as described in the previous paragraph.

Our implementation includes detection of ambiguous filters, which is executed whenever a filter is added. If it is determined that the newly installed filter results in an ambiguity in the resulting DAG, this condition is reported to the user (the entity that installed the filter). In addition, our implementation also finds and reports the set of disambiguating filters that would be needed to resolve all ambiguities. The user can respond to this by either ignoring the condition (in which case he is willing to tolerate the ambiguity), or by adding the set of disambiguating filters to the already installed filters.

Our DAG-based scheme is loosely related to Cecilia Tries [27]; the DAGs can be thought of as multi-bit Cecilia Tries with header fields being used instead of bit fields, and a generic matching function (e.g., best-matching prefix) in place of exact matching of multibit fields to integer values.

The DAG-based algorithm is simple and easy to implement (our implementation requires approximately 800 lines of C code including ambiguity detection), and it is much faster than the 'typical' filter algorithms used in existing implementations [16, 21]. While most of these existing techniques require $O(n)$ time, n being the number of filters, our solution when used with a state-of-the-art best matching prefix algorithm (e.g., controlled prefix expansion [24]), is more or less independent of the number of filters. If we were to characterize the performance of our DAG approach, it would be $O(f)$, where f is the number of fields in a filter specification. Since any packet classifier has to look at least once at each field in the packet (except when the set of filters is trivial, e.g. all wildcards), we argue that our scheme is theoretically optimal in speed. From a practical standpoint, our current implementation does not exploit hardware properties such as the machine's cache subsystem architecture or main memory quirks to improve performance. Given the large improvements in performance that other researchers have achieved by exploiting such properties in the context of best matching prefix algorithms, we believe that there is room for future work in applying similar techniques to our packet classification algorithm. Also, as mentioned earlier, the memory requirements for our DAG-based filter tables can be substantial for large filter sets. We are currently working on techniques to improve the memory utilization of our algorithm. It is important to note that because of the modular character of our implementation, we can easily replace our DAG based classifier with a new classifier plugin if and when better approaches become available.

In this section, we have attempted to provide an overview of the DAG based packet classification algorithm. A description of the implementation details are beyond the scope of this paper and will be covered in a future paper. Section 6 provides some performance results from our current implementation of the DAG-based packet classifier.

4.2 Flow Table Implementation Using Hashing

The flow table is used to cache flow information for individual end-to-end flows. In other words, each entry in the flow table corresponds to a flow with a fully specified filter (one that contains no wildcards). Since there is no wildcarding, hashing can be used to implement flow table lookups efficiently.

Out implementation of the flow table uses the five tuple of header fields <*source address, destination address, protocol, source port, destination port*> from the packet to calculate the hash index. The code that is used for this calculation has been kept very simple to improve performance. It is executed in 17 processor cycles on a Pentium, and is described in Section 5. Hash collisions are resolved by storing all entries in the same hash bucket on a singly linked list.

The array for the hash table is allocated at system boot time. Its size is dependent upon the environment in which the router is used (LAN vs. regional vs. backbone router); the default value used in our kernel is 32768.

Each flow record in the hash table includes space for:

1. The six tuple of the corresponding filter
2. A pair of pointers for each gate that is implemented in the core. One pointer points to the plugin instance that has been bound to the flow. The second points to private data for that plugin instance; it is used by the plugins to store per-flow "soft" state. This is used, for example, by the DRR plugin (Section 5.1) to store a pointer to a queue of packets for each active flow.
3. A pointer to the filter record from which this flow was derived.
4. A pointer which is used to link the record onto either a free list or onto the linked list for a hash bucket.

A small number of flow records is allocated at system boot time and linked into a free list (default is 1024). More records are added as the need arises, with the number of allocated records increasing exponentially (e.g. 1024, 2048, 4096, ...) to adapt to the environment as fast as possible. The system can be configured to stop allocating new flow records after a given maximum number of records have been allocated. Once this point has been reached, the oldest flow records are recycled (i.e., the old entries in the cache are replaced with new ones). Different cache replacement policies can be used with different performance trade-offs. To keep our discussion simple, we will not go into the details of the cache replacement policy that we have implemented in our current setup.

Performance results from our flow table implementation are presented in Section 6.

5 Example of Plugins

We will look at two plugins for packet scheduling in this section to give the reader a better feel for how plugins interact with our architecture and how they are implemented. The two plugins we consider are a port of Carnegie Mellon University's (CMU) Hierarchical Fair Service Curve (H-FSC, [25]) algorithm and our own implementation of a simple weighted Deficit Round Robin (DRR, [22]) plugin. These two plugins are complementary in the sense that DRR is particularly useful to implement fair queuing among best-effort flows, whereas H-FSC implements hierarchical scheduling similar to Class Based Queuing (CBQ, [10]) with several advantages over CBQ. In the current implementation, packet scheduling plugin instances are chosen per interface. We plan to implement a Hierarchical Scheduling Framework (HSF) which will allow different instances of packet scheduling plugins to be placed at individual nodes in the scheduling hierarchy. For example, this will allow us to combine both the H-FSC and the DRR scheduling schemes, where DRR could be used to do fair queuing for all flows ending in the same H-FSC leaf node. Note that in its current implementation, H-FSC uses FIFO queueing for all flows matching the same leaf node, which may result in unfair service to different flows.

5.1 The Weighted DRR Plugin

The Deficit Round Robin (DRR, [22]) algorithm is a very simple yet powerful packet scheduling scheme which provides fair link bandwidth distribution among different flows. The original implementation comes from the WFQ module found in the ALTQ [5] software distribution. The ALTQ WFQ modules implement fair queueing for a limited number of flows, which it distributes over a fixed number of queues. ALTQ came with a basic packet classifier which mapped flows to these queues by hashing on fields in the packet header. Since our architecture already offers mechanisms to store per-flow information in the flow table records, it was straightforward to add a queue per flow which guarantees perfectly fair queuing for all flows. In order to allow bandwidth reservations, we have implemented a weighted form of DRR which assigns weights to queues. These weights are fixed for all best effort flows and dynamically recalculated for reserved flows if a new reserved flow is added to the system. Since packet classification is already done very efficiently by the AIU, the actual scheduler plugin is very simple (less than 600 lines of C code). It turned out to be extremely useful for demonstrations of the link-sharing capabilities of our architecture.

Shown below are the commands necessary to load and configure the DRR plugin; this will give the reader a feel for the simplicity and elegance with which plugins can be put into operation. Note that these commands can be executed at any time, even when network traffic is transiting through the system. *pmgr* is our Plugin Manager program, and *modload* is the NetBSD command that is used to load kernel modules.

- Loading the plugin: the plugin registers with the PCU under the name 'DRR'.

```
wooster# modload -o drr -e drr combined.o
Module loaded as ID 0
```

- Creating an instance: this creates an instance of plugin named 'drr' of type 'ps' (packet scheduler), and binds it to the ATM interface en0.

```
wooster# ./pmgr create_instance pn=drr pt=ps if=en0
Created plugin instance, handle = 1
```

- Adding a filter: this specifies a filter which matches all traffic originating at IPv6 source address `3ffe:2000:400:11::4`, and sets the reserved bandwidth for all flows matching this filter to 80%. Usually, the filter and it's associated QoS would be set by a daemon (RSVP or SSP) through a library call, but we implemented it on the command line as well to allow for simple testing.

```
wooster# ./pmgr add_filter sa=3ffe:2000:400:11::4 pt=ps ih=1 bw=80
Filter Handle = 3
```

From now on, all flows originating from the specified source address will get at least 80% of the link bandwidth. Note that the packet scheduler can be turned off any time by freeing the instance or unloading the plugin module (which frees all instances of the plugin automatically):

- Freeing an instance: frees the instance of the plugin named 'drr', type 'ps' (packet scheduling) with handle '1'.

```
wooster# ./pmgr free_instance pn=drr pt=ps ih=1
```

We demonstrated the DRR packet scheduling plugin by sending a video stream on a 155 Mb/s ATM link through a router that implemented our framework. We created UDP streams to generate noise, with the objective of disrupting the video. As expected, we observed very good quality video when using bandwidth reservations for the video stream, and a significant degradation of the quality without reservations. In case of a reserved video flow, the reserved flow remained within approximately 1% of it's reserved bandwidth, regardless of the number of active UDP streams.

5.2 The Hierarchical Fair Service Curve Algorithm

Our implementation of the H-FSC algorithm is a port of an implementation provided by CMU. Since the developers at CMU properly separated the packet classifier from the scheduling algorithm, it was relatively easy to convert the code to use our DAG-based packet classifier which is resident in the AIU. Besides that, we converted it into a dynamically loadable module and changed the interface to work with our environment. The algorithm is well documented in [25] and our results are consistent with that paper. We believe that H-FSC represents the state-of-the-art in packet scheduling. One of its main advantages is the decoupling of delay and bandwidth allocation, which is very useful if both real-time and hierarchical link-sharing services are required concurrently. There are just two potential disadvantages of this algorithm. The first is its complexity and therefore its overall throughput performance. As we show in section 6, the simple DRR plugin requires less processing overhead than H-FSC. Second, we plan to improve H-FSC to provide fair queuing among all flows of the same leaf traffic class by adding our DRR plugin to the leaf nodes of the scheduling hierarchy. This will be very simple once our Hierarchical Scheduling Framework (HSF) is in place.

6 Performance

In this section, we elaborate on the performance of our architecture and implementation. One of the drawbacks of modularity is that modules are accessed using indirect function calls. We show, however, that the indirect function calls do not significantly affect the overall performance of the system. We also look into the performance of the hashing algorithm which is especially critical since it is touched by every single packet. Thanks to the careful implementation of the hash function, packet to flow mapping based on hashing requires 1.3 μs for an IPv6 flow if the relation between the number of flows and the size of the hashtable is reasonable. We show performance of our DAG-based filter lookup algorithm by giving a worst case estimate for the number of memory accesses which is 24 for IPv6. We conclude with measurements of the overall packet forwarding delay and throughput supported by our kernel. We found that even in a highly modular environment we add only 8% overhead compared to a best-effort kernel. We provide equal or better performance than most integrated services platforms.

We did all of our measurements on a Pentium Pro with 512 KB Level 2 cache running at 233 MHz. For our measurements, we used the VTUNE [13] tool to obtain dynamic clock cycle counts. Further we used special functions to access the Pentium's processor clock register (tsc) which is incremented by one every cycle and allows for very accurate measurements. The Pentium processor provides two instruction pipelines to execute instructions in parallel if no explicit dependencies exist among instructions which prevent parallelization. Since the number of cycles consumed by a single instruction therefore depends on its context, the cycles indicated in this section represent worst case values.

6.1 Indirect Function Calls

A regular function call (hard coded, not indirect through a function pointer variable) requires 1 cycle and can usually be executed in parallel with the previous instruction. An indirect function call through a variable requires 3 cycles, and since an explicit flow dependency occurs, no parallel execution is possible. In case the function's address has to be fetched out of the second level cache or main memory, wait states are necessary. For a RAM with 60ns memory access time, roughly 15 cycles are spent for every access that cannot be satisfied from the first (L1) or second level (L2) caches. In case of a hit in the L2 cache, 8 cycles are expended for the address fetch. Since all function pointers come from entries in the flow table which have been accessed just before the call to the function, it is fair to assume that the pointer would come out of L2 cache in the worst case. Consequently, an indirect function call would require 12 cycles or 52 ns. This overhead has to be multiplied by the number of gates throughout the kernel. In a typical scenario, with one gate each for IP security, IP options, Routing, and Packet scheduling, we would require roughly 200 ns in the worst case to call all of the plugin instances. These back-of-the-envelop calculations give us an idea of the overhead associated with modularization. We provide actual measurement results in the following subsections.

6.2 Flow Detection: Hashing

Flow table lookups are a key function in our architecture as they are executed for every packet. We use the following function to calculate the hash key.

```
if(flow_label) {
    key = flow_label & (TABLESIZE-1);
} else {
    key = BSWAP(LOWEST_32BITS(source address) +
        LOWEST_32BITS(destination address)) +
        BSWAP(SRC_PORT+DST_PORT) & (TABLESIZE-1);
}
```

The same code applies to IPv4 and to IPv6. `flow_label` is the IPv6 flow label, and it gets used if it is non zero. In case of IPv4, the `LOWEST_32BITS` macro simply returns the address itself, while for IPv6 it fetches the least significant 32 bits of the IPv6 address.

BSWAP executes the Pentium's *bswap* instruction. We need to execute a byte swapping function on little endian machines because the data is in network byte order (big endian) and we would not get even hash key distribution without it. We are using *bswap* instead of the NetBSD function *ntohl* because it turned out to be significantly faster. *ntohl* executes a sequence of *rotate* instructions which consume a total of 7 cycles, whereas *bswap* requires a single cycle. We use an AND function (requires 1 cycle) instead of the modulo function usually applied for hashing (which requires 41 cycles). The complete calculation requires 17 cycles or 73 ns, and turned out to result in a reasonably even distribution of individual flows over the hash table. We simulated hashing in user mode to get a better idea of how well this algorithm would perform. From [15] we found that a large backbone router (FIXWEST2) has to manage an average of 22000 active flows concurrently. We set the hash table size to 32768 and used larger numbers of flows to show how hash table overload affected performance. We passed 50 million IPv6 packets through the flow lookup in the AIU, and measured hash table lookup times. Note that IPv6 flow labels were not used for our measurements. The results are shown in Table 2. A flow lookup requires about 1.3μs under regular circumstances without overload. 200% overload causes the lookup time to degrade to 1.7μs which is still acceptable.

# of Flows	Size of hash table	hash table load	collisions	average # cycles for flow lookup	average flow lookup time
22000	32768	67%	0%	310	1.3μs
44000	32768	134%	25%	350	1.5μs
65535	32768	200%	50%	400	1.7μs
131072	32768	400%	150%	580	2.5μs

Table 2: Flow Lookup

6.3 Packet Classification

The DAG scheme uses a best matching prefix (BMP) algorithm for address lookups, and simple indexing for port numbers and the protocol field. Recently, several new BMP algorithms have been proposed to replace the PATRICIA [23] algorithm found in many of today's BSD-based routing engines: Binary search on prefix lengths (BSPL, [25]), Multiway and Multicolumn search (MMS, [14]), and Controlled Prefix Expansion (CPE, [24]), to name just a few. Most of these schemes are optimized for one lookup table which they usually try to fit into the processor's cache. Performance measurements published in these papers cannot directly be applied to our scenario, because we have a potentially large number of smaller lookup tables, one per pertinent node of the DAG. Furthermore, performance of these schemes largely varies with the type and size of the working data set. Such trace-driven simulation cannot be applied to our framework because appropriate data sets of real-world filter patterns are not available. However, the metric for the worst case number of memory accesses of the BMP algorithms is an interesting measure since it would allow us to give a good worst case estimate of how the classification algorithm performs. Table 3 gives

Scheme	Prefix length	Worst case memory accesses
PATRICIA	32	32
	128	128
BSPL	32	5
	128	7
CPE	32	4
	128	5

Table 3: Worst Case Memory Access Time for Different BMP Schemes

an overview of the worst case number of memory accesses for the three different schemes for 32 bit (IPv4 addresses) and 128 bit (IPv6 addresses) prefix lengths. These numbers are valid for a very large number of prefixes, usually in the order of 50000. So far, we have implemented PATRICIA and BSPL as plugins, and plan to add at least CPE since it shows very interesting numbers. In case of BSPL, the number of worst case memory accesses for a full filter lookup calculation is shown in Table 4. Since the operations to calculate the hash values are inexpensive compared to memory accesses, a reasonably good estimate of the worst case filter lookup time can be calculated by multiplying the number of memory accesses with the memory access delay (60 ns). This leads to a worst case filter lookup time of 1.4 us and has to be multiplied by the total number of gates in use to get a worst case estimate of the total lookup time of the packet. Again, since this is a worst case

Access to function pointer for BMP function	1
Access to function pointer for index hash	1
IP address lookup (2*$\log_2$(32)/2*$\log_2$(128))	10/14
Port number lookup	2
Access to DAG edges	6
Total	20/24

Table 4: Memory Accesses for a Filter Lookup

number, we expect much better results in real world scenarios where the number of filters is typically much smaller, and we could benefit from various optimizations in the DAG data structures, as shown in section 4. In any case it is important to note that this number is independent of the number of filters in use and how they are organized. While we expect faster schemes for filter lookups to emerge in the future which exploit hardware properties in the same way BMP algorithms do, we believe that the DAG-based scheme already offers a significant improvement over current implementations of packet classifiers, which typically traverse a linear list of all filters. We are currently working on more detailed performance evaluations in terms of processor cycles for different filter sets and plan to publish them as soon as they become available.

6.4 Overall Packet Processing Time

Overall throughput was measured using the Pentium's cycle counter. We added a time stamp function into the ATM device driver which timestamped every incoming packet just after the data was received from the network card. This value was compared to the CPU cycle counter right before the packet was output to the hardware of the ATM card again. We sent 8 KByte UDP/IPv6 datagrams (IPv6 flow label NOT used) belonging to three different flows concurrently through our router. The ATM MTU was 9180, so there was no fragmentation. We sent a total of 100 packets per flow, and calculated the average processing time. This was repeated 1000 times. The system had 16 filters installed. We installed three gates which called empty plugins for the first test and only one gate for packet scheduling in case DRR was turned on. The results are shown in Table 5 The

Kernel	Average Cycles	Average Time [us]	Additional overhead [us]	Relative Overhead	Throughput [packets/s]	Throughput [Gb/s][a]
Unmodified NetBSD 1.2.1	6460	27.73	-	-	36800	2.3
NetBSD with our Plugin Architecture	6970	29.91	2.19	7.89%	34100	2.1
NetBSD with ALTQ and DRR	8160	35.0	-	-	28600	1.9
NetBSD with our Plugin Architecture and a DRR plugin	8110	34.8	(0.2)	(0.61%)	28729	1.9

Table 5: Overall Packet Processing Time

a. The throughput numbers reported in this table do not take into account the data copy overhead between memory and the network interface.

first row shows the processing time of the unmodified NetBSD 1.2.1 kernel. A packet is received, forwarded and sent back to the ATM hardware within 6460 cycles or 28 μs. With our framework turned on, flow detection and the three function calls caused an overhead of roughly 500 cycles or 2.2 μs as expected based on our simple calculations from Section 6.1. Note that filtering has a minor impact on the overall throughput since it happens only for the first packet of each flow. With our DRR plugin installed and guaranteeing fair queueing among the three flows, we measured similar performance as an ALTQ system running the same algorithm. Since the packet scheduling code is similar in both implementations (our implementation of DRR is derived from ALTQ), we benefit only from faster hashing in terms of performance. Packet scheduling introduces an overhead of 20% compared to a best-effort kernel. While 20% overhead may sound excessive, it corresponds to the numbers reported by others. Although H-FSC has very different scheduling characteristics from DRR, thereby making any direct comparison difficult, [25] reports between 6.8 and 10.3 μs[3] for packet queueing overhead, which would correspond to about 25% to 37% overhead.

It is important to see that every integrated services platform requires some sort of packet classification. By carefully implementing packet classification, we achieve faster lookups for IPv6 than other integrated services platforms for IPv4 (e.g, [25] states that they require 2.6 μs for packet classification for IPv4 packets), even though IPv6 addresses are larger. Once the flow a packet belongs to is detected, picking the right instance of a plugin to which the packet should be passed does not cost more than an indirect function call. Thus we showed that on integrated services platforms, a very flexible and modular architecture can be introduced with almost no additional processing cost.

[3] Stoica, Zhang, and Ng's measurements on a Pentium 200 were scaled to our 233 MHz Pentium.

7 Related Work

In this section, we review work (one commercial, three research) which is closely related to ours.

Microsoft's Routing and Remote Access Service for Windows NT (previously known as "Steelhead") is an add-on to Windows NT 4.0 or 5.0 server. It allows regular Windows NT PC's to be used as IP or IPX routers and is targeted to routing small to midsized networks. It's main advantage is price and ease of use, since it can benefit from NT's graphical user interface for configuration and monitoring tasks. It implements most of the important routing protocols like RIP and OSPF as well as support for packet filtering and remote network management via RPC. It provides one API to implement custom routing protocols and another API to extend management support. Both APIs are interfaces to user-space components and do not allow the extension of the networking subsystem in the kernel. To add new network protocol implementations like IPv6, the usual TDI/NDIS interfaces have to be used and the entire protocol stack has to be implemented as a block. The system does not allow for replacement of individual functional modules like routing lookups or packet schedulers. Further, it does not implement any integrated services components (yet).

ALTQ [5] is an attempt to unify the interfaces to packet schedulers. It comes with a port of an implementation of Class Based Queuing (CBQ, [10]) and a DRR module which we changed to provide support for reserved bandwidth. Further, it provides modifications to the NetBSD ATM driver to support packet scheduling. While ALTQ bundles interesting components, it does not implement a lot of functionality on it's own. In particular, it does not separately implement a packet classifier but uses packet classifiers that are glued to the various packet schedulers (which, as we mentioned earlier, is the wrong place for doing packet classification). It does not allow dynamic loading of modules nor does it feature any functionality to bind code to flows.

The x-kernel [11] is a well known object-based framework for implementing network protocols. It defines an interface that protocols use to invoke operations on one another and a collection of libraries for manipulating messages and operating system resources. Over the last several years, the x-kernel has served as a research platform for investigating end-to-end issues related to computer networks. The x-kernel focuses on the implementation of network software running on the end systems, contrary to our work which concentrates on the router.

The same university released Scout [19], which is an operating system implementation based upon x-kernel, with a special focus on network appliances. Still mainly end-system oriented, it comes with a router module which implements QoS functionality and IPv6. Various packet scheduling schemes like DRR, Virtual Clock [26] and Weighted Fair Queuing [8] are included in the router module. The problem with implementing a new operating system seems to be that it requires an enormous amount of time and manpower to provide features comparable to matured implementations like BSD Unix. Whereas we rely on state-of-the-art IPv6 and packet scheduler components implemented by others, and focus on implementing our specific new contributions, everything had to be reimplemented for Scout. Thus, some of the provided components lack several important features. For example, the IPv6 module does not implement ICMPv6, and routing lookups are done in a strongly simplified fashion using hashing. At this point of time, Scout does not appear to provide the feature set of state-of-the-art implementations, such as those available for NetBSD and used in our system.

8 Conclusions and Future Work

We presented an extensible and modular software framework to implement high-performance integrated services routers. It allows code modules called plugins to be dynamically loaded into the kernel and configured at run time. Instances of plugins can be bound to individual flows. We have implemented a fast packet classification algorithm which makes use of existing highly optimized best-matching prefix algorithms, and which provides acceptable worst-case filter lookup times that are independent of the number of installed filters. We clearly separated packet classification from packet scheduling, and showed that a high degree of modularity can be incorporated in an integrated services platform without considerably affecting performance. We plan to freely distribute our source code, with the objective of providing the research community with a state-of-the-art integrated services platform to build upon. We expect to release this software in the public domain by the time this paper is published.

Our future plans include implementing the Hierarchical Scheduling Framework (HSF) to provide a more sophisticated environment for packet scheduling than what we've presented so far. The HSF will allow the combination of different scheduling algorithms at different levels in the scheduling hierarchy. As one application, we plan to show the combination of H-FSC and fair queuing algorithms like DRR, where DRR would do fair queuing among the flows in a H-FSC leaf node. We believe that the integration of routing into the packet classifier makes a lot of sense. While this is conceptually very simple, it requires some amount of work to do this in a standard BSD Unix kernel, since the routing functions are not very well isolated. By unifying routing and packet classification, we get QoS-based routing/Level 4 switching for free. We believe that these enhanced routing technologies have interesting properties and a lot of potential. The integration of routing will make fast packet classification schemes even more important. While we believe that our DAG algorithm is a valid contribution to the state-of-the-art, we plan to pursue research in packet classification algorithms, and incorporate enhanced implementations and algorithms into our framework.

9 Acknowledgments

We would like to acknowledge the help of Marcel Waldvogel (ETH Zurich) for his invaluable contributions to our design effort. The current organizational architecture of our router plugin framework owes a lot to his input in the early stages of our design. He also contributed the code for the patented "Binary Search on Prefix Lengths" (BSPL) algorithm, which we use as an example plugin for fast IP address prefix matching.

Our thanks also go to Ron Cytron (Washington University, St. Louis), whom we approached (as a compiler expert) for insight into possible solutions to the packet classification problem. His feedback was crucial to our development of the DAG-based classification algorithm.

We would also like to acknowledge the help of Fred Kuhns, Hari Adiseshu, and John Dehart (all from Washington University, St. Louis); they were all involved in the project at various stages of its development, and their comments and criticisms were important to success of this project. In particular, we would like to thank Fred, who was involved in writing portions of the code for the Plugin Manager; and Hari, who contributed the code for the SSP daemon. Finally, we would also like to thank George Varghese, who always made himself available to answer questions, and be generally supportive and helpful.

References

[1] Adiseshu, H., and Parulkar, G., "SSP: A State Setup Protocol", to be published

[2] Bailey, M., et al., "Pathfinder: A pattern-based packet classifier", In *Proceedings of the First Symposium on Operating Systems Design and Implementation*, November 1994

[3] Bennett, J.C.R. and Zhang, H., "Hierarchical Packet Fair Queueing Algorithms", In *Proceedings of SIGCOMM'96*, August 1996.

[4] Bennett, J.C.R., and Zhang, H., "WF2Q: Worst-case Fair Weighted Fair Queueing", In *Proceedings of INFOCOM'96*, March 1996

[5] Cho, K., "A Framework for Alternate Queueing: Towards Traffic Management by PC-UNIX Based Routers.", To appear in *Proceedings of USENIX 1998*, June 1998

[6] Cisco Corporation, web pages on IOS, http://www.cisco.com/public/sw-center/sw-ios.shtml

[7] Deering, S., Hinden, R., "Internet Protocol, Version 6 (IPv6), Specification", RFC 1883, December 1995

[8] Demers, Keshav, Shenker, "Analysis and Simulation of a Fair Queueing Algorithm", In *Proceedings of SIGCOMM'89*, August 1989

[9] Engler, D., Kaashoek, M., "DPF: Fast, Flexible Message Demultiplexing using Dynamic Code Generation", In *Proceedings of SIGCOMM'96*, August 1996

[10] Floyd, S., Jacobson, V., "Link-sharing and Resource Management Models for Packet Networks", In *IEEE/ACM Transactions on Networking, Vol. 3 No. 4*, August 1995

[11] Hutchinson, C., Peterson, P., "The x-Kernel: An architecture for implementing network protocols", *IEEE Transactions on Software Engineering*, January 1991

[12] INRIA ftp site for IPv6 source code. ftp://ftp.inria.fr/network/ipv6

[13] Intel Corporation, web pages on VTUNE, http://developer.intel.com/design/perftool/vtune/index.htm, 1997

[14] Lampson, B., Srinivasan, V., Varghese, G., "IP Lookups using Multiway and Multicolumn Search", In *Proceedings of INFOCOM'98*, April 1998

[15] Lin, S., McKeown, N., "A Simulation Study of IP Switching", In *Proceedings of SIGCOMM'97*, September 1997

[16] Linux kernel packet filter implementation, http://wafu.netgate.net/linux/index.html

[17] Microsoft Corporation, "Update to Routing and Remote Access Service for Windows NT Server 4.0", Review and Evaluation Guide, March 1997

[18] Microsoft Corporation, web pages on RRAS SDK,
http://premium.microsoft.com/msdn/library/sdkdoc/pdnds/remacces_8085.htm

[19] Mogul, J.C., Rashid, R.F., Accetta, M.J., "The packet filter: An efficient mechanism for user-level network code", In *Proceedings of the Eleventh ACM Symposium on Operating Systems Principles*, November 1987

[20] Mosberger, D., "Scout: A Path-based Operating System", *PhD Dissertation*, Department of Computer Science, University of Arizona, July 1997

[21] Reed, D., "IP Filter", http://www.cyber.com.au/users/darrenr/

[22] Shreedar, M., Varghese, G., "Efficient Fair Queueing using Deficit Round Robin", In *Proceedings of SIGCOMM '95*, August 1995

[23] Sklower, K., "A tree-based routing table for Berkeley Unix", *Technical report*, University of California, Berkley, 1993

[24] Srinivasan, V., Varghese, G., "Faster IP Lookups using Controlled Prefix Expansion", *Submitted to SIGMETRICS'98*

[25] Stoica, I., Zhang, H., Ng, T.S.E., "A Hierarchical Fair Service Curve Algorithm for Link-Sharing, Real-Time and Priority Services", In *Proceedings of SIGCOMM'97*, September 1997

[26] Suri, S., Varghese, G., Chandranmenon, G., "Leap Forward Virtual Clock", In *Proceedings of INFOCOM'97*, April 1997

[27] Tsuchiya, P., "A Search Algorithm for Table Entries with Non-contiguous Wildcarding", unpublished paper, 1992

[28] Waldvogel, M., et al., "Scalable High Speed IP Routing Lookups", In *Proceedings of SIGCOMM'97*, September 1997

[29] Zhang, L, et al., "RSVP: A New Resource ReSerVation Protocol", In *IEEE Network Magazine, Vol. 7, No. 5.*, September 1993