Router Plugins
A Software Architecture for Next Generation 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

1. 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/2.1ms per packet when running on a P6/233.

1.1 Keywords

High performance integrated services routing, modular router architecture, router plugins

2. 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:

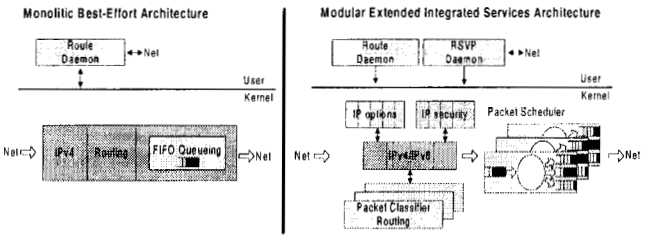

Figure 1. : Best Effort vs Extended Integrated Services Router (EISR)

- Integrated/differentiated Services
- Enhanced routing functionality (level 3 and level 4 routing and switching, QoS routing, multicast)
- 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 [8])

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.

In this paper, we propose a software architecture 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*[1].

- **Extensibility**: New plugins can be dynamically loaded at run time.
- **Flexibility**: Instances of plugins can be created, configured, and *bound to specific flows*. Plugins can be all-software modules, or they can be software drivers for specialized custom hardware.
- **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.

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 [13] and packet schedulers [27, 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 [11]). 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.

[1] 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.

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, [23]) for fair queuing, and the Hierarchical Fair Service Curves (H-FSC, [27]) scheduler for class-based packet scheduling; Implementation of plugins for IP security [2].

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" [18, 19]) 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 [12, 20, 21]. 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. We discuss these related approaches in more detail in [7].

In Section 3, we describe our architecture and explain how it achieves modularity, extensibility, and flexibility while maintaining high-performance. In Section 4, we describe the implementation of a module called the Plugin Control Unit (PCU), which is responsible for all control path interactions with plugins. Section 5 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 6, we elaborate on example plugins (packet schedulers) which we implemented or adapted for our environment. Section 7 presents performance results from our implementation, and Section 8 summarizes our ideas.

3. 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 which 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 some 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 5.
- Easy integration with custom hardware for high performance processing of specialized tasks. This is enabled by plugins which are software drivers for hardware that implements the desired functionality. For example, a plugin could control hardware engines for tasks such as packet classification or encryption.

In order to describe our framework, we first look at the different components and how they interact in the control path. In the Section 3.2, we will look at the data path, and how individual packets are processed by our architecture.

3.1 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:

- **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.

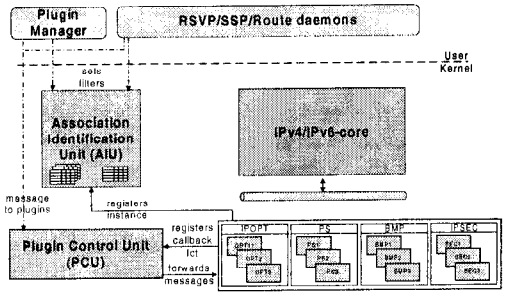

Figure 2. : System Architecture and Control Path

- **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 plugin monitoring TCP congestion backoff behaviour, 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 6.
- **Daemons:** The RSVP [31], 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.

3.2 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

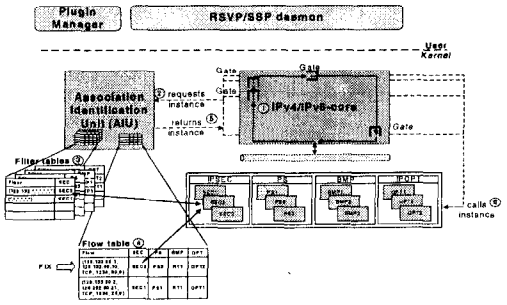

Figure 3. : System Architecture and Data Path

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, IP security processing has to be done if the system is configured as entry point into a virtual private network. In our system, IP security 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 IPv6 security 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.

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 5.1 describes a very fast filter table lookup implementation based on directed acyclic graphs (DAGs). Section 5.2 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 IP security processing will be handled. The task of this gate is to find the plugin instance which is responsible for applying security processing (authentication and/or encryption) to 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 security 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 IP security gate. The resulting plugin instance pointer is returned to the calling gate ("SEC2" 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 SEC2 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

DOCKET ALARM

Explore Litigation Insights

Docket Alarm provides insights to develop a more informed litigation strategy and the peace of mind of knowing you're on top of things.

Real-Time Litigation Alerts

Keep your litigation team up-to-date with **real-time alerts** and advanced team management tools built for the enterprise, all while greatly reducing PACER spend.

Our comprehensive service means we can handle Federal, State, and Administrative courts across the country.

Advanced Docket Research

With over 230 million records, Docket Alarm's cloud-native docket research platform finds what other services can't. Coverage includes Federal, State, plus PTAB, TTAB, ITC and NLRB decisions, all in one place.

Identify arguments that have been successful in the past with full text, pinpoint searching. Link to case law cited within any court document via Fastcase.

Analytics At Your Fingertips

Learn what happened the last time a particular judge, opposing counsel or company faced cases similar to yours.

Advanced out-of-the-box PTAB and TTAB analytics are always at your fingertips.

API

Docket Alarm offers a powerful API (application programming interface) to developers that want to integrate case filings into their apps.

LAW FIRMS
Build custom dashboards for your attorneys and clients with live data direct from the court.

Automate many repetitive legal tasks like conflict checks, document management, and marketing.

FINANCIAL INSTITUTIONS
Litigation and bankruptcy checks for companies and debtors.

E-DISCOVERY AND LEGAL VENDORS
Sync your system to PACER to automate legal marketing.