Rapid Service Creation using the JUNOS SDK

James Kelly
Juniper Networks, Inc.
jamesk@juniper.net

Wladimir Araujo
Juniper Networks, Inc.
waraujo@juniper.net

Kallol Banerjee
Juniper Networks, Inc.
kallolb@juniper.net

ABSTRACT

The creation of services on IP networks is a lengthy process. The development time is further increased if this involves the equipment manufacturer adding third-party technology in their product. In this work we describe how the JUNOS SDK (part of Juniper Networks Partner Solution Development Platform) facilitates innovation and can be used to considerably shorten the development cycle for the creation of services based on embedding third-party software into Juniper Networks routers. We describe how the JUNOS SDK exposes programmatic interfaces to enable packet manipulation by third-party software and how it can be used as a common platform for deploying unique services through the combination of multiple components from multiple parties.

Categories and Subject Descriptors

C.2.4 [**Computer-Communication Networks**]: Distributed Systems—*Network operating systems*; C.2.6 [**Computer-Communication Networks**]: Internetworking—*routers*; C.3 [**Special-purpose and Application-based Systems**]: *Real-time and embedded systems*

General Terms

Design

Keywords

JUNOS, Internet Protocol, Rapid Application Development, Network Services, Programmable Routers, Network Operating System

1. INTRODUCTION

Traffic manipulation or monitoring services are deployed on Internet Protocol (IP) networks by either introducing a specialized device on the path of the traffic or incorporating specialized functionality on devices already present on the network. Generally, management and ownership costs increase with the number and diversity of devices on a network.

Customarily routers have been closed devices. Incorporating new software-based features has been a process controlled by the equipment manufacturer since it required access to the code base related to the device in question. Granting access to third parties to modify such code base is a complex business proposition. The costs and time involved in such a process can make integration of third-party technologies prohibitive, leaving the option of using appliances as a frequent compromise. A means to simplify the process of embedding third-party software on routers is, therefore, very desirable.

A technology to solve this problem must address several key issues:

- Rapid and simple development or porting
- Interaction with the platform and the environment
- Support for interacting components from several providers
- Security, reliability, and availability

Providers expect a flexible and uncomplicated set of interfaces to interact with the platform's environment, along with a simple deployment model. The solution should allow technologies from multiple third-party providers to coexist and even collaborate. More importantly, technologies from different providers and from the device manufacturer should be able to interact.

A router deployed on a network is expected to remain reliable even in the presence of failures of the third-party software. Providers expect to be able to leverage the high-availability features of these devices to improve availability of their software despite any failure of their own component or the platform.

We describe the JUNOS SDK, a Juniper Networks solution for embedding third-party technology on JUNOS Software [2], and how it satisfies the requirements stated above. The JUNOS SDK provides APIs to allow software to plug-in to the JUNOS Software infrastructure and dynamically control device behavior. The JUNOS SDK is available as part of the Juniper Networks Partner Solution Development Platform (PSDP) [5]. The following section describes the high-level architecture of a router running the JUNOS Software operating system. Section 3 describes the JUNOS SDK, and how it can be used to create services. Section 4 describes two simple case studies of how to build services with the JUNOS SDK. Finally, the last section summarizes the contributions of the JUNOS SDK to service creation.

2. BACKGROUND ON JUNOS

In this section we present the JUNOS Software architecture at a high level and with further detail where it pertains to understanding the contributions of the JUNOS SDK. JUNOS Software is a single network operating system integrating routing, switching, and security. Most Juniper Networks hardware platforms run JUNOS

PRESTO'09, August 21, 2009, Barcelona, Spain.

Software (herein JUNOS), and many of these platforms support the JUNOS SDK. In JUNOS there is a fundamental division into three elements: the control plane, the data plane, and the services plane.

2.1 Control, Data, and Services Planes

The *control plane's* role is to manage and control the behavior of the device including the other two planes. The control plane in a JUNOS-based router runs on hardware called a Routing Engine (RE). There is often an option for a redundant backup RE.

The RE's most key element is the JUNOS Software operating system. The basis of the JUNOS kernel comes from the FreeBSD UNIX operating system [1], an open-source software system. This mature, general-purpose system provides many of the essential basic functions of an operating system, such as the scheduling of resources. To transform it into a network operating system, it has been extensively modified and hardened for the specialized requirements of networking. The task of managing the router is shared among the JUNOS kernel, many JUNOS daemons, and some ephemeral utility-style applications launched on demand.

JUNOS can be controlled by any of several user interfaces, but always in either an *operational* or *configuration* mode. As an operator issues operational or configuration commands to control the device, they first flow through the JUNOS management daemon. It directs operational requests and manages changes to the configuration database. Commands issued in the operational mode upgrade software, trigger events, or show status and statistics to inspect the system's operation. The configuration database is persistent across reboots. Configuration changes, made in configuration mode, are propagated to any daemons that subscribe to be notified about the given area of change in the database. The daemons have read-only access to the database so that they can detect changes, and change their behavior accordingly. The management daemon may also cause other daemons to be started or stopped because of additions or deletions of configuration.

The database governs the enduring behavior of the system (mostly the daemons) between configuration changes. In addition to manual changes made to the configuration, changes in behavior can naturally arise due to outside communication or inter-process communication with another process or the kernel. In JUNOS, the control plane model stipulates internal behavior changes are consistent with the configuration database.

On the control plane, the JUNOS kernel and many daemons expose interfaces so that other processes can dynamically and programmatically manipulate their states and make use of their services. The routing protocol and dynamic firewall daemons both expose such interfaces and eventually control a large part of the data plane of the router. The routing protocol daemon (rpd) manages the routes that eventually form the forwarding table. The dynamic firewall daemon (dfwd) controls stateless packet filters and rate limiters that are applied to the router's network interfaces (i.e. ports) on ingress or egress.

The *data plane's* role is to forward traffic according to the forwarding table and firewall filters. Firewall filters, managed through dwfd, filter packets based on layer-2 through layer-4 protocol headers, and they may discard, redirect, count, log, rate limit, or change quality-of-service (QoS) parameters. The JUNOS data plane spans many aspects of the chassis and its modules. It is collectively referred to and abstracted as the Packet Forwarding Engine (PFE), and comprised of ASIC-based hardware and software microcode to perform packet processing. The PFE's extended abilities include rate limiting, shaping, and other QoS functions, all of which are controlled on the control plane. Aiming to perform at fast wire speeds and within its hardware resource limits, the PFE generally defers stateful packet processing to the services plane.

The *services plane* can be thought of as an optional extension to the data plane to perform stateful services or any services non-native to the PFE. An example of a JUNOS service is a stateful firewall. The services plane runs on optionally installable and hot swappable hardware, which we generically name MultiServices (MS) modules [4, 3] herein. These connect to the data plane at speeds up to 10Gbps. The services plane is the collection of all MS modules in a chassis, and a given service can be deployed on more than one MS module.

Each MS module runs JUNOS Software with real-time processing capabilities. Yet while the kernel is basically the same one that is present on an RE, there are far fewer and different daemons running here. Also, the hardware resources differ greatly from those on the RE. Each MS module has a multiprocessing, multithreaded CPU and usually more memory than an RE, so that the software-based services here can process packets in parallel and maintain a large amount of state.

2.2 Managing the Services Plane

The control plane works with the services plane in three key ways.

First the service software must be installed on MS modules targeted for servicing. Based on configuration, a process on the RE pushes the software on to each MS module before it can be run, since the modules have no disk. This could involve installing different software on different MS modules.

Second, once the service software starts on an MS module, it needs to be configured with policies to administer, and it may want to report back information through the control plane's user interface. In JUNOS, the model used to achieve this is to have a central *management component* (daemon) on the control plane for each type of service. It handles control-plane-specific functionality such as loading configuration changes, and can communicate that information (e.g. policies) to the corresponding service application running on an MS module. Communication can flow in the other direction as well if the service application, for example, wanted to send statistics for eventual display in response to an operational command.

Third, the management component controls the data-plane to steer packets to the services plane for servicing on an MS module. We examine three of many approaches to doing this.

Service routes are the simplest steering approach. A service route is a route like any other, containing a prefix against which to match and a next hop; but the next hop is an MS module, as opposed to an external address. Once this route is installed in the forwarding table, packets matching it are redirected to the MS module.

The second option is to create and apply *service sets* to network interfaces on ingress or egress. A service set captures one or more services' policies to be applied and an MS module to which to redirect packets for servicing. Finally, when the service set is applied to an interface, the data plane marks packets flowing through that interface as part of the service set, and accordingly steers them to the specified MS module for servicing. The interface's traffic is filterable so that only certain packets are redirected to the service set. This is based on any combination of match conditions offered by JUNOS stateless firewall filters.

A third option uses the JUNOS *sampling* framework. This feature of the PFE duplicates packets according to a configurable profile of how often to do so and for how long. All packets enter the PFE sampling engine by way of a stateless firewall filter ap-

plied to an interface. These duplicate packets may be steered to an MS module for service processing. Given the original packet is forwarded without impact and is not modifiable by the service, sampling is best suited to monitoring-style services.

These options in combination with other JUNOS features make for a large array of possibilities to design a service solution.

3. CREATING SERVICES WITH THE JUNOS SDK

In this section we maintain a focus on the services and control planes discussed in the previous section, while looking specifically at the JUNOS SDK to see what it enables, how it deals with multiple providers of software, and what issues surround availability and reliability of the JUNOS Software and applications. We start by examining the JUNOS SDK in the environment of the control plane and how it relates to creating services.

The JUNOS SDK exposes a growing amount of JUNOS functionality through application programming interfaces (APIs) that interact with various JUNOS components. These APIs are C-programming language headers and libraries. They are compatible with C++ as well. The POSIX and principal APIs used in FreeBSD are available, such as libc.

3.1 Creating a Management Component

The control-plane functionality available in JUNOS is centrally controlled on the RE by the JUNOS kernel and a suite of daemons. The *Routing Engine SDK* (RE SDK) exposes much of this functionality, allowing the creation of a broad class of management applications and signaling protocols that run in user-space as a non-root user. RE-SDK applications, like JUNOS processes, are generally event driven, and either daemons or utility-style applications launched on demand from an operational command.

The support for multiple providers is inherent for RE-SDK applications, where application packages, upon install, are unpacked into file-system paths containing a provider ID extracted from a certificate inside the package. This Juniper Networks assigned identifier is unique per provider. It prevents collisions, and identifies non-native JUNOS Software requiring verification before being run. This involves a certificate validity check (signed by Juniper Networks), and configured permission of the provider ID by the router operator. Additionally, all third-party code is controlled under a flexible resource control policy as to not overrun or disrupt other providers' or native processes. This imposes limits on memory usage, CPU usage, the number of open file descriptors, and socket permissions.

The kernel's functionality plays a key role in enabling application high-availability. The RE SDK provides APIs to access and store opaque data in the JUNOS kernel. If valuable processed data is at hand, the kernel's storage can be used to help the application pick up where it left off in the event of a restart. The JUNOS Software also features the support of a backup RE that can take over as the master RE in the event of a failure. This feature, called graceful RE switchover, when turned on, also causes the master RE's kernel to automatically replicate this opaque data to the backup RE. An RE-SDK daemon standing by on the backup RE can retrieve this data. If there is no daemon running on the backup RE, it would be launched upon switchover and retrieve the last saved data.

A management component of a larger service application would automatically take advantage of and fit into the model described thus far, but likely also use a good deal of the interfaces available in the RE SDK. There are many JUNOS SDK libraries, but one such example of a library exposing JUNOS functionality is a library that communicates with the SDK Service Daemon, namely libssd. This library may be used to manipulate routes (including service routes) which are ultimately managed by rpd. Libraries may also communicate directly with the JUNOS process that manages some area of functionality. A library to dynamically manipulate and apply stateless firewall filters and rate limiters, libdfwd, is one such case. Both libssd and libdfwd provide programmatic access to manipulate the forwarding behavior on the device. Other APIs are available that communicate with the kernel to, for example, subscribe to interface state change notifications. A management component could use some of these tools to steer traffic to an MS module and for other purposes.

A chief motivation for the management component is its ability to act as a proxy between the user interface and the services plane. An RE-SDK application, when installed, can extend the operational and configuration user interfaces in several ways.

The RE SDK provides a new language to facilitate the job of extending the schema for the hierarchy of available operational commands. It allows the definition of the syntax for any new commands and registering the RE-SDK application as the handler. When the application receives the command, it may want to return some resultant data. This return is facilitated by another new language used to define the structure of an XML response and how to format it if it is to be displayed to a user.

The schema for the hierarchy of object nodes available in the configuration database may be extended as well. This capability allows the operator to configure an RE-SDK application with new syntax. Another language facilitates the task of defining new configuration syntax and designating points within the configuration database that, if present, require an application to be run or notified about a configuration change. Read-only access to the configuration database is possible through APIs exposed by the RE-SDK.

Finally the control and services planes are tied together with inter-process communication (IPC) mechanisms. Usually an RE-SDK-implemented management component pushes configuration to the service application through some IPC mechanism. Such configuration consists of application-specific policies. If the management component is managing a service on more than one MS module, it must send the correct policies to the service application on each MS module. The RE SDK provides synchronized, segregated IPC and facilitated addressing in libraries based on TCP/IP sockets or through an extension of the kernel's data store mechanism described above.

3.2 Creating a Service Component

User-space applications for an MS module are constructed with the *Services SDK*, which exposes most of the same interfaces as does the RE SDK (e.g. libssd and libdfwd), with the addition that it makes use of the MS module's packet-processing-oriented resources.

Because the MS module runs the JUNOS kernel with the JUNOS TCP/IP stack, creating server-style applications with the Services SDK is equivalent to using the RE SDK. The focus of the services plane and this work, however, is not on this type of applications, but on applications that process selected packets transiting the data plane. We call these packets *data traffic*. These packets are selected using the aforementioned steering methods. Two models are available for building such applications in the Services SDK. An individual MS module supports only one model at a time.

3.2.1 The Process Model

In this model, one constructs a two-component daemon to run on the MS module as in Figure 1. The *control component* communi-

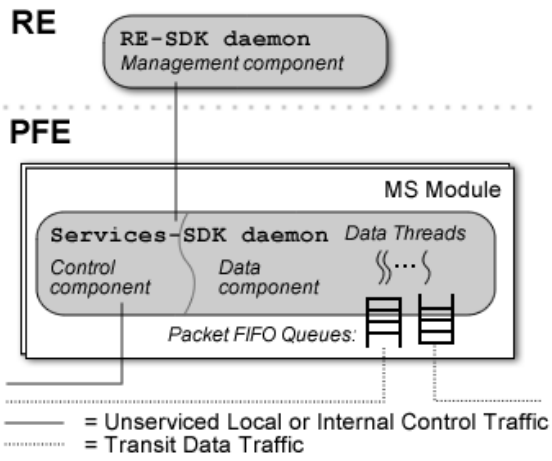

Figure 1: Architecture of a Service Application

cates with the management component. At a minimum it receives and stores the configured service policies, but sending statistics and status information is commonplace as well. The *data component* uses these stored policies and performs the servicing on the data traffic. The data component spans many real-time software threads, each tied to a single and exclusively used hardware thread of the MS module's CPU. Each of these *data threads* spins in a *data loop* polling for packets. During the startup phase before these threads are started, the Services SDK and JUNOS kernel set up a series of zero-copy input and output first-in-first-out (FIFO) queue pairs, where eventually one data thread is tied to and services one pair of queues in its data loop. Generally the steps of each loop are to receive a packet from the input queue if one is available, process it, and enqueue it in its output queue. Processing is done with either the service policies stored in the memory shared with the control component or saved application state (usually for a flow or session).

The kernel is responsible for putting the data traffic steered from the data plane into the input queues, and re-inserting the packets from the output queues back into the data plane. The MS module also has many configurable boot-time properties. One that is of interest here is how the kernel distributes data traffic among the input queues of the data threads, since there could be between 1 and 21 such threads and queues. There are two options: round-robin distribution or flow affinity. Flow affinity indicates that packets of the same flow are always directed to the same input queue, and hence, to the same data thread. The definition of a flow in this case comes from a 3-tuple: the IP addresses and the identifier for protocol the IP packet is carrying.

Part of the Services SDK provides the APIs that assist the setup of this multithreaded environment to work with the hardware threads of the CPU. It also includes fast-access shared memory APIs, locking constructs suitable to the environment, and packet manipulation functions.

3.2.2 The Plug-in Model

The Services SDK provides the plug-in model to allow several data-traffic-oriented services to coexist and even cooperate through an event framework within a single MS module. Plug-ins can send and receive custom events in a loosely coupled way, but here we focus on the standard system events. Furthermore, this model facilitates maintaining the service policies and accessing flow- or session-based state. In this case a flow is defined by the standard 5-tuple: 3-tuple with port numbers when available. Also, IP fragments are automatically reassembled before processing.

In this model, a JUNOS daemon named mspmand starts and creates its own data component to run one or multiple plug-ins linked into it as shown in Figure 2. The plug-ins are implemented as shared libraries. Each has an entry function that, when called, registers callback functions to serve as the control and data event handlers. *Control events*, called from non-data threads, can initialize the plug-in or update downloaded policies according to a management component. Data threads are managed by mspmand. Each data thread internally polls for packets and dispatches them as *data events* serially through a chain of plug-ins. This model works only with the service-set steering method, which allows for a service order to be specified by the operator per service set. This order ultimately determines the order of packet delivery, and hence service processing.

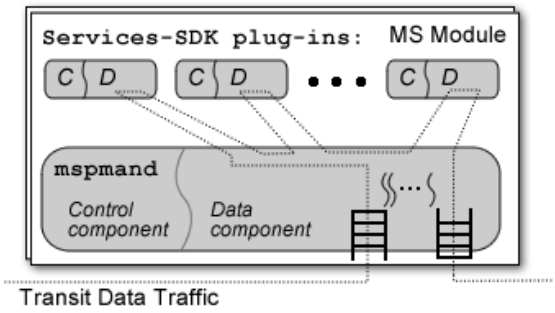

Figure 2: Packet distribution to plug-ins from mspmand

A session context to store custom session state is delivered as meta-data accompanying packets with each data event. This mechanism can expedite processing if plug-ins store policy decisions in the context once they have been looked up for the first packet of a session. Furthermore, packets of the same session are always dispatched one at a time to reduce lock contention for session state access.

This model has numerous advantages for some styles of service applications. First, if a service is flow or session based, the framework sets up the data loops and provides a flow context with every packet. Services from multiple providers (Juniper Networks and third parties) can run on a single MS module to collaborate in a configurable order. Also, this model supports pairing two MS modules together, where one is standing by with replicated state information in the event of a failure. Overall, this model generally provides more features and facilitates rapid development where service processing is the main focus.

4. CASE STUDIES: SERVICES CREATED WITH THE JUNOS SDK

In this section we describe our steps to build two different applications with the JUNOS SDK, and how the concepts presented in the previous sections were put into real applications.

4.1 MoniTube

MoniTube is an application that can be used to monitor the quality of IPTV streams at any point where it is deployed on the network by using a metric known as the media delivery index (MDI) [6]. MoniTube can also mirror IPTV streams to other locations.

When MoniTube is installed, the JUNOS user interface is seamlessly extended to allow for its configuration and for reporting of

the streams' quality through operational commands. Configurations can then be entered to separately identify the streams of interest for monitoring and mirroring. A MoniTube management application runs on the router's RE to load this configuration and transfer it to a MoniTube service application running on an MS module.

The MoniTube application needs to see all packets of the streams it is to monitor and mirror with no need to alter the original streams. Therefore, the application's packet-processing functionality runs on an MS module that does not receive the original streams, but rather copies of all its packets.

To receive copies of packets, we take advantage of the JUNOS sampling extension that provides the ability for an operator to send sampled packets to an MS module where third-party code can process them. Our application needs to receive all packets selected for sampling, so the sampling rate is configured as 1, indicating that every packet steered to the JUNOS sampling framework is sampled to the target module, and hence, the MoniTube service.

We use firewall filters applied to the router's interfaces to select packets for which sampling is required. This approach gives the operator flexibility to select all or a subset of the router's interfaces and traffic to be under MoniTube's management. For example, if there were a variety of traffic flowing into the router through a given interface, the operator could setup a filter to match all UDP traffic in 226.0.1.0/18 and 228.1.2.3/32 to be sampled, and thus, directed to MoniTube.

The MoniTube service is implemented as a multithreaded daemon running on the MS module. Thus, it follows the Services SDK's process model. Its main thread (control component) is primarily responsible for communication with the management component; but more interesting are its (up to) 18 other real-time data threads dedicated to packet processing. These threads poll the input packet queues, dequeue packets, perform the monitoring calculations, do any packet manipulation for mirroring (e.g. changing the destination address), and then if the packet is not mirrored, dispose of the packet since it is a duplicate of the real selected traffic. If the packet is to be mirrored, then we send it as modified during the processing, enqueuing it into an output queue. This application architecture can easily be applied to dealing with real packets instead of copies, where the original monitored packet is sent by default and not dropped. However, with sampled traffic, we know we will never adversely impact the original IPTV streams.

4.2 Equilibrium

Equilibrium is an application that provides two simple functionalities that are implemented and potentially deployed as separate services: load balancing and traffic classification.

The load-balancing service has some configured façade addresses, and for each has a pool of real addresses. When it sees traffic destined to a façade address, it redirects it to an address from the pool with the least load based on the number of connections. The classification service provides the ability to redirect traffic to a single different destination address, but based on a match with a destination port number rather than a destination (façade) address.

Once installed, just as with MoniTube, a management component runs on the RE, reading in any configuration for each of these services. It passes this information down to the service's control component running on the MS module. Multiple MS modules may also be configured to run one or both of the Equilibrium services, which work through JUNOS service sets. The management component is also responsible for gathering this information, and for sending only the required policies to the Equilibrium service running on each module.

The motivation for using JUNOS service sets was twofold. First, unlike the case of MoniTube, both services need to act on real traffic transiting the router. A service set specifies an ordered set of services with this requirement, and one or more sets may be associated with an MS module. Data traffic is redirected to the service set when filtering on interfaces detects matched packets that need to be serviced. Second, we want to allow Equilibrium services to be run together on one MS module, and potentially with other JUNOS or third-party services. The plug-in model achieves this aim, and generally plug-ins are meant to work with service sets.

In accordance with the configured order in the service set, packets pass through the data-event handlers of the Equilibrium services and any other services configured in the same set. When seeing the first packet of a session, we look into our configuration policies for façade-address or port matches and store the corresponding action in the session context. For subsequent packets of the same session this action is available in the context and immediately applied. This method expedites servicing, as we simply retrieve the original action taken on packets of each flow.

Both of the applications described are sample applications provided with the JUNOS SDK, where all code, build, and deployment instructions are provided.

5. CONCLUSION

We have explained the three main conceptual facets of JUNOS, and we focused on how the JUNOS SDK permits service-plane development with the Services SDK. Specifically, developed software can be collaborative, modular, and allows for its flexible arrangement for service processing. By examining the RE SDK, we also saw that numerous libraries expose the ability to create a broad range of applications in the control plane, and that by use of either the RE or Services SDK, the data plane (PFE) hardware resources are malleable under the control of third-party code. JUNOS SDK application packages are easily installed in familiar ways on a JUNOS system while security is still maintained. Furthermore, the applications created with the JUNOS SDK already work on the standard releases of JUNOS that are installed on a substantial number of supported Juniper Networks devices.

The JUNOS SDK, offered through Juniper Networks Partner Solution Development Platform enables rapid service creation. It provides straightforward extensibility and mitigates well-known issues with the classic closed nature of networking equipment. As a result, the JUNOS SDK offers a novel approach to facilitate innovation in the network.

6. REFERENCES

[1] FreeBSD web site. `http://www.freebsd.org/`.

[2] JUNOS Software, Technical Documentation, Juniper Networks. `http://www.juniper.net/techpubs/software/junos/index.html`.

[3] Multiservices DPC Datasheet, Juniper Networks. `http://www.juniper.net/us/en/local/pdf/datasheets/1000258-en.pdf`.

[4] Multiservices PIC Datasheet, Juniper Networks. `http://www.juniper.net/us/en/local/pdf/datasheets/1000199-en.pdf`.

[5] Partner Solution Development Platform (PSDP), Juniper Networks. `http://www.juniper.net/us/en/products-services/nos/junos/psdp/`.

[6] J. Welch and J. Clark. A Proposed Media Delivery Index (MDI). IETF RFC 4445, April 2006.