Software Architecture Overview

Control Plane-Platform Development Kit 2.11

March 2004

Contents

Figures

Tables

Revision History

Revision	Description	Date	Author
2.11	Updated for Release 2.11	March 2004	Anantha Rathnam
2.1	Updated for Release 2.1	December 2003	Anantha Rathnam
2.0	Updated for Release 2.0	August 2003	Anantha Rathnam

Part 1: Overview

1 Overview

Network elements such as switches and routers can be classified into three logical operational components:

- Control plane
- Forwarding plane
- Management plane

The control plane controls and configures the forwarding plane. The control plane executes different signaling or routing protocols and provides all the routing information to the forwarding plane.

The forwarding plane manipulates the network traffic. The forwarding plane makes decisions based on the network traffic and performs operations on packets such as forwarding, classification, filtering, and so on. An orthogonal management plane manages the control and forwarding planes. For example, the control plane in a router executes routing protocols, the forwarding plane performs hardware-based switching, and the management plane starts or stops routing process or performs logging.

The introduction of standardized APIs within the above-mentioned planes can help system vendors, OEMs, and end users of these network elements to mix and match components available from different vendors and achieve a device of their choice. The Network Processing Forum (NPF) API is designed for this purpose, and it presents a flexible and well-known programming interface to the control plane applications. The NPF API makes the existence of multiple forwarding planes, and vendor-specific details, transparent to control plane applications.

The hardware properties and nature of interconnect used between the control and the forwarding planes are isolated. Thus, the protocol stacks and network processors available from different vendors can be easily integrated with the NPF APIs. The APIs included in the Control Plane Platform Development Kit are based on the NPF APIs. For more information about NPF, refer to http://www.npforum.org/.

The software architecture overview document explains the high-level architecture of the Control Plane (CP) Platform Development Kit (PDK), which implements the NPF APIs.

1.1 Terminology

Table 1 lists the terms used in this document and provides definition for each term.

Table 1. Terms and Acronyms

Term	Description
ARP	Address Resolution Protocol
Co-located PDK	A version of the CP-PDK where the control and forwarding planes execute in the same process. Differences between the co-located and standard PDK are limited to the Transport Plug-in
Control Element (CE)	In a separated control/data system, refers to the processor(s) responsible for control and configuration of forwarding elements. Used interchangeably with Control Plane (CP)
Control Plane (CP)	See Control Element (CE)
COPS	Common Open Policy Service protocol

Term	Description
CORBA	Common Object Request Broker Architecture (http://www.omg.org)
Core Component	An object representing a single instance of packet processing, conceptually containing a classification and action execution phase. Some Core Components have well-defined APIs
CP-PDK	Control Plane Platform Development Kit
DiffServ	Differentiated Services. A layer-3 packet-forwarding scheme where all packets receiving the same per-hop forwarding behavior are marked identically using the IP header DSCP field. Standardized by the IETF
ForCES	Forwarding and Control Element Separation protocol, currently being standardized by the IETF
Forwarding Element (FE)	In a separated control/data system, FE refers to the processor(s) responsible for fast path forwarding of data. It is used interchangeably with the FP
Forwarding Plane (FP)	See Forwarding Element (FE)
GSMP	General Switch Management Protocol
ICMP	Internet Control Message Protocol
IntServ	Integrated Services. An Internet service model that includes best-effort service, real-time service, and controlled link sharing
IXA	Internet eXchange Architecture
MPLS	Multiprotocol Label Switching
NPF	Network Processing Forum
OSPF	Open Shortest Path First (routing protocol)
RIP	Routing Information Protocol

1.2 References

Table 2 lists the documents that are referenced in this document.

Table 2. References

Reference	Document
[1]	NPF API Framework, Version 1
[2]	Platform Independence Layer API Reference
[3]	Namespace API Reference
[4]	IPv4 API Reference

Part 2: Control Plane PDK Architecture

2 Control Plane PDK Architecture

The architecture of the control plane PDK defines three main components as shown in Figure 1:

- Control plane module
- Transport plug-ins
- Forwarding plane module

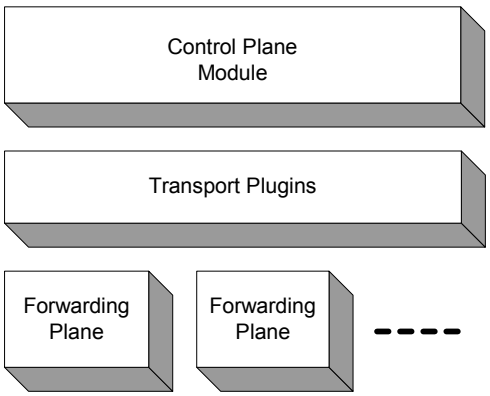

Figure 1: Control Plane PDK Architecture Components

2.1 Control Plane Module

The NP Forum defines the following two sets of APIs that are explained below:

- NPF Application API
- NPF Management API

The NPF application API is a collection of sub-APIs, each of which is focused on providing some specific functionality. Examples of sub-APIs are the IPv4 Unicast Forwarding API, MPLS API, DiffServ API, and so on.

The APIs are based on the standardization efforts within the NP Forum. Due to the evolution of the standardization process, API implementation requires modification. The control plane module provides the implementation of these APIs, and other infrastructure required to plug in legacy applications, and routing protocols to the platform.

Some features of the control plane module are as follows:

1. The control plane is aware of the presence of multiple forwarding planes and has the intelligence to perform one-to-many mapping. For example, if the control plane is present, for each new route specified by the routing table manager, **n** routes are downloaded to **n** forwarding planes.
2. The control plane module is aware of the underlying forwarding plane topology (in the case of multiple forwarding planes) and if required, it enables inter-forwarding plane forwarding.
3. Binding and capability discovery of the forwarding planes detects new forwarding elements, thus providing support for hot-swap of forwarding planes.

4. An FP plug-in API abstracts the transport mechanism and protocol used for communicating with the forwarding planes. This is very similar to the corresponding NPF APIs except that the users of this API must be aware of multiple forwarding planes.
5. The control plane module exposes and implements several NPF APIs, such as the IPv4 Unicast Forwarding API, Namespace API, Configuration and Management API, Classification API, and some forward-looking APIs such as MPLS and DiffServ.
6. The control plane module uses an OS abstraction layer to achieve independence from the control plane hardware and OS.

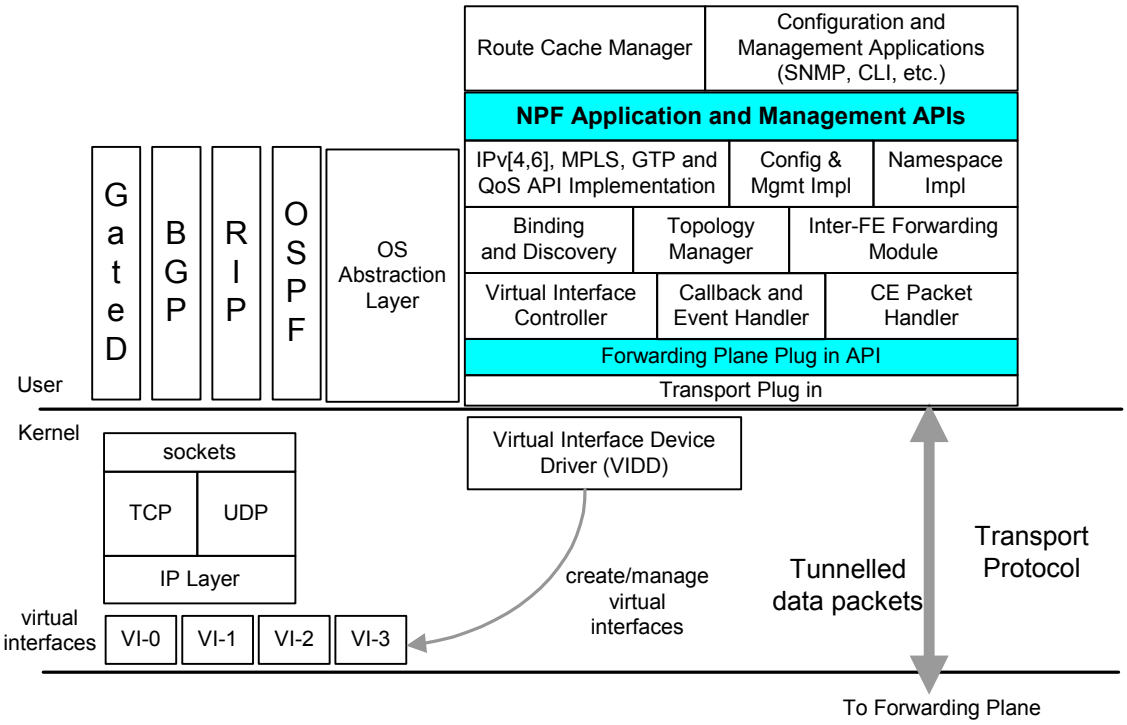

Figure 2: Control Plane Sub-modules of the Control Plane PDK

As shown in Figure 2, the control plane module is composed of the following modules:

1. Application API implementation module that implements the NPF Application APIs such as IPv4 Unicast Forwarding
2. Configuration and Management module that implements Configuration and Management APIs
3. Namespace module that implements the Namespace API
4. Binding and capability discovery module
5. Forwarding plane topology manager
6. Inter-FE forwarding module
7. Callback and event handler
8. Operating system abstraction layer, which abstracts out the underlying control plane hardware and OS
9. CP module manager

10. CP multi-client module
11. Protocol support services module that provides support for legacy applications and routing protocols. This consists of:

    a. Virtual interface device driver for providing an abstraction of the multiple forwarding plane interfaces as virtual interfaces in the control plane

    b. CE packet handler

    c. **Route Cache Manager:** The RCM acts as a proxy for the routing table manager (RTM) and allows the RTM to function without CP-PDK-specific modifications. If the RCM module is not present, the RTM should be modified to make IPv4 API calls instead of the **ioctls,** which is used for downloading the routing table entries to the Linux kernel.
    The routing cache manager module operates by opening a routing socket to the Linux* kernel and receiving notification of any changes made to the kernel route table by the RTM or any other module (ICMP). This RTM module makes the corresponding API call to the PDK on receipt of notification from the RTM. Thus the routing cache manager invokes the APIs on behalf of the RTM.

2.1.1 Application API Implementation Module

The API implementation module implements the NPF-defined Application API, which is a collection of APIs that provide support for developing well-known Layer 2, Layer 3 (IP) or higher control applications. The application APIs hide the existence of multiple forwarding plane s to applications and expose the details only when a control application needs them. This is accomplished with the use of the NPF-defined namespace and objects that are described later in this document. The IPv4 and IPv6 Unicast Forwarding APIs provide the following:

- Configuration and management of objects such as the IP route tables and ARP tables
- Receiving notifications such as ARP events and so on.

The MPLS API allows configuration and management of the control and forwarding planes for MPLS support. For the IXP platform, this includes API support for configuring the MPLS Core Component, setting up the labels required for label swapping, and so on. The ATM API provides for configuring and managing Virtual Path (VP), Virtual Circuit (VC), and traffic parameters. The GTP API provides for establishment, modification, and deletion of the GTP tunnels.

The QoS API consists of IntServ and DiffServ components and configures the control and forwarding planes for QoS support. The IntServ implementation is internally mapped to the DiffServ API within the Control Plane. The QoS API allows for configuration of the DiffServ Core Component, including QoS elements such as classifiers and markers for the IXP.

2.1.2 Configuration and Management Module

The configuration and management module implements the methods defined by the NPF Configuration and Management (CM) API. This module is used by management applications for configuring and managing different networking devices. These APIs facilitate the configuration and management of the following:

- Layer 2 objects such as bridges and forwarding databases
- Ports such as Ethernet

- Layer 3 IPobjects such as interfaces, the IP Route table, configuration for supporting MPLS, DiffServ, IPv6, and so on.

The configuration information of the forwarding planes is maintained in a persistent manner and made available during subsequent boots. There can be cases where transient data of the running system is stored by the relevant modules. For example, the CM API implementation module might store/cache information about individual forwarding elements such as the number of ports reported by the FE, their attributes and capabilities, IP addresses assigned to the ports, and so on.

2.1.3 Namespace Module

The namespace module implements the NPF-defined **namespace** that is used by multiple management applications to identify, locate, and group managed objects in an NPF-compliant system. The CP-PDK implements the namespace API, contains the named objects in the system, and possibly some state associated with them.

The CP-PDK provides mechanisms through the namespace API to access and invoke methods on them and manipulate the namespace. This includes methods to create, populate, and delete the nodes in the namespace tree, methods for enumeration and traversing the namespace tree, and so on. Other sub-modules of the control plane module interact with the namespace module to populate, modify, and query the namespace.

For example, the binding and discovery module populates the namespace with the objects, properties and attributes that were discovered on the forwarding planes. Figure 3 shows an example of how the namespace could be structured and maintained by the PDK.

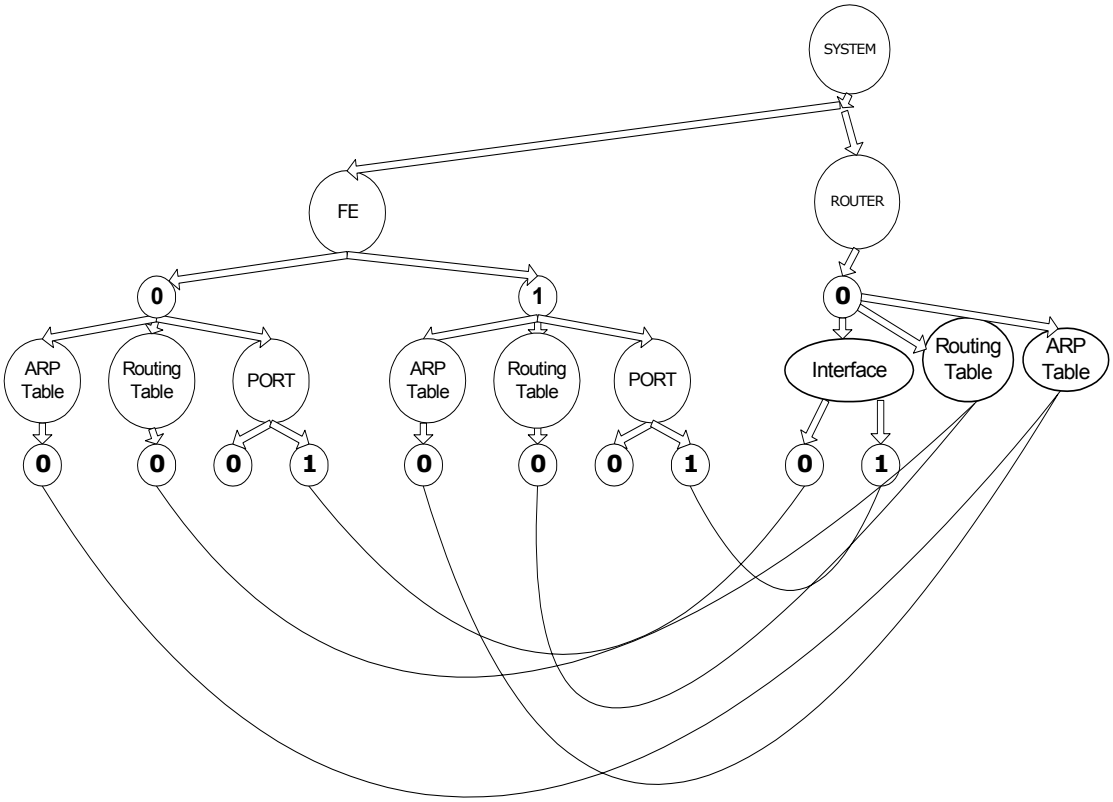

Figure 3: Sample Namespace Maintained by the Control Plane PDK

The namespace API is hierarchical and contains all configurable objects in the system. This includes objects that represent individual entities on the various forwarding plane s such as physical interfaces, forwarding and ARP tables, and so on. There are also objects that represent a set or a collection of related objects.

For example, if there are multiple forwarding planes, there will be a logical object that represents all the forwarding tables of all the forwarding planes. In Figure 3, there are two forwarding elements in the system that are configured as a single virtual router, with the interfaces on the router being mapped to ports on different FEs.

Similarly, the physical ARP and routing tables of each FE are mapped into a single logical ARP table and routing table. Thus the application level API can make use of the objects as needed. The API implementation takes care of generating appropriate messages for all the objects that it contains, for every invocation on a grouping object.

2.1.4 Binding and Capability Discovery Module

There is a forwarding plane dependent sub-module that is responsible for binding and capability discovery of the underlying forwarding planes. This module can also perform any forwarding plane specific initialization such as binding of Core Components. This module exposes a flat interface to the rest of the PDK implementation. It also exposes uniform semantics for discovery of the forwarding planes and their resources and capabilities to the PDK.

The binding and capability discovery module provides consistent semantics for heterogeneous forwarding planes. For example, in the case of IXP 1200, the presence of an MPLS Core Component on the forwarding plane would be seen as an MPLS-capable Port on the IXP 1200 by this module. This sub-module invokes the configuration and management sub-module and populates the namespace with appropriate objects based on either the learned capabilities of the forwarding planes and/or some static configuration.

2.1.5 Forwarding Plane Topology Manager

The way a particular control operation such as a route update is done depends largely on the topology in which forwarding planes are connected. For example, the forwarding planes could be connected in a bus, mesh, star, or other topology, like nested, that affects the control operations and the control data they carry. The control data being downloaded must also be slightly modified in some cases to simulate the **one virtual router** behavior. The forwarding plane topology manager sub-module handles these issues.

2.1.6 Inter-FE Forwarding Module

The inter-FE forwarding module is responsible for assigning labels to be used for inter-FE forwarding on per-router label information base. The labels are based on the next hop of the route or MPLS entries and are reference counted for packets coming in on different entry, destined for the same next hop. These labels are then installed using the MPLS module.

2.1.7 Callback and Event Handler Module

The APIs are designed to be asynchronous in nature. Thus, the applications using the APIs register callbacks that are invoked on completion of the call. In addition to this, the events are reported to applications through similar callbacks. The callback and event handler module is responsible for

maintaining all API callbacks registered by the applications and also the callbacks registered for event notifications.

2.1.8 CP Module Manager

The CP module manager is responsible for the initialization and shutdown of the CP module. The CP module manager starts all the sub-modules in the CP in a well-defined order, including the CP Agent, which is a part of the transport plug-in. The CP module manager also provides a robust infrastructure for servicing application callbacks so as to prevent the PDK from getting into a blocked state by the application function.

The callback infrastructure consists of a pool of callback threads and a FIFO queue of the callback messages. The PDK components push the callback messages into the queue and the first available thread picks up the topmost messages to callback to the application. The number of the callback threads and the message in the queue is adjustable to fit the demands of the application at the compilation time.

2.1.9 CP Multi-client Module

The CP multi-client module allows the PDK to run with multiple clients on Linux. The clients can be either on the same machine or access the PDK remotely across a network using TCP. To achieve this, the multi-client module uses ONC RPC and it consists of a client and server. The PDK runs inside the server, while the client runs along the user application clients of the PDK.

2.1.10 Protocol Support Services

The CP-PDK must provide additional support in order to support existing routing protocols and legacy control applications without modifications. . This is based on the fact that existing routing protocols and legacy applications use the traditional **socket** interface to send and receive protocol/data packets. Internally, the socket interface uses the different physical interfaces available on the device, corresponding to physical ports in a device-independent manner. The following modules provide complete support for legacy applications on the CP and are explained in the following sections:

- Virtual Interface Device Driver (VIDD)
- CE packet handler
- Routing cache manager

Virtual Interface Device Driver (VIDD)

By providing the control plane applications with **virtual interfaces** that represent actual physical forwarding plane interfaces, routing protocols and other control plane applications see the physical forwarding plane interfaces as virtual local interfaces on the control plane. Thus, the VIDD allows the forwarding plane to function without any modifications. Thus the Virtual Interface Driver (VIDD) module in the CP-PDK simulates all forwarding plane physical interfaces to the networking stack on the control plane.

The VIDD module is responsible for providing a uniform abstraction of the underlying multiple forwarding plane interfaces as **virtual interfaces**. A network element typically consists of a control plane blade and one or more forwarding plane blades. In Figure 4, each forwarding plane blade has five physical interfaces for example, five 10/100 Ethernet ports.

The routing protocols execute on the control plane blade and assume a socket interface. In order to preserve all the semantics of the socket interface, all the ten physical interfaces must be simulated to the

IP stack on the control plane. This simulation is handled by the VIDD. As a result, the routing protocols in the control plane can be executed without any modifications to the packet data unit (PDU) send and receive socket interface. Figure 4 illustrates the function of the VIDD.

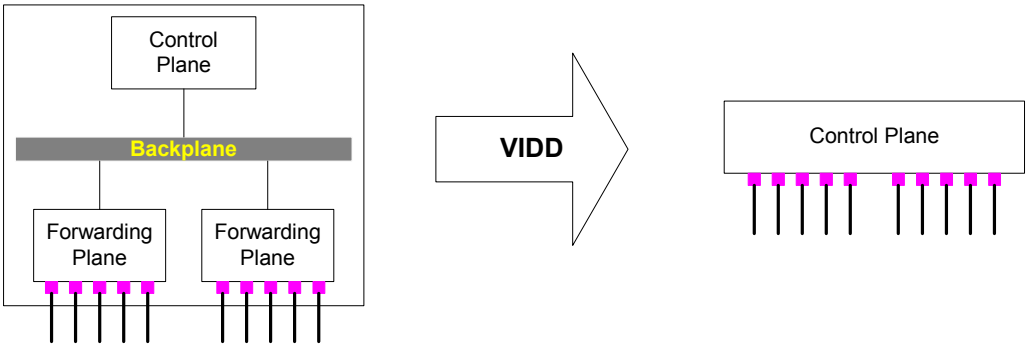

Figure 4: VIDD Virtualization Effect

The VIDD module also performs packet classification on the control plane. When the classification API is available, it will be implemented to allow control applications to register for any kind of packets/events, and so on. For example, a control application interested in ICMP packets arriving at a particular forwarding plane interface will receive them through the VIDD module. For the IXA platform on the forwarding plane, the pseudo device driver functionality will be a part of the stack driver.

CE Packet Handler

In addition to the virtualization of interfaces, the packets destined for protocols/applications on the control plane must be **transported** from the forwarding plane. The packets from the control plane transmitted to the virtual interfaces (which are actually meant to be sent to the corresponding physical interfaces on the forwarding plane) must also be transported to the forwarding plane. Such packets are termed as **data packets**.

Since virtualization of forwarding plane interfaces is essential for supporting unmodified control plane protocol stacks, the exact mechanism and protocol used to transport or tunnel these packets is implementation dependent and is independent of virtual interfaces. In the non co-located CP-PDK, such data packets are transported using an implementation of the ForCES protocol.

Route Cache Manager

A mechanism must also be provided to synchronize the routing table maintained in the control plane (usually in the kernel) with the routing tables maintained on the individual forwarding elements. Routing stacks such as GateD use the **ioctl interface** to populate the kernel FIB. With the VIDD in place, routing protocols running on the control plane continue to add and delete routes in the kernel FIB of the control plane. While these routes are needed in the control plane for correct functioning of the routing protocols, they must also be sent to the forwarding plane blades for configuration of the FIBs so that the packets can be forwarded.

An application called **Route Cache Manager** runs on top of the CP-PDK in order to distribute the route to forwarding plane FIBS. The route cache manager acquires these routes and invokes the IPv4 APIs. Then the CP-PDK handles the population of the FIBs on various forwarding plane blades with appropriate routes.

2.1.11 Operating System Abstraction Module

This module is responsible for abstracting out all the control plane hardware and OS dependencies from the API Implementation. It abstracts OS services such as threads, semaphores, critical sections, timers, memory, persistent storage, and so on. Control applications using NPF APIs may also use these services. It is the only component of the CP-PDK that directly interfaces with the OS and hardware. It could also be responsible for tracking resource usage, such as memory usage.

2.2 Transport Plug-in

The control plane and forwarding plane(s) can have different communication mechanisms or protocols to exchange information with each other. These protocols can be either IETF standard protocols such as ForCES/COPS/GSMP, or mechanisms such as CORBA, shared memory, and so on. The planes can also be connected using a number of different types of interconnects. Examples of such interconnects are InfiniBand, PCI, various back plane switching fabrics, shared memory, and so on.

The transport plug-in provides in-process communication between the control plane and the forwarding plane in the case of co-location where no interconnect is necessary. The transport plug-in abstracts out the type and the details of the communication mechanisms from the rest of the PDK, providing a plug-and-play functionality for different communication mechanisms.

Thus, you can place different types of transport plug-ins between the planes in such a manner that the control and forwarding planes communicate transparently. It is also possible (but not implemented) that in a single instantiation, different forwarding planes can be connected to the control plane over different interconnects. Figure 5 displays the communication of two forwarding planes with the control plane. Each forwarding plane is connected to the control plane over a different interconnect type.

In order to support the concept of transport Plug-ins, the CP-PDK introduces the concept of forwarding plane plug-in API (FP Plug-in API). The FP-plug-in API is very similar to standard NPF application level APIs with minor differences, which are explained below.

The FP-Plug-in API provides the layer of abstraction for the different transport plug-ins. Thus, the transport plug-in sends API invocations from the control plane to the forwarding planes. The forwarding plane also uses the transport plug-in to send control data and events to the control plane for processing.

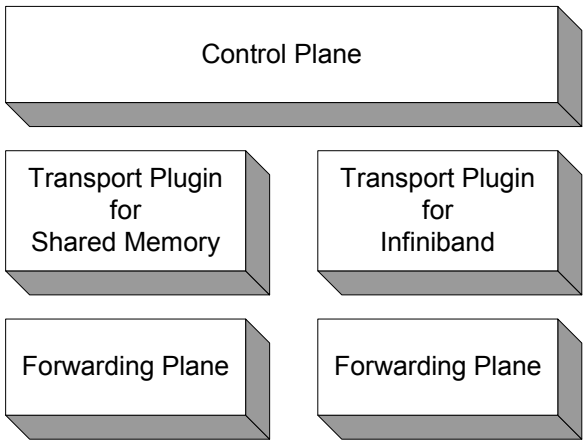

Figure 5: Transport Plug-ins for Different Interconnects

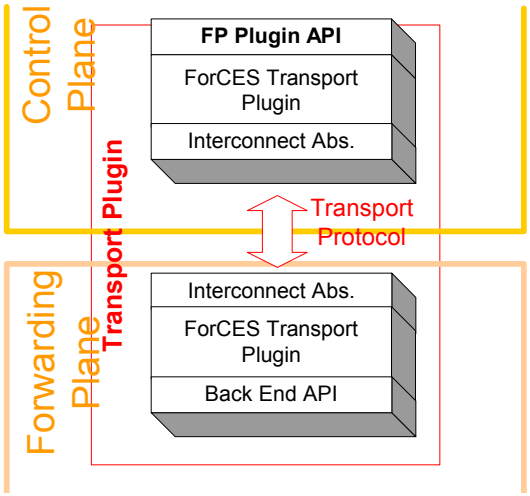

Figure 6: Transport Plug-in Architecture

The transport plug-in is composed of four distinct parts, as shown in Figure 6:

1. FP plug-in API
2. Transport protocol
3. Interconnect abstraction layer (not present in the co-located PDK)
4. Back end API

2.2.1 FP Plug-in API

The FP plug-in API hides the transport plug-in details and presents a uniform API that is invoked by the NPF API implementation modules. The FP plug-in API is similar to standard NPF application level APIs except for one critical difference; the FP plug-in API calls have knowledge of individual FPs, whereas the NPF APIs operate on higher-level objects.

The following is the application level API for adding a new ARP entry to the forwarding plane.

```
NPF_error_t NPF_IPv4UnicastToL2EntryAdd(
NPF_IN NPF_callbackHandle_t callbackHandle,
NPF_IN NPF_correlator_t correlator,
NPF_IN NPF_errorReporting_t errorReporting,
NPF_IN NPF_IPv4UnicastAddressResolutionTableHandle_t tableHandle,
NPF_IN NPF_IPv4UnicastNextHopResolutionEntry_t *entry);
```

The parameter t tableHandle represents an ARP table to which the entry must be added. This t tableHandle could represent a single ARP table on a single forwarding plane or multiple ARP tables on multiple forwarding planes, as discussed earlier. This one-to-n mapping between the control and forwarding planes is hidden from the control plane application that adds the new ARP entry.

In contrast, the corresponding API on the forwarding plane is as follows:

```
FPPI_RET FPPAPI_ipv4_unicastToL2Add(
      FPPI_FEID                    feid,
      FPPI_CORRELATOR              correlator,
```

```
FPPI_CBHANDLE                 cbhandle,
uint32_t                      entry_count,
FPPI_IPv4UniNHResolEntry_t*   entry_array)
FPPI_RET FPPAPI_ARP_AddEntry(
FPPI_FEID                     feid,
FPPI_CORRELATOR               correlator,
FPPI_CBHANDLE                 cbhandle,
npf_ipv4_ARPEntry_t*          entry_array,
uint32_t          e           ntry_count);
```

The parameter **feid** represents the id of the forwarding plane to which the new ARP entry must be added. The parameter **entry_array** contains information, which the forwarding plane uses to update the appropriate entries in its ARP table(s).

2.2.2 Plug-in Back End API

The plug-in back end API is the API exposed by the transport plug-in on the forwarding plane, and the FP Module of the PDK uses it. The plug-in back end API allows the FP module to do the following:

- Receive configuration and other requests from the CP
- Respond to the CP requests
- Send and receive data packets such as RIP, OSPF, and so on to and from the CP

2.2.3 Transport Protocol

The transport protocol can either be an IETF standard protocol such as ForCES/COPS/GSMP, or any other messaging system such as CORBA that can be used for transporting messages between the control and forwarding plane(s). The messages are sent between the control and forwarding planes via local queues and inter-thread events in the co-located PDK.

- **Control Plane Transport Plug-in:** The control plane transport plug-in resides on the control plane and implements the transport plug-in specific transport protocol and messaging. It is invoked by the FP plug-in API and converts the API calls to appropriately formatted messages that are sent to the forwarding plane agent.
- **Forwarding Plane Transport Plug-in:** The forwarding plane transport plug-in resides on the forwarding plane. It passes the transport protocol messages and generates well-known messages that are used by the forwarding plane module to invoke the vendor-specific API for the forwarding plane. This process is described in detail in section 2.3.

2.2.4 Interconnect Abstraction Layer

The interconnect abstract layer hides the interconnect details and is used by the transport protocol to send and receive messages. This provides an abstraction layer that hides the interconnect technology details from the transport plug-in. The transport plug-in uses this layer to send and receive messages without knowing if the interconnect is PCI, InfiniBand, or Ethernet.
Note: No interconnect abstraction layer is necessary for the co-located PDK as the control and forwarding planes execute in the same process.

2.3 Forwarding Plane Module

The forwarding plane (FP) module resides on the forwarding plane. It is specific to the forwarding plane API (FPAPI), which is used as the programming interface for the forwarding plane. The FP module receives the NPF API invocations from the FP agent and maps them to corresponding FP API invocations.

For example, the IXP-based FP maps the NPF API invocation to Core Component invocations. The FP module is also responsible for the binding and discovery of the FP. Figure 7 shows the internals of the FP module.

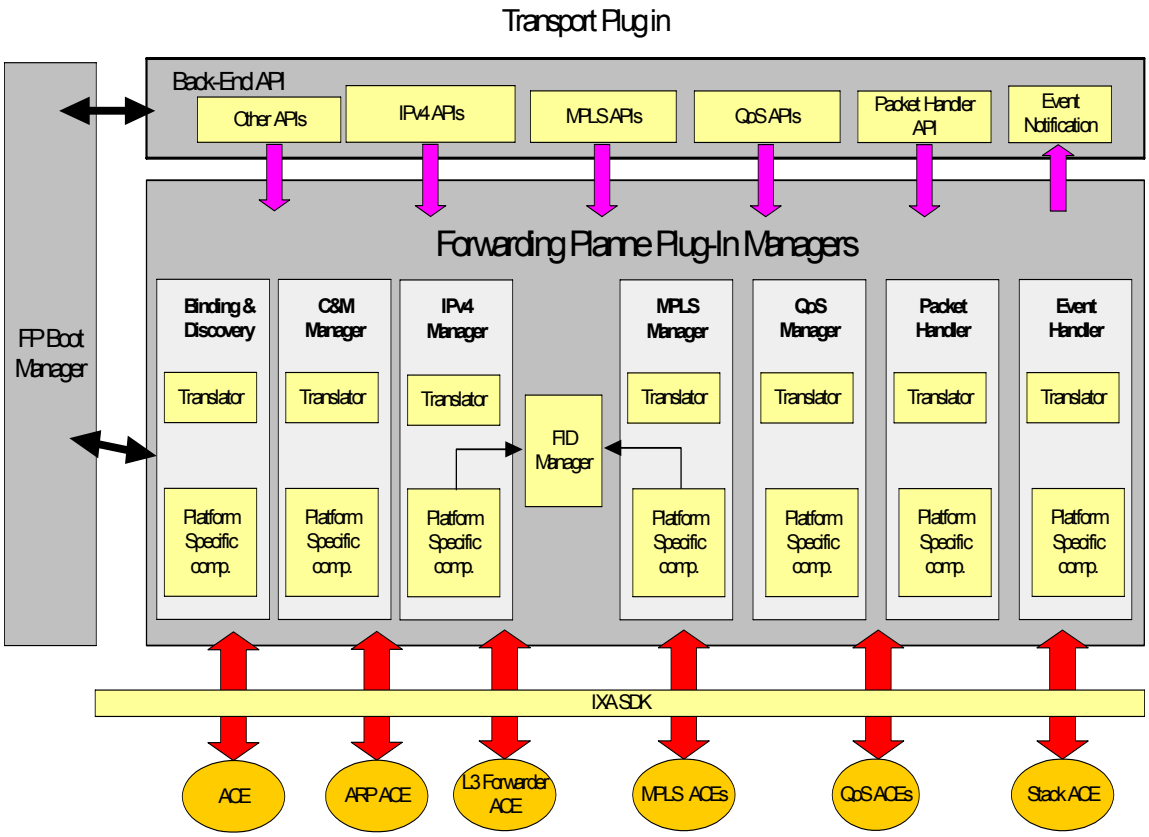

Figure 7: Forwarding Plane Module

The FP module has the following components:

1. FP boot manager
2. A number of FP plug-in managers each of which consists of the following:

   - Translator – This is a platform/OS independent component that translates NPF APIs, performs any required processing, and invokes the platform-specific implementation module.
   - Platform-specific Component – This module is usually very specific to the platform being used such as IXA SDK 2.0, IXA, and so on.

2.3.1 FP Boot Manager

The main responsibility of the FP boot manager is the initialization and shutdown of the forwarding plane module of the CP-PDK. It performs any platform specific initializations and starts up all the FP plug-in managers.

Each FP plug-in manager has to expose its initialization and shutdown routines to the FP boot manager. After initializing all the FP plug-in managers, the FP boot manager **runs** the forwarding plane. This results in initializing the binding process to a control Plane.

2.3.2 FP Plug-in Manager

Each FP plug-in manager handles a specific functionality; for example the IPv4 manager handles all the IPv4 support by working with the IPv4 component of the underlying NPU (IXA SDK, for example). Each FP plug-in manager is composed of a translator component and a platform specific component.

2.3.2.1 Translator

The translator provides a generic interface for each of the FP plug-in managers. It serves as an abstraction for the forwarding plane API specific component. This makes it easier to port all the FP plug-in managers to a different forwarding plane. For example, a Linux based FE will have Linux-specific components talking to the translators. The transport plug-in and the each FP plug-in manager's translator communicate through the backend API. Each translator contains algorithms to translate NPF API method invocations and dispatches them to the respective platform-specific component.

2.3.2.2 Platform Specific Component

Each platform-specific component is responsible for invoking the appropriate underlying NPU functionality for example IXA SDK 3.51 API. The platform specific components expose the internal APIs that are invoked by the translator. For example, the Add Label NPF MPLS API invocation requires classification and marking actions. This functionality is embedded in two Core Components within the forwarding plane. The MPLS manager's platform specific component receives the NPF API invocation and makes the two underlying IXA SDK 3.51 function calls corresponding to the single NPF MPLS API invocation.

Some examples of plug-in managers are shown in Figure 7:

- **Binding and Discovery:** The binding and discovery functionality of the FP module is partly FP specific. In the case of the IXA SDK, the binding and discovery functionality consists of querying the underlying IXA SDK for the FE capabilities such as number of ports, individual port attributes, QoS/MPLS capabilities, and so on.
- **Configuration and Management Manager:** The configuration and management manager is responsible for mapping the NPF interface APIs for configuring and managing interfaces on the forwarding plane.
- **IPv4 Manager:** The IPv4 manager maps NPF IPv4 Unicast APIs to the underlying FE.
- **Packet Handler Manager:** The packet hndler manager is responsible for interfacing with the IXA SDK to deliver and receive packets destined to/from the control plane protocols and applications. Data packets destined for the control plane such as packets destined for routing protocols, applications, etc., running on the CP must be transported to the control plane. The exact manner in which this is done can vary.
  In the CP-PDK, an FE packet handler module determines whether the data packet must be sent to a remote control plane. The FE packet handler module uses the transport plug-in to transport

the packet to the remote control plane. You can also use other mechanisms such as IP-in-IP tunneling. This additional module on the control plane completes the support required for running legacy routing protocol stacks, and applications on the control plane.

- **QoS Manager:** If the FE is QoS-enabled, the QoS manager maps the NPF differentiated services API to the underlying FE..
- **MPLS Manager:** If the FE is MPLS-enabled, the MPLS manager maps the NPF MPLS API to the underlying FE.
- **Event Manager:** The event manager is responsible for delivering events from the FE such as link status events, exceptions and so on.

The actual presence of each managers and their respective functionality are dependent on the system being deployed. For example, you can run the FE without QoS and MPLS capabilities. In such cases, the respective managers are not used.