Co-Located Transport Plug-in

Design Specification

Control Plane-Platform Development Kit 2.11

March 2004

Contents

Tables

Revision History

Revision	Description	Date	Author
2.11	Updated for Release 2.11	March 2004	Udaya Shankar
2.1	Updated for Release 2.1	December 2003	Udaya Shankar
2.0	Updated for Release 2.0	August 2003	Udaya Shankar

Part 1: Introduction

1 Introduction

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 and the forwarding plane manipulates the network traffic. The control plane executes different signaling or routing protocols and provides all the routing information to the forwarding plane.

The forwarding plane makes decisions based on this information 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, and performs logging.

The introduction of standardized APIs within the above-mentioned planes can help system vendors, Original Equipment Manufacturers (OEMs), and end-users of these network elements to mix and match components available from different vendors to achieve a device of their choice. The Network Processing Forum (NPF) API is designed for this purpose, as it presents a flexible and well-known programming interface to the control plane applications. It makes the existence of multiple forwarding planes, as well as 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. The protocol stacks and network processors available from different vendors can be easily integrated with the NPF Application Program Interface (API). The APIs included in the Control Plane Platform Development Kit (CP-PDK) are based on the NPF APIs. For more information about NPF, refer to http://www.npforum.org/.

1.1 Co-Located Transport Plug-in

The standard CP PDK includes an implementation of the control plane, the forwarding plane, and a transport plug-in that implements remote communication between the control and forwarding planes. Details concerning the standard PDK are in [1]. This document describes the design of a transport plug-in without remote communication capabilities appropriate for PDK co-location.

The co-located PDK differs from the standard PDK as the control and forwarding planes are not remotely separated but are executed in the same process. This eliminates the need for a specialized communication component within the transport plug-in component of the PDK. PDK changes for co-location are isolated to the transport plug-in component. This document focuses on the design of the co-located PDK transport plug-in.

1.2 Terminology

Table 1 lists terms used in this document and provides an expansion for each term.

Table 1. The terminology table

Term	Description
CP	Control Plane
FP	Forwarding Plane
ForCES	Forwarding and Control Element Separation protocol
NPF	Network Processing Forum
PDK	Platform Development Kit
TPI	Transport Plug-in

1.3 References

Table 2 lists documents referenced in, or related to, this document.

Table 2. The reference table

Reference	Document Name
[1]	Software Architecture Overview
[2]	Forwarding Plane Plug-in API Reference
[3]	Transport Plug-in Design Reference
[4]	Conformance Test Framework User's Guide

Part 2: Overview

2 Overview

The co-located PDK executes all PDK modules, including the control and forwarding planes, in a single process. Such an arrangement is beneficial to users who do not require control and forwarding plane separation. Co-location removes a level of complexity associated with remote communication between the control and forwarding planes.

2.1 Requirements

The co-located TPI must provide the PDK with:

- All the functionality of the standard PDK.
- The same callback behavior as the standard PDK. This implies that external CP API methods must return immediately before the method callback is invoked.
- Performance in terms of API execution speed and method invocation callbacks, when compared with remote module. Refer toTransport Plugin Design Reference[3].

2.2 High-Level Functionality Overview

The standard PDK consists of a control plane and one or more forwarding planes. Forwarding planes reside on and interact with network forwarding devices. The control plane configures and controls forwarding planes. The standard PDK allows for physical separation of the control and forwarding planes as shown in Figure 1. In this case, the control plane controls one or more remote forwarding planes. In the figure, only a single forwarding plane is shown for simplicity. The transport plug-in abstracts and provides communication functionality between the control and forwarding planes [3]. To achieve this, components of the Transport plug-in reside on the control plane and on the forwarding plane.

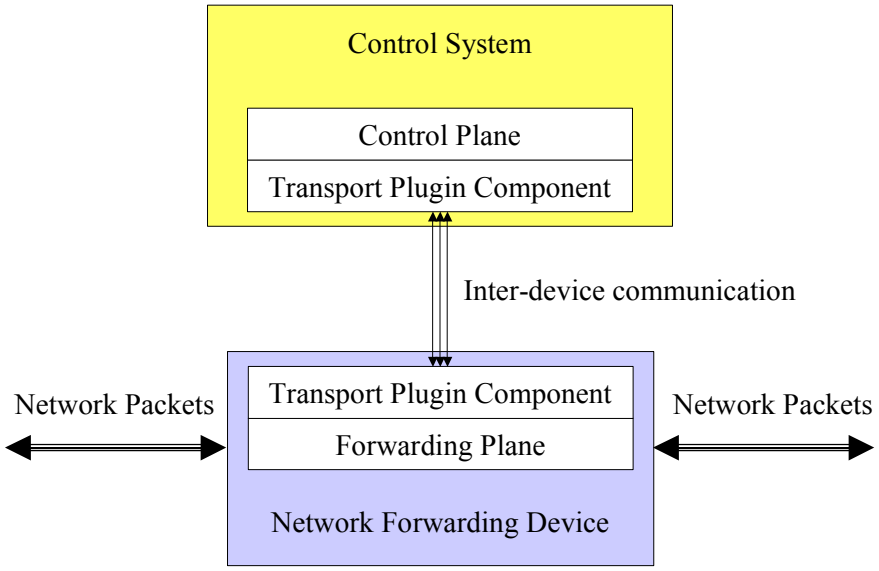

Figure 1: High-level overview of CP-PDK with remote control and forwarding planes

The co-located PDK executes the control and forwarding planes on the same network device as shown in Figure 2. The design differs from that of the standard PDK where in the transport plug-in is reduced to a lightweight version that does not offer any remote communication abstraction or functionality.

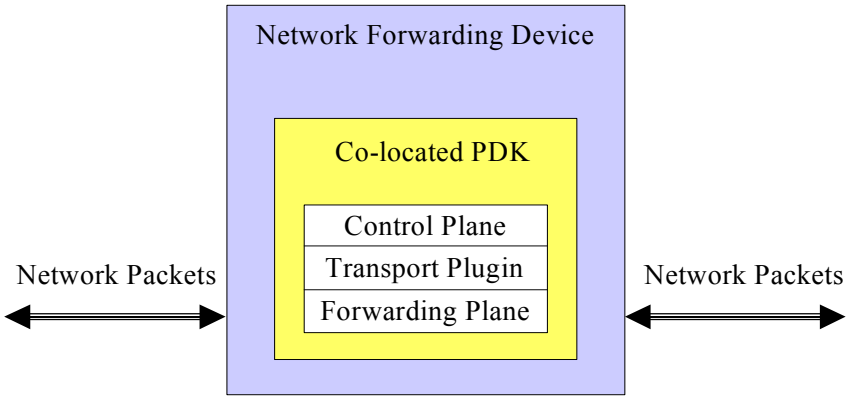

Figure 2: High-level overview of co-located PDK

2.3 Design Considerations, Assumptions, and Dependencies

The design of the co-located TPI was based on a number of considerations. The considerations are as follows:

1. The current PDK user experience includes an immediate return when a CP API method is invoked. PDK makes a callback into the calling application, returning any data requested by the API method. The co-located design preserves this behavior using message queues and threads.
2. The co-located design provides a level of thread separation between the CP and FP. This is implemented using two inter-thread message queues in the TPI and two servicing threads that service the queues. Using this model, any call made from the CP into the FP returns immediately. A TPI thread relays the message to the FP in a TPI local thread. In the same way, calls made from the FP to the CP are mediated by another TPI local thread.
3. The memory management model of the standard PDK is retained in the co-located TPI. In this model, the calling method frees its own memory. The called method is responsible for copying any memory it may require into its own memory space. To preserve this memory management model, the co-located TPI acts as a memory boundary. CP memory does not penetrate the FP and vice versa. The TPI copies data passing through it to ensure this behavior.

The TPI must be designed to absorb all necessary PDK changes to prevent, or at least minimize, changes to other PDK modules.

Part 3: Co-located PDK Design

3 Co-located PDK Design

A requirement of the co-located TPI design is to impart minimal impact on other PDK modules. To achieve this, all of the new co-location functionality is implemented through the new lightweight transport plug-in. The new TPI replaces the TPI of the standard PDK. In addition, the PDK build process is changed for co-location and a minimal amount of non-TPI source code files are slightly altered as described in the following sections.

3.1 High-level Overview

Since the designs of the control and forwarding planes remain unaltered between the standard PDK and the co-located PDK, we will not discuss their component architecture in this document. Their description can be found in [1]. A high-level view of the co-located TPI components is shown in Figure 3.

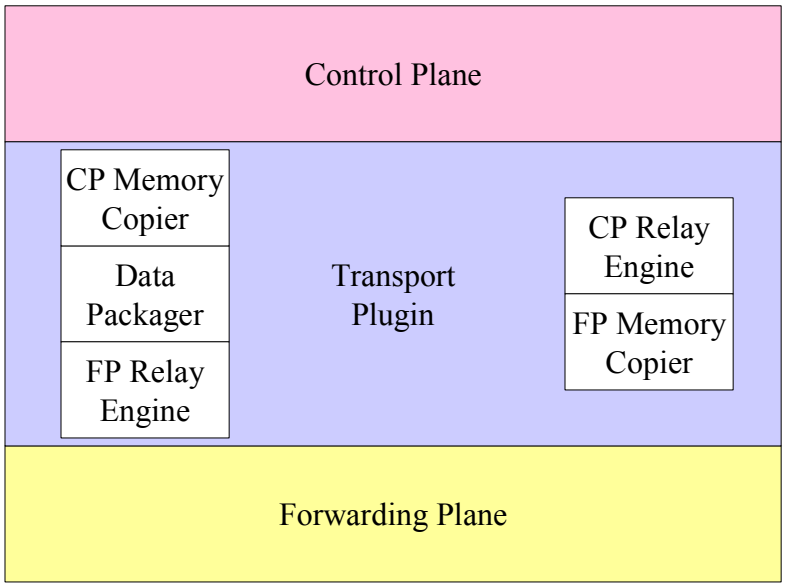

Figure 3: TPI component level diagram

The existing memory management model of the standard PDK is kept in the co-located design. The CP memory copier component copies data passed from the CP to the FP into TPI local memory. The FP memory copier component copies data passed from the FP to the CP into TPI local memory. The data packager component bundles data and API parameters from the CP into the types expected by the FP for each API. The FP relay engine invokes the appropriate method within the FP, passing all parameters stored in TPI memory. The CP relay engine invokes the appropriate method within the CP, passing all parameters stored in TPI memory. In each direction, calls include original methods and callback methods.

3.2 External API

The external API of the TPI in the co-located design is identical to the forwarding plane plug-in API and backend API of the transport plug-in in the standard PDK [2].

3.3 Implementation Details

This section describes the build procedure, startup and shutdown process of co-located transport plug-in.

3.3.1 Build Procedure

The build procedure of the PDK is augmented to accommodate the special needs of the co-located PDK. A new build variable is added which identifies that the build is either for a co-located PDK or for the standard PDK. When the build variable is specified to indicate a co-located PDK build, the following events transpire:

- Specialized code is included to provide independent global lists and other utility instances for the control plane and forwarding plane.
- Code responsible for remote communication between the control and forwarding planes is excluded.
- All relevant PDK components are built into a single executable.

3.3.2 Startup

Initialization of the various modules that comprise the PDK takes place upon PDK startup. The initialization routine of the standard PDK is altered somewhat for the co-located PDK. All of the changes are isolated to the TPI to eliminate impact on other modules.

The co-located PDK initialization process is shown in Figure 4 with emphasis given to TPI initialization. The CP registers callback methods with the callback manager component. The TPI is then initialized before initialization of most other CP-PDK components. During the TPI initialization process TPI specific objects are created such as lists, inter-thread events, and synchronization objects.

After TPI initialization, all remaining components of the CP-PDK are initialized. Upon completion of PDK initialization, the PDK start method is invoked. This method starts the TPI threads and waits for an inter-thread event from each of the new threads indicating that the new threads have completed startup and are ready to handle messages. The TPI then invokes the startup method of the forwarding plane.

The forwarding plane launches its own initialization routines, including spawning any threads needed for operation and registration of callback methods with the callback manager component. As part of the startup process, the FP invokes a bind request message to the CP. This message must be relayed by the TPI to the CP, necessitating complete readiness of the TPI at this point in the PDK startup process. The CP responds to the bind request with a ind response message to the FP. If the CP has accepted the FP bind request, the FP reports its capabilities to the CP by sending

it a capabilities request message. The CP responds with a capabilities response message, indicating its readiness to accept asynchronous events from the FP.

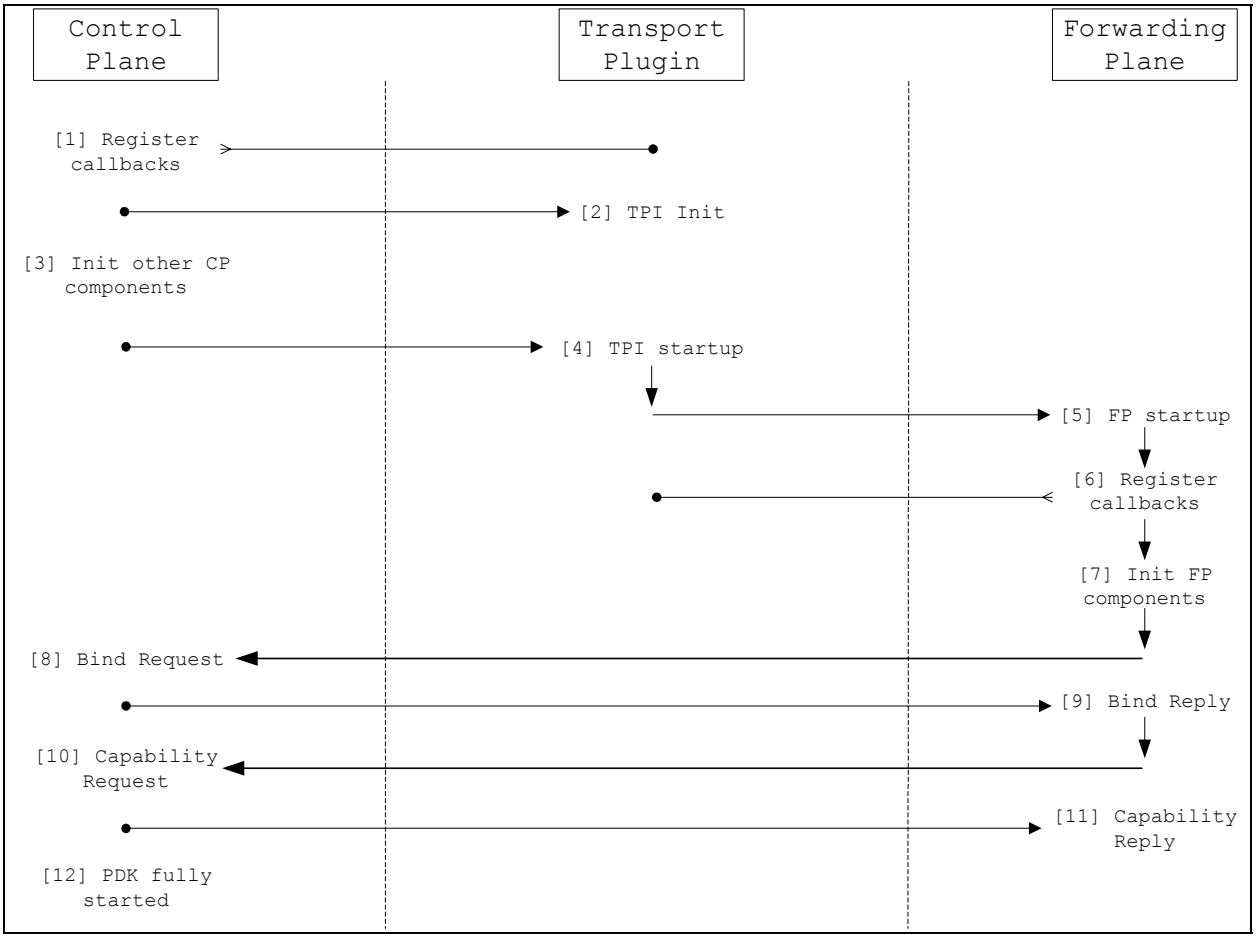

Figure 4: Co-located PDK initialization and startup sequence

3.3.3 Shutdown

The co-located PDK shutdown procedure is shown in Figure 5.

4. The CP deregisters callbacks with the TPI.
5. The CP then sends an unbind request event to the FP.
6. The FP then deregisters callbacks with the TPI and shuts down its internal components.
7. The CP shuts down its internal components before invoking the TPI shutdown method.
8. The CP frees all remaining resources and exits.

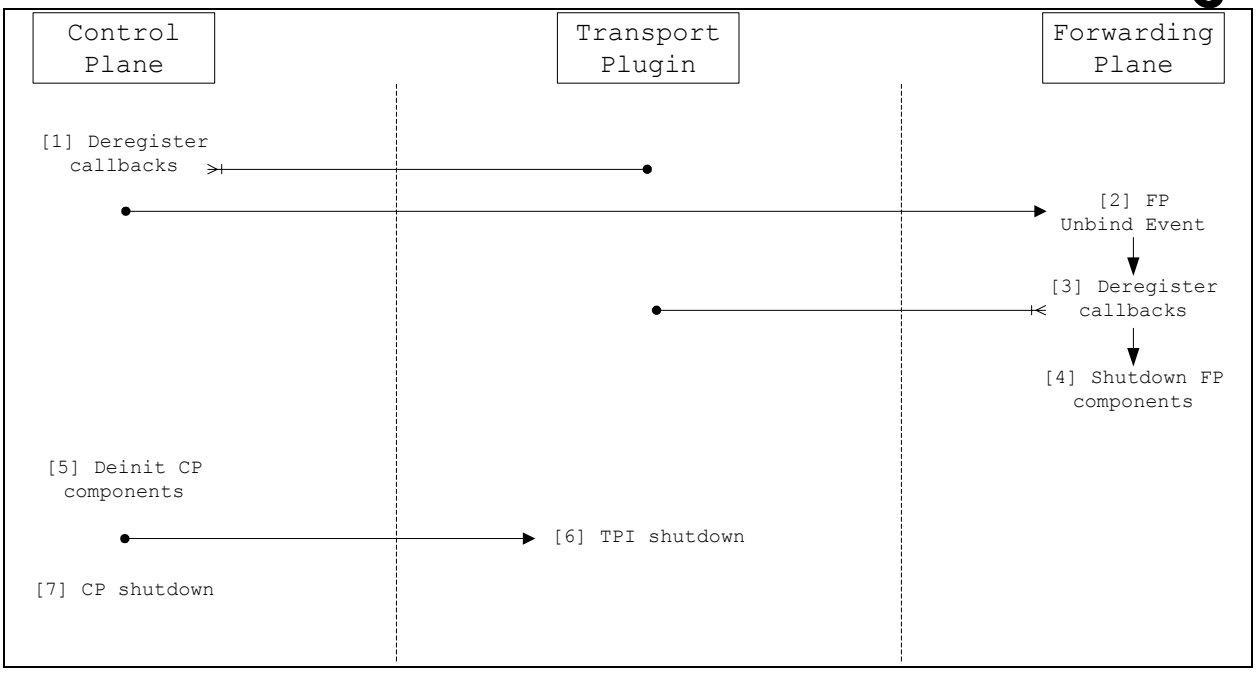

Figure 5: Co-located PDK shutdown sequence

3.3.4 Major Data Structures

A new data structure is introduced in the co-located PDK to equip the message queue service threads with the ability to service the message queues in a thread-safe manner. The data structure contains the following elements:

- A synchronization object to protect all structure elements
- A message queue
- A thread handle variable so that the master thread can terminate the servicing threads in emergency situations
- An inter-thread signaling object/event to alert the servicing thread when a message has been placed in the message queue
- An inter-thread signaling object/event to alert the control plane when the service thread is up and running during startup
- A return value to relay the last return code from the service thread before it exits

```
typedef struct tag_QUEUE_SERVICE_PARAMS
{
    DList               msgQueue;
    PilCriticalSection  queSyncObj;
    PilThread           thrdHandle;
    PilEvent            newMsgNotifier;
    PilEvent            thrdUpNotifier;
    FPPI_RET            retVal;
```

```
} QUEUE_SERVICE_PARAMS, * LPQUEUE_SERVICE_PARAMS;
```

A pair of these data structures is started by the main application thread. Each of the two TPI message-servicing threads is passed a pointer to one of the data structures.

Another new data structure for the co-located PDK TPI is a message structure. This structure is used as the format of all messages passed between the CP and FP through the TPI. The structure contains the following elements:

- A coarse grain event type used by the TPI to identify the message as one of the following:
  - Message to FP
  - Event to CP
  - Response to CP
  - Shutdown event
- A callback type ID used by CP and FP on which to base further action.
- A forwarding element ID to identify the specific FE in case there is more than one.
- A unique ID used to correlate callbacks with the original corresponding methods.
- A pointer to the PDK data being passed between the CP and FP.

```
typedef struct tag_DRS_GENERAL_EVENT_MSG
{
    uint32_t         evType;
    NPF_CBCAT        cbtype;
    FPPI_FEID        feid;
    void*            correlator;
    void*            outObj;
} DRS_GENERAL_EVENT_MSG, * LPDRS_GENERAL_EVENT_MSG;
```

3.3.5 Threading

The TPI provides a level of isolation between the CP and the FP to provide inter-plane thread separation. The isolation is implemented using two inter-thread message queues and two message service threads. The message queues are the means through which threads communicate with each other. For each queue, there is a message service thread that waits for messages to be inserted into the corresponding mailbox. This interaction is illustrated in Figure 6 and Figure 7.

For method invocations originating in the CP, the CP inserts FP-bound messages into the FP queue in a CP controlled thread as shown in Figure 7. The FP queue service thread blocks while waiting for incoming messages. When a message arrives, it wakes up and responds by calling into the FP. When the FP makes the TPI registered callback, a CP-bound message is placed into the CP queue. The CP queue service thread blocks while waiting for incoming messages. When a message arrives, it wakes up and responds by invoking the CP registered callback.

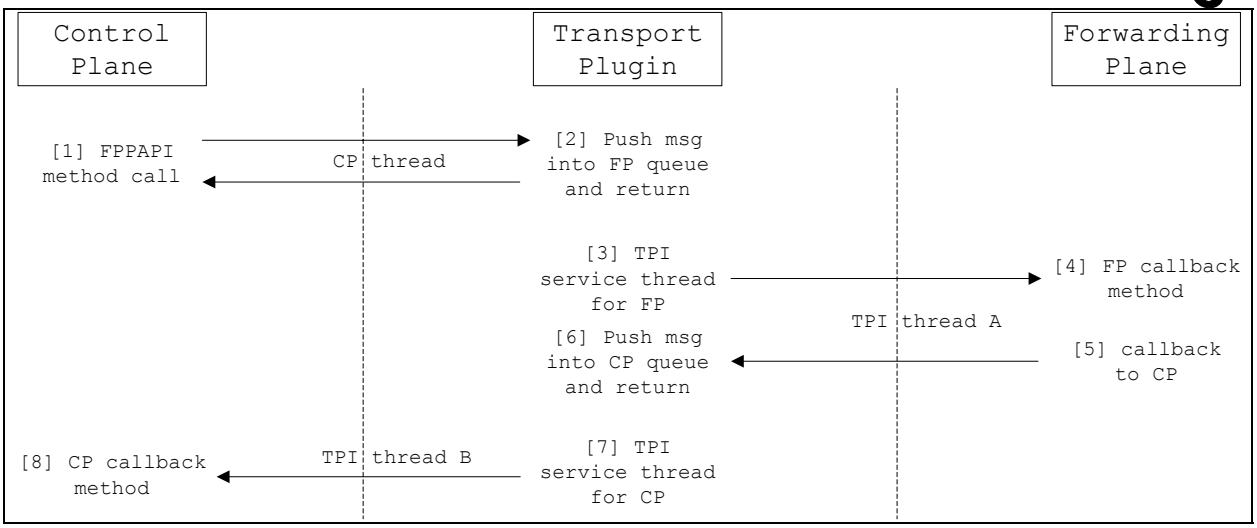

Figure 6: Sequence diagram and threading activity for calls originating in the CP

For method invocations originating in the FP, the FP inserts CP-bound messages into the CP queue in a FP controlled thread. The CP queue service thread wakes up and responds by calling into the CP. When the CP makes the TPI registered callback, an FP-bound message is placed into the FP queue. The FP queue service thread wakes up and responds by invoking the FP registered callback.

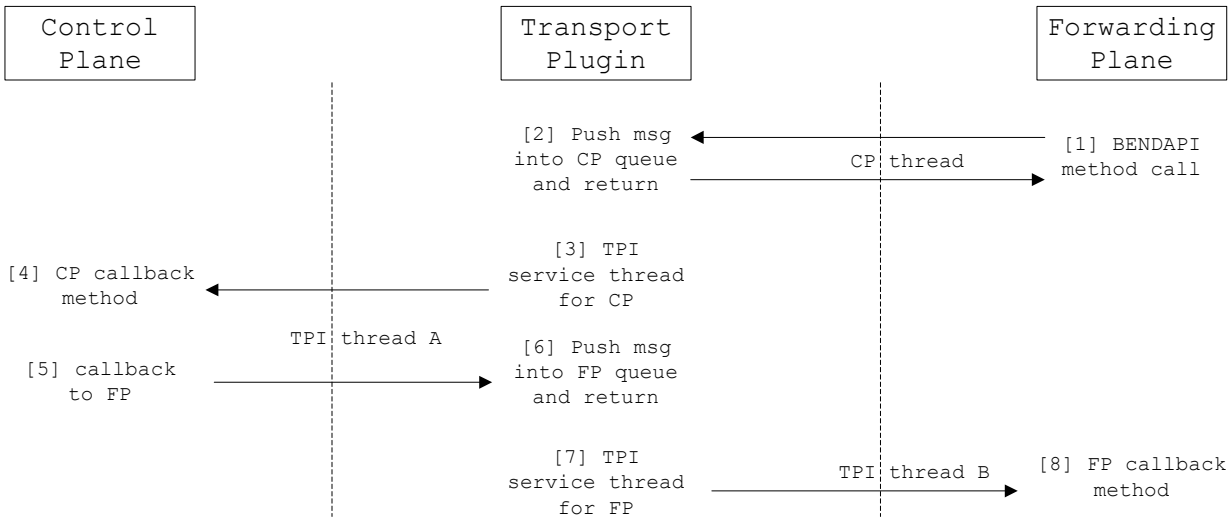

Figure 7: Sequence diagram and threading activity for calls originating in the FP

The queues are implemented as linked lists of pointers to dynamically allocated data structures. The data structures contain all the necessary message data.

3.3.6 Synchronization (reentrancy)

The new structures defined in the Overview section [2] of this document contain a thread synchronization object to protect all the elements of the structure, including the message queue. The structure also contains an inter-thread signaling object that other threads use to signal when a message has been inserted into the message queue.

The message servicing threads each wait on the corresponding inter-thread signaling object. When the object is signaled the thread wakes up and enters the following cycle:

1. Lock the synchronization object
2. Pop one message from the queue
3. Unlock the synchronization object
4. Execute a code path based on the message type, which includes a callback into the CP or FP.

The above cycle is repeated until all messages have been popped from the queue. When the queue is empty the thread again enters the wait state for the synchronization object to become signaled.

It is important that the synchronization object is unlocked before the message is acted upon in case the specified action involves locking the same synchronization object.

The inter-thread signaling object is created with an automatic reset feature. This allows other threads to signal the object when the servicing thread is not waiting on it. The signal will not be lost before the servicing thread enters the wait cycle again. At that time, it will immediately wake up again and repeat the cycle listed above.

3.3.7 Memory Policy

The memory management model of the standard PDK [1] has been retained in the co-located design. In this memory management model, the caller frees memory. If data is to be accessed by the called method at any time after the method returns, the called method must copy the data into local memory before the method returns.

When the CP invokes TPI methods, the TPI copies all data from CP memory into local TPI memory before returning. The FP queue service thread invokes the appropriate FP callback method, passing in TPI resident data. Before the FP callback method returns, it copies all necessary data into FP memory. When the method returns, the TPI frees its own memory. This same procedure is mirrored when the FP invokes TPI methods.

3.3.8 Common PDK Objects

Due to the single execution environment of the co-located PDK, implementation details of the global callback lists and common component initialization are altered from their implementation in the standard PDK. The callback list is re-implemented as a list of lists. The forwarding plane instantiates a callback list and the control plane instantiates its own callback list. Each is contained as elements in the global list of callback lists. A compiler symbol, defined in component make files, is used to differentiate between the CP and FP lists during compilation and subsequent runtime callback list method invocation.

Initialization of the platform independence layer and the logging utility are each made in the FP and the CP in the standard PDK. In the co-located PDK, initialization of these two components is suppressed in the FP.

Part 4: Use Case

4 Use Case

A use case of the co-located PDK is described in this section with focus given to the TPI. The use case consists of the user adding a classifier, a queue, and a scheduler to the forwarding element. A sequence diagram illustrating the use case is shown in Figure 20. The dotted return arrows imply boundaries between threads by indicating where threads return. For example, after the CP thread calls into the TPI, eventually placing the AddClassifier message into the TPI CP queue, a TPI event is fired and the CP thread returns.

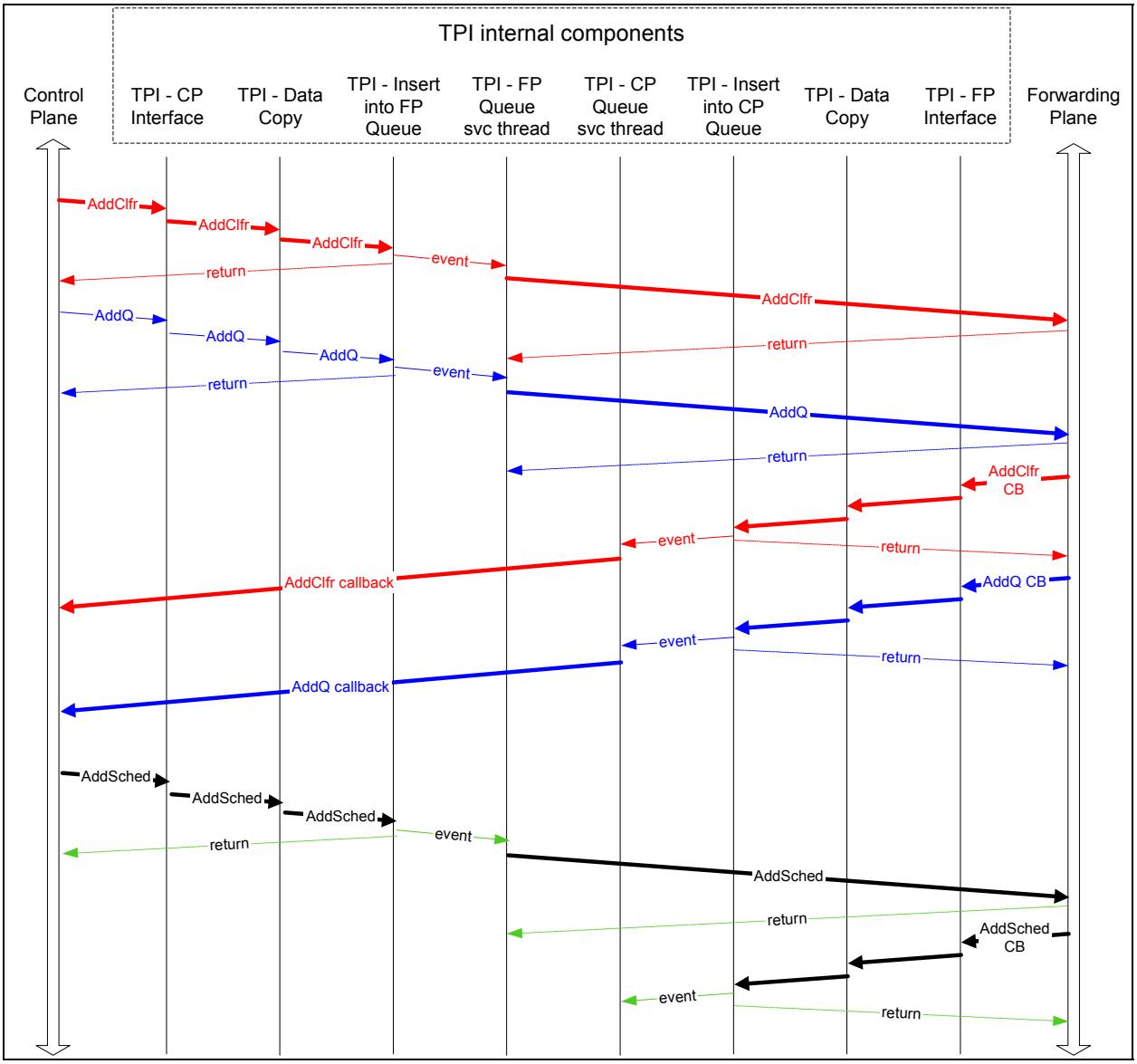

Figure 20: Sequence diagram showing addition of a classifier, a queue and a scheduler

Note the asynchronous behavior of method invocations into the FP. The forwarding plane copies method parameter data then returns. The FP then internally executes the requested operation in its

own thread and makes the appropriate callback into the TPI when completed. For example, after the TPI FP queue service thread calls the FP AddScheduler method, the FP method returns after the data copy. The FP adds the scheduler to the appropriate data plane components then calls the TPI callback method.