Interface Manager

Design Specification

Control Plane-Platform Development Kit 2.11

March 2004

intel

Contents

Revision History

Revision	Description	Date	Author
2.11	Updated for Release 2.11	March 2004	Ds Sreedhara
2.1	Updated for Release 2.1	December 2003	Ds Sreedhara
2.0	Updated for Release 2.0	August 2003	Ds Sreedhara

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 Application Program Interface (API) within the above-mentioned planes can help system vendors, Original Equipment Manufacturer (OEM), 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 APIs. 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 Interface Manager

This document provides information on the internal design of the interface management (IM) component of the CP-PDK. This document includes the description and design of the main internal data structures and algorithms used within the component.

The IM is responsible for creating, configuring and enabling various types of interfaces that are used by the internal and external components to configure system information.

This document describes the design of the IM component. It does not address the individual IM APIs implemented by this component. The intended audience for this document are developers implementing and/or maintaining the IM component, or test engineers who are performing quality analysis (QA) on the IM component.

1.2 Assumptions

Some assumptions are made to simplify the design:

- The configuration and management module (C&M) refers to the information represented by the namespace nodes. In this design of the IM, the C&M handle corresponds to the IM interface handle.
- The APIs assumes the existence of a C&M on the system. This component stores the system information.
- The APIs assumes the existence of a compliant namespace present on the system. The namespace provides handles for applications and components to access objects.

1.3 Dependencies

The IM component depends on the namespace and C&M to create and store information and the FP plug-in component for contacting FEs. The IM component must also make use of several other components:

Double Link List (DList)	DList is the double-link list designed in C. It provides multiple thread safety operations to manipulate the list.
Platform Independence Layer (PIL)	For reuse of software components for different applications, writing the modules in a high-level language is not sufficient. Differences within the operating systems on the platform and the compilers can make porting difficult. The PDK uses PIL to minimize compiler exceptions and provides an operating system independent API, reducing the porting effort.

1.4 Terminology

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

Table 1. Terminology table

Term	Description
CE	Control Element
FE	Forwarding Element
IXA	Internet eXchange Architecture
NPF	Network Processing Forum

PDK	Platform Development Kit
ATM	Asynchronous Transfer Mode
IM	Interface Management
CMM	Configuration and Management Module
NS	Namespace

1.5 References

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

Table 2. Document reference table

Reference	Description
[1]	Software Architecture Overview
[2]	Forwarding Plane Plugin API Reference
[3]	Platform Namespace Design Reference
[4]	Platform Independence Layer API Reference
[5]	API Framework Reference
[6]	Configuration and Management Design Reference

Part 2: Overview

2 Overview

The interface manager is responsible for creating, configuring, and enabling various types of interfaces that are used by the internal and external components to configure system information. Figure 1. displays the NPF PDF architecture.

Figure 1. NPF PDK architecture

The interface represents the state information as seen by the forwarding plane. It is not a cache for the control plane and it does not allow any `get * attribute` style APIs.

Part 3: Interface Management API Design

3 Interface Management API Design

3.1 Adaptation to Existing Implementation

This design is based on reusing the existing namespace and C&M module and extending it to provide the special functionality of the API, as shown in Figure 2. .

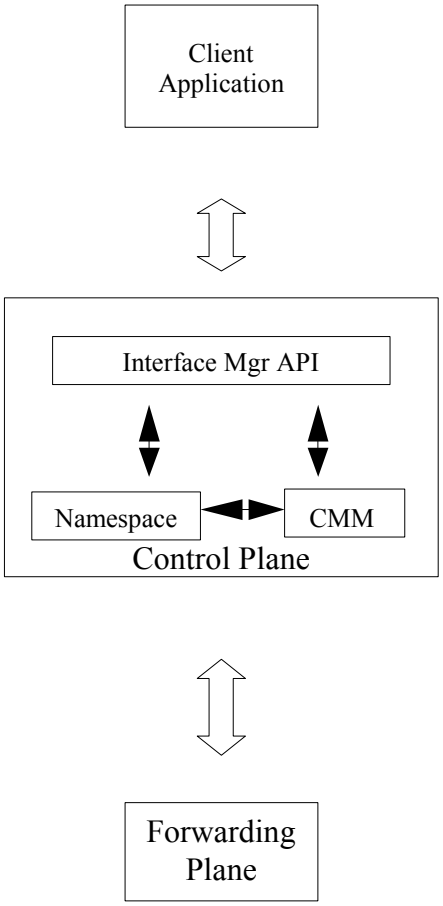

Figure 2. Interface management API interactions

The namespace creates nodes corresponding to various network entities and provides handles to access them. The information associated with the nodes is stored in the C&M module and can be accessed through the C&M API.

The design of this component would different if the NS and C&M were not already in use by the other PDK components.

The interface manager APIs are divided into the following types:

1. Registering and deregistering callbacks

2. Creation and deletion of interfaces
3. Enabling or disabling interfaces
4. Setting the various interface attributes
5. Getting statistics for various interfaces - POS, ATM, IPv4 and LAN

The first type can be implemented independently in the interface manager.

Types 2, 3, and 4 of APIs use the existing C&M to store and access information. Type 5 of APIs are not provided very easily by the C&M, so it is necessary for this module to communicate with the FP plug-in API directly to get the statistics.

In this design of the IM, the C&M handles correspond to the IM interface handles.

The IM uses the namespace to set up the interfaces and the C&M to access it. The namespace nodes need to be set up for each interface that may be added or updated. The initial namespace tree is setup during the initialization and each interface added from the IM API forms a node in this tree.

The attributes of the interfaces form the attributes of the namespace nodes. For example, the namespace represents an interface on an FE as `/System/0/Router/0/Interface/2`. The IM should create /`System/0/Router/0` at initialization and then listen for FE bind events to create the `/System/0/Router/0/FE/1`. The FE bind event also creates the ports, such as, `/System/0/Router/0/FE/1/Port/4`. The attributes of the port may be changed or updated by the application.

When an interface is added it creates `/System/0/Router/0/Interface/2`. An FE bind causes a port to be added `as /System/0/FE/1/Port/4` by the IM into the namespace, along with any other attributes configured by the application. Interfaces of type LAN, POS and ATM has port as an attribute. This port attribute is a logical port number.

Users of CP-PDK are expected to create the logical port number in the namespace as `/System/0/Router/0/IfPort/3` and associate the logical port to physical port /System/0/FE/1/Port/4 using namespace association. Users could attach LAN and POS interface below the IPv4 interface. This task can be performed using interface bind API. On interface bind, IMM find the logical port number from the LAN or POS attributes. It finds the associated physical port and performs the namespace association between IPv4 interface and physical port.

Creation of namespace nodes for each ATM Vcc is expensive, as a system can have thousands of Vcc. For the same reason it is not reasonable to store the attributes of each Vcc. IM does not store the attributes, it transparently passes the attributes to core components through FPPAPI and get the attributes from core components as needed.

Table 3. Mapping the interfaces API to various internal components

Interface Management API call	Dependent Components
NPF_IfRegister	Internally handled in IM
NPF_IfDeregister	Internally handled in IM
NPF_IfEventRegister	Internally handled in IM
NPF_IfEventDeregister	Internally handled in IM

NPF_IfCreate	Uses Namespace and CMM
NPF_IfDelete	Uses Namespace and CMM
NPF_IfBind	Uses Namespace
NPF_IfGenericStatsGet	Uses Interfaces core component via the FP Plugin
NPF_IfAttrSet	Uses Namespace and CMM
NPF_IfEnable	Uses Interfaces core component via the FP Plugin
NPF_IfDisable	Uses Interfaces core component via the FP Plugin
NPF_IfOperStatusGet	Uses Interfaces core component via the FP Plugin
NPF_IfLAN_Src_AddrSet	Uses Namespace and CMM
NPF_IfLAN_MAC_RcvAddrListAdd	Uses Namespace and CMM
NPF_IfLAN_MAC_RcvAddrListAdd	Uses Namespace and CMM
NPF_IfLAN_PromiscSet	Uses Namespace and CMM
NPF_IfLAN_PromiscClear	Uses Namespace and CMM
NPF_IfLAN_FullDuplexSet	Uses Namespace and CMM
NPF_IfLAN_FullDuplexClear	Uses Namespace and CMM
NPF_IfLAN_SpeedSet	Uses Namespace and CMM
NPF_IfLAN_FlowControlTxEnable	Uses Namespace and CMM
NPF_IfLAN_FlowControlTxDisable	Uses Namespace and CMM
NPF_IfLAN_FlowControlRxEnable	Uses Namespace and CMM
NPF_IfLAN_FlowControlRxDisable	Uses Namespace and CMM
NPF_IfIPv4AddrSet	Uses Namespace and CMM
NPF_IfIPv4MTUSet	Uses Namespace and CMM
NPF_IfIPv4FIBSet	Uses Namespace and CMM
NPF_IfATM_VccSet	Uses Namespace and CMM
NPF_IfATM_VccBind	Uses Namespace and CMM
NPF_IfATM_VccDelete	Uses Namespace and CMM

3.2 Initialization

Interface manager initialization should be called after C&M, namespace and binding and discovery (B&D) and FP plug-in. It should be done before any APIs or internal components that are interested in event notification or API callbacks in order to provide registration service.

During initialization in the co-located case, the root is setup but in the remote case, the IM must setup the namespace root and register with the CMM to use the FP plug-in callback service that it has registered for. The IM also initializes its internal lists to save callback state information from the FP plug-in and directly registers for the callback service with the FP plug-in. Since the CM already has the lists, it is recommended to use them because replication of information could potentially cause consistency issues.

3.3 Shutdown

As with initialization, IM shutdown must be called after all the internal components have deregistered the IM callback service. The IM shutdown must deregister callbacks to the CM, B&D and the FP plug-in callback. IM must then destroy all state information.

In the PDK shutdown process, the PDK manager should invoke the IM shutdown before the CM has been shutdown.

3.4 API Calls

The IM APIs are split into two different sections - synchronous and asynchronous. In synchronous IM APIs, the API gets the result almost immediately. In asynchronous IM APIs, the API returns almost immediately but it does not get the result right away. The result is called back some time later by the IM implementation. The reason for asynchronous calls is that some APIs must communicate with forwarding planes, which may be connected through the wire, and they need time for the control plane to communicate with the forwarding plane and return the result.

The synchronous calls consist of registering and deregistering callbacks.

The asynchronous calls have the following categories:

1. Creation and deletion of interfaces
2. Enabling or disabling interfaces
3. Setting the various interface attributes
4. Getting statistics for various interfaces - POS, ATM, IPv4 and LAN

The first three types of APIs use the existing C&M to store and access information. The type of APIs are not provided very easily by the C&M, so it is necessary for this module to communicate with the FP plug-in API to get the statistics.

3.4.1 Synchronous APIs

The callback register, deregister, and IM initialization/shutdown calls are all synchronous calls. The application registers for event callbacks like NPF_IF_UP, NPF_IF_DOWN and NPF_IF_COUNTER_DISCONTINUITY or API callbacks.

3.4.2 Asynchronous API Calls

There is no top-down NPF API for events, except the register and deregister. For the normal NPF asynchronous API calls, the user program must use the unique PDK callback handle. The PDK callback handle is created at registration along with the NPF interface handle, user correlator, error reporting strategy, and request user data to make the subsequent API call. After IM receives the NPF API calls, it validates the input arguments, such as, callback handle and NPF interface handle and type. Then it calls the FP plug-in or the C&M. It also creates and stores the callback state, released after the receiving the callbacks and IM has returned them to the client application.

3.4.2.1 Setting the Interface Attributes

The IM exposes one or multiple calls to SET properties of an interface. For a small change, it is recommended to use a set of single property, such as, `NPF_IfLAN_PromiscSet` or `NPF_IfLAN_SpeedSet`. Using `NPF_IfCreateandSet` or `NPF_IfAttrSet` may be easier to initialize the interface. The user program may also want to assign one value at a time. These calls map to the C&M that allows setting one property at a time.

The IM must save state and make multiple calls to the C&M. Once all the callbacks from the C&M have been received, it compiles the callbacks to make a single callback into the client application. This operation is done in the callback thread of the C&M. In the current implementation, most of the IM APIs directly calls the FP plug-in APIs and updates the C&M data structures on the response from FP plug-in APIs.

3.4.2.2 Setting an Array of Information

The IM functions take an array of information elements and the length of the array but the C&M APIs takes one information element. The IM shall call the C&M function multiple times, one per information element. The IM shall provide unique correlator value for each C&M function call and shall store the state of each C&M call. On receiving asynchronous response, the IM shall update the return status. When all responses are received, send a single response back to user of IM. In the current implementation, most of the IM APIs directly call the FP plug-in APIs and update the CMM data structures on the response from FP plug-in APIs.

3.4.3 Asynchronous API Completion Callback

After the user program makes asynchronous API calls, the commands are sent to the C&M or forwarding plane and a response later returns to the IM from the C&M or FP plug-in. When the C&M or FP plug-in calls back the IM, the callback is executed in a thread different from the original thread that made the request.

The call and the callback refer to the callback state stored in a callback info list. It is recommended to put the callback info lists into IM global data and synchronize it with a lock. When all callbacks from the C&M that relate to a single callback in the IM, the state associated with it is cleaned out and a compiled callback is returned to the IM. The FP plug-in always calls back the IM and C&M API is with the verbosity set to always callback the IM. This prevents a memory leak in the IM due to callback state stored while making a call but never cleared up when a callback completes.

The response data is dynamically created by the FP plug-in and passed into the callback functions to the user program. After all the callback return, the FP plug-in deletes the response data immediately. It is a responsibility of the user program or internal component to make a copy of the response data, if needed after exiting the callback context.

3.4.3.1 Example: General SET Algorithm

The general callback algorithm for a set API is listed as follows:

5. The C&M or FP plug-in is invoked with the interface handle and its attributes to be set.
6. The C&M or FP plug-in invokes the callback function provided by IM registered at IM initialization time. The function parameter contains feid, correlator and response data.
7. In the callback functions, IM validates the correlator and feid.
8. If the validation passes, the IM may need to copy the callback data into the IM response into the correct data object.
9. At each callback, the state is checked to see if any more callbacks are expected. When all the callbacks completed, the callback to the client application is invoked

3.5 API extensions

The current NPF specification only contains interfaces of types: LAN, IPv4 and ATM. This API needs to be extended to add support for creation and deletion of IPv6 interfaces and to get statistics for them in the following generic interfaces management API.

Table 4. API extension table

APIs	Dependent Components
NPF_IfCreate	Uses Namespace and CMM
NPF_IfDelete	Uses Namespace and CMM
NPF_IfBind	Uses Namespace
NPF_IfGenericStatsGet	Uses Interfaces core component via the FP Plugin
NPF_IfVccStatsGet	Uses Interfaces core component via the FP Plugin
NPF_IfAttrSet	Uses Namespace and CMM
NPF_IfEnable	Uses Interfaces core component via the FP Plugin
NPF_IfDisable	Uses Interfaces core component via the FP Plugin

NPF_IfOperStatusGet	Uses Interfaces core component via the FP Plugin

The data structures `NPF_IfCallbackType`, `NPF_IfAsyncResponse, NPF_IfType` and `NPF_IfGeneric_t` should be extended to add support for IPv6 and ATM OAM. For more information on this, please refer to the Interfaces API document.

3.5.1 IPv6

It must also provide support the following IPv6 specific APIs, which are not part of the NPF specification.

Table 5. IPv6-specific API table

APIs	Dependent Components
NPF_IfIPv6AddrAdd	Uses Namespace and CMM
NPF_IfIPv6AddrDelete	Uses Namespace and CMM
NPF_IfIPv6UCZoneSet	Uses Namespace and CMM
NPF_IfIPv6MTUSet	Uses Namespace and CMM
NPF_IfIPv6FIBSet	Uses Namespace and CMM
NPF_IfIPv6inv4DstAddrSet	Uses Namespace and CMM
NPF_IfIPv6inv4SrcAddrSet	Uses Namespace and CMM
NPF_IfIPv6inv4TTLSet	Uses Namespace and CMM
NPF_IfIPv6inv4DSCPSet	Uses Namespace and CMM
NPF_IfICMPv6RateLimitSet	Uses Namespace and CMM

3.5.2 ATM OAM

It must also provide support the following ATM OAM specific APIs, which are not part of the NPF specification.

Table 6. ATM OAM-specific API table

APIs	Dependent Components
NPF_IfATMOAMSetStatistics	Uses Namespace and CMM
NPF_IfATMOAMSetContinuityCheck	Uses Namespace and CMM
NPF_IfATMOAMSetLoopback	Uses Namespace and CMM
NPF_IfATMOAMSetAlarm	Uses Namespace and CMM

NPF_IfATMOAMSetPM	Uses Namespace and CMM
NPF_IfATMOAMSetConnectionPointType	Uses Namespace and CMM
NPF_IfATMOAMGetProperties	Uses Namespace and CMM

Part 4: Runtime Interactions

4 Runtime Interactions

Figure 3. displays the sequence diagrams for the synchronous APIs.

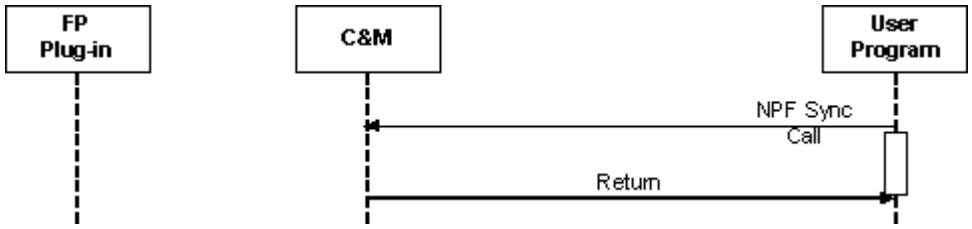

Figure 3. Sequence diagram for synchronous APIs

Figure 4. displays the sequence diagrams for the asynchronous APIs.

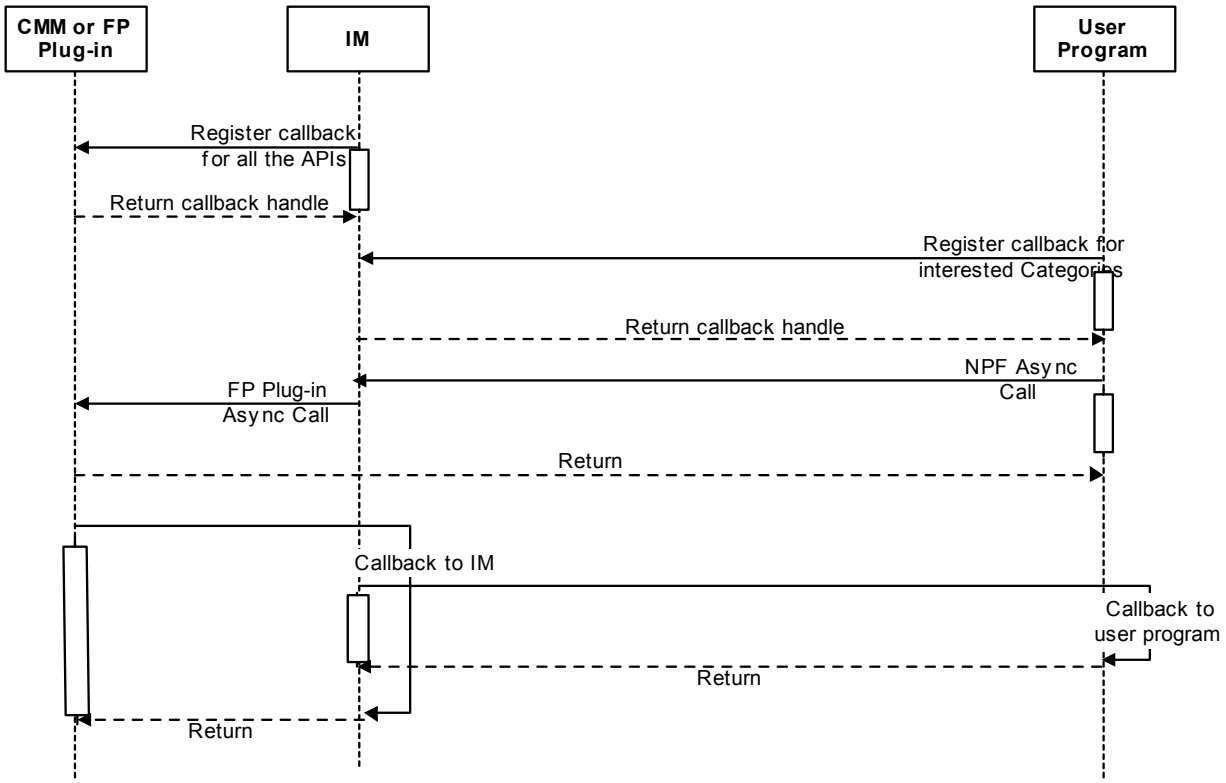

Figure 4. Sequence diagram for asynchronous APIs

Figure 5. displays the sequence diagrams for the event notification.

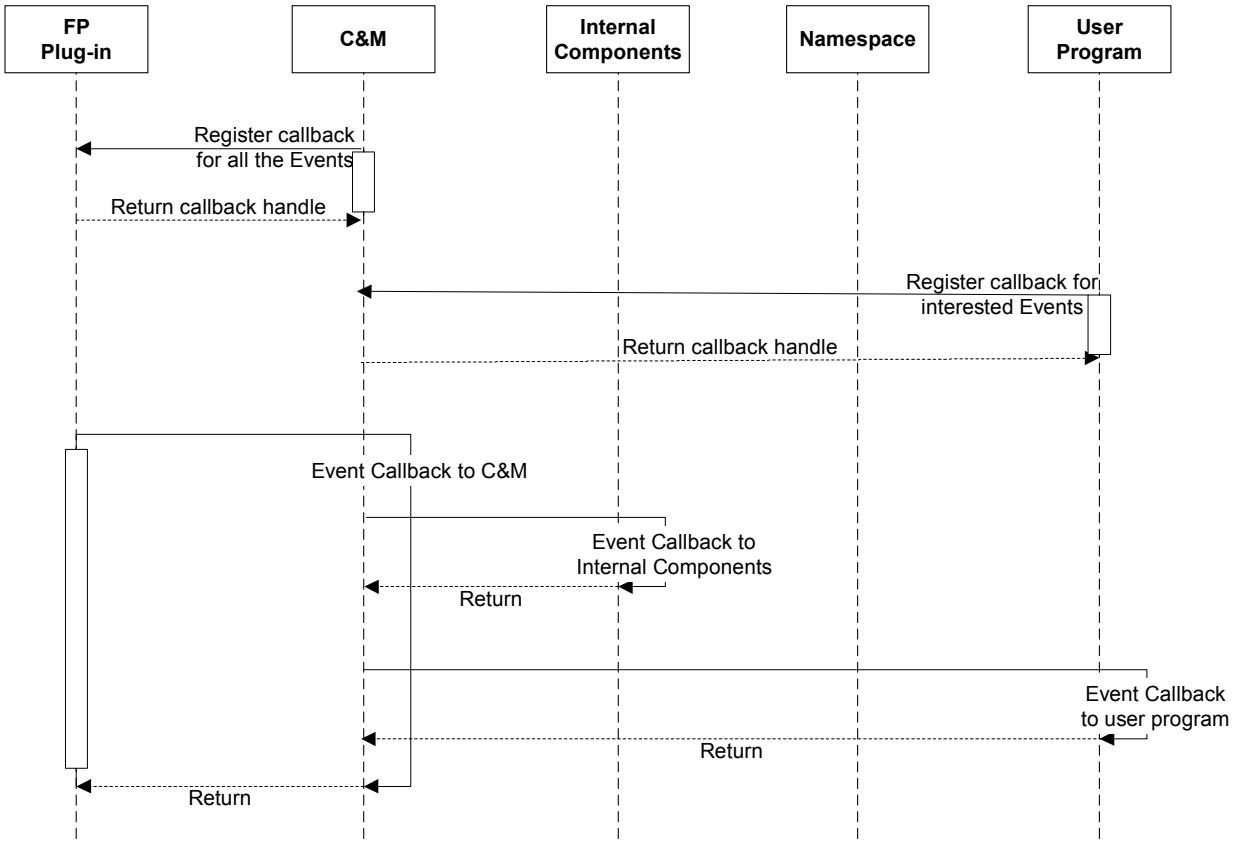

Figure 5. Sequence diagram for event notification

Part 5: Implementation details

5 Implementation details

5.1 Data structures

A list of active application requests is kept for processing callbacks. The structure is as follows:

```
typedef struct {
FPPI_FEID           id;
NPF_boolean_t       ok; /* Set to TRUE when req = resp */
uint32_t            req; /* No. of requests for this FE id */
uint32_t            resp; /* No. of responses for this FE id */
} feReq_t;
```

```
typedef struct {
NPF_IfCallbackType_t     type;          /* callback type */
NPF_callbackHandle_t     cbh;           /* callback handle */
NPF_userContext_t        context;       /* user context */
NPF_correlator_t         corr;          /* user correlator */
NPF_errorReporting_t     report;        /* user error report */
uint32_t           no;    /* No. of requests generated */
uint32_t           fpReq;         /* No. of FP API requests to be mabde */
uint32_t           fpResp;        /* No. of FP responses generated */
uint32_t           resp;          /* No. of responses generated for this request */
NPF_IfCallbackData_t     data;          /* user callback data */
NPF_DS_CallbackFunc_t    callback;      /* user registered callback function */
NPF_IfHandle_t     ihandle;      /* interface handle */
NPF_error_t        error;         /* error code */
If_t               ifc;           /* interface for which request made  */
uint32_t           id;            /* last FE id this request sent */
void         *asyncResp[IF_MAX_NO];
uint32_t     ecount;
void         events[IF_MAX_NO];
NPF_boolean_t      ok;            /* Set to TRUE if all FE of this interface responded
*/
feReq_t      fe[MAXFENUM];  /* Request per FE */
      } request_t;
```

A linked list is used to keep a number of these structures. Whenever the application makes a request, a new callback structure is allocated and added to the list. This request is sent to the FP. When all asynchronous responses associated with the call are received from the forwarding plane, the list is searched to retrieve the appropriate callback structure and, with the information in the

structure, appropriate callback of the application is invoked. After the callback is invoked, the structure is removed from the list and allocated memory is freed.

5.2 Memory Allocation

In the PDK IM implementation, the same data is passed down to the FP plug-in. After the callback, IM copies this data into its callback data structure. The callback data structure contains an array of response pointers. The IM needs to make a copy of the response data before passing it to the user. This step needs to be performed for each registered user.

5.3 Memory Free

Since the NPF user is running in a thread different from FPPAPI callback thread, IM or C&M shall not free the allocated memory as soon as it sends the data to the other thread using pipe/mail box. For maintaining data integrity, the user shall free memory after use.

5.4 Threading and Synchronization

The IM API component does not create any new threads. The functions are called by the application and the callbacks may be in different thread contexts. For this reason, the list that holds callback states must be locked when they are being accessed. The IM uses `PIL_EnterCriticalSection` and `PIL_LeaveCriticalSection`.

Part 6: Pseudo Code

6 Pseudo Code

6.1 Initialization

The following is the pseudo code for initializing the interface manager:

```
NPF_ERROR_t im_init(void){
    // int CB state list
    npf_list_init(&CBList, PIL_FreeMemory );

    //register CMM cb
    npf_category_register();
    //register fpplugin cb
    fppapi_deregister_cb();
        return NPF_SUCCESS;
}
```

6.2 Shutdown

The following is the pseudo code for shutting down the interface manager:

```
NPF_ERROR_t im_destroy(void)
{
    for each registered cb fppapi_deregister_cb();
    for each registered CMM cb npf_category_deregister();
    // destroy the callback state list
    npf_list_destroy(&CBList);
    return NPF_SUCCESS;
}
```

6.3 Asynchronous API Calls

The following is the pseudo code for assigning the port IP address:

```
NPF_RET NPF_IfIPv4AddrSet (NPF_callbackHandle    cbid,
                           NPF_correlator_t  correlator,
                           NPF_errorReporting_t   verbosity,
                           NPF_unir32_t     n_hanldes,
                           NPF_IfHandle_t *ifHandleArray
                           NPF_IIIPv4NetAddr_t *ipaddr  )
{
    // check if the CB exists
    if (cm_isCBIdExist(cbid) == -1)
```

```
        return NPF_INVALID_PARAMETERS;
    // create callback state info  IM_CBState_t;
    cbState = PIL_MemoryAllocate(sizeof(IM_CBState_t), 0);
for (int I = 0; I < n_handles; I++)
{
            //  create fppi_correaltorState_t for each entry
       corrState =
PIL_MemoryAllocate(sizeof(fppi_correaltorState_t), 0);
       npf_list_push_back(cbState->fppi_CorrelatorStateList,
corrState);
       npf_set_ipaddr(imcbHandle,
                    fppi_correlatorState.fppi_corr,
                    verbosity, ifHandleArray[I], ipaddr[I]);
}
    npf_list_push_back(cbStateList, cbState);
}
```

6.4 API Callback

The following is the pseudo code for the callback on setting the port IP address:

```
Void IfAPICB (NPF_userContext       cmcontext,
            NPF_correlator    cmcorrelator,
            NPF_CallbackData        data    )
{
      //  Find cbstate
      FindCBStatefromcontext(cbStateList, context, &cbstate)
      // Find corrState
      FindCorrStatefromcorrelator(
                cbState->corrStateList, correlator, &corrState)
//  Update state of correlator
      corrState->done = 1;
      //  Update callback state
      UpdateCallbackState(cbState, data);
      //Check to see if all callbacks completed
     itr = npf_list_itrCreate(cbState->corrStateList);
     allDone = 1;
     for(itr = npf_list_first(); itr != -1; itr =
npf_list_itrNext())
     {
           corrState = npf_list_itrGetData(itr);
           if (corrState->done !=  1)
           {
              allDone = 0; break;
           }
    }
```

```
    if(allDone)
    {
     // GetIMCallbackfunc by appcorrelator
     IMCallbackFunc = GetIMCallbackfunc(cbState.correlator)
     // Call the IM Callback
     NPF_IMCallbackFunc(cbState.context, cbState.correlator,
cbstate.ifcallbackData)
     // free the temporary data and c&m correlator
     PIL_FreeMemory(corrState);
     PIL_FreeMemory(cbState);

    }
    return;
}
```