Configuration and Management

Design Specification

Control Plane-Platform Development Kit 2.11

March 2004

Contents

Configuration and Management i

Contents iii

Part 1: Overview 5

1 Overview 7

1.1 Purpose 7
1.2 Scope 7
1.3 Configuration and&Management Component Overview 8
1.4 Assumptions 8
1.5 Dependencies 9
1.6 Terminology 9
1.7 References 9

Part 2: Configuration and Management API Design 11

2 Configuration and Management API Design 13

2.1 Object Lists 13
2.2 Initialization 16
2.3 Shutdown 16
2.4 API Calls 16
2.4.1 Synchronous APIs 17
2.4.2 Asynchronous APIs 17
2.4.3 Snapshot Get and Individual Set 20
2.5 Asynchronous API Completion Callback 20
2.5.1 General GET/SET Algorithm 20
2.5.2 Shutdown Algorithm 21
2.6 Event Notification Callback 22
2.7 Helper Functions 23

Part 3: Runtime Interactions 25

3 Runtime Interactions 27

Part 4: Memory Allocation and Locking 31

4 Memory Allocation and Locking 33

4.1 Memory Allocation 33
4.2 Locks 33

Part 5: Pseudo Code 35

5 Pseudo Code 37

5.1 Initialization 37
5.2 Shutdown 37
5.3 Registration & Deregistration 37
5.4 Asynchronous API Calls 38
5.5 API Callbacks 39

Revision History

Revision	Description	Date	Author
2.11	Updated for release 2.11	March 2004	Shailesh Suman
2.1	Updated for release 2.1	December 2003	Shailesh Suman
2.0	Updated for Release 2.0	August 2003	Shailesh Suman

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 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 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. 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 (CP-PDK) are based on the NPF APIs. For more information about NPF, refer to http://www.npforum.org/.

1.1 Purpose

This document specifies the internal design of the configuration and management (C&M) component of the CP-PDK. This includes the description and design of the main internal data structures as well as algorithms used within the component.

1.2 Scope

This document describes the design of the C&M component. It does not address the individual C&M APIs implemented by this component. For API definitions refer to C&M API Reference [2]. The intended audience for this document is developers implementing and/or maintaining the C&M component, or test engineers who are performing Quality Analysis (QA) on the C&M component.

1.3 Configuration and&Management Component Overview

The C&M module is split into two logical sub-components, the C&M API implementation module and the callback and event module. The implementation module is the code that implements the C&M APIs. This sub-component is responsible for validating input, constructing the necessary object, and invoking the FP plug-in API to send requests to the forwarding planes. The callback & event module is responsible for implementing the callback functions invoked by the forwarding plane (FP) plug-in. It modifies the system data and invokes the user program's callback to return the response data. These two modules are usually running in the different contexts, so they are split into two different sub-components. Both modules are discussed in more detail in the following sections.

Figure 1 shows the configuration and management sub-components.

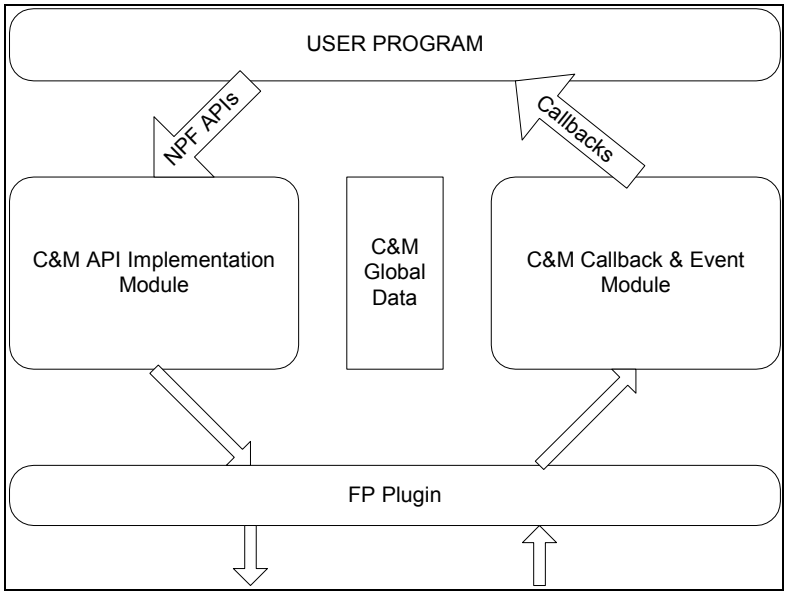

Figure 1: Configuration and management sub-components

1.4 Assumptions

The following assumptions have been made to simplify the design. Some assumptions are made to simplify the design:

- The APIs assumes the existence of a compliant namespace present on the system. This namespace provides handles for applications and components to access objects.
- The PDK must handle the packet flows to traverse over multiple blades inside a single ON router. The application running on top of the APIs is hidden from the fact of multiple forwarding planes.

Refer to the API Framework Reference [6] for details of the callback mechanisms and parameters.

1.5 Dependencies

The C&M component depends on the FP plug-in component for contacting the forwarding element (FE). In addition, the C&M component must also make use of several other components. Table 1 lists various components used by C&M component

Table 1. Component dependency table

Terms	Expansion
Callback Library	A central PDK callback library is provided to all the internal components for register, deregister and retrieve callback functions. The external user programs should not use this, but should go through the existing PDK APIs for registering/deregistering.
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)	In reusing software components for different applications, writing the modules in a high level language is not sufficient for easy re-use. Differences in the operating system on the platform and the compilers can make porting difficult. The PDK uses PIL to minimize compiler exceptions and provide an operating system independent API, greatly reducing the porting effort.

1.6 Terminology

Table 2 lists terms used in this document and provides an expansion for each term.

Table 2. Terminology table

Terms	Expansion
CE	Control Element
FE	Forwarding Element
IXA	Internet eXchange Architecture
NPF	Network Processing Forum
PDK	Platform Development Kit
ATM	Asynchronous Transfer Mode
MTU	Maximum Transmission Unit

1.7 References

Table 3 lists documents referenced in, or related to, this document.

Table 3. Reference table

Terms	Expansion
[1]	Software Architecture Overview
[2]	Configuration and Management API Reference
[3]	Forwarding Plane plug-in API Reference
[4]	Platform Namespace Design Reference
[5]	Platform Independence Layer API Reference
[6]	API Framework Reference

Part 2: Configuration and Management API Design

2 Configuration and Management API Design

2.1 Object Lists

The configuration and management (C&M) module maintains several objects lists inside the C&M scope, such as, system, virtual router, FE, port and interface. These object lists can only be accessed by C&M. The other internal components or external user programs cannot directly access the data, as they always operate the data through the NPF object handle. The object handle is stored in the namespace component, and other components should request the object handle first before operating on the object via C&M. Each object data structure contains an NPF_HANDLE field to identify itself. C&M searches the object list to match the object handle and retrieve the object data to do perform the operation.

The object data structures that follow are not the ones returned to the user program when it makes a GET properties call. Instead of the original object data structure, C&M returns the subset of the original data structure that does not contain the internal information, for example, the reference count and related handles.

NPF_DATATYPE

```
typedef enum {  NPF_SYSTEM,
                NPF_IP_ROUTER,
                NPF_ATM_VIRTUAL_DEVICE,
                NPF_FE,
                NPF_PORT,
                NPF_ATM_LINK,
                NPF_INTERFACE } NPF_DATATYPE;
```

PDK_SYSTEM

```
Definition: typedef struct pdk_system_tag {
        NPF_DATATYPE    type;
        NPF_HANDLE      handle;
        char            name[128];
        char            desc[256];
        int             num_FEs;
        int             refcount;
} PDK_SYSTEM_t
```

PDK_IP_ROUTER

```
Definition: typedef struct pdk_ip_router_tag {
        NPF_DATATYPE    type;
        NPF_HANDLE      handle;
        char            name[128];
        char            desc[256];
        int             refcount;
} PDK_IP_ROUTER_t
```

PDK_ATM_VIRTUAL_DEVICE

```
Definition: typedef struct pdk_atm_virtual_device_tag {
        NPF_DATATYPE    type;
        NPF_HANDLE      handle;
        char            name[128];
        char            desc[256];
```

```
        int             refcount;
} PDK_ATM_VIRTUAL_DEVICE_t
```

PDK_FE

```
Definition: typedef struct pdk_fe_tag {
        NPF_DATATYPE    type;
        NPF_HANDLE      handle;
        char            desc[256];
        char            name[128];
        time_t          stamp;
        uint32_t        id;
        int             num_ports;
        int             status;
        HWADDR          hwaddr;
        IPADDR          ipaddr;
        int             refcount;
} PDK_FE_t
```

PDK_INTERFACE

```
Definition: typedef struct pdk_interface_tag {
        NPF_DATATYPE    type;
        NPF_HANDLE      handle;
        char            desc[256];
        char            name[128];
        int             status;
        IPADDR          ipaddr;
        int             refcount;
} PDK_INTERFACE_t
```

PDK_PORT

```
Definition: typedef struct pdk_port_tag {
        NPF_DATATYPE    type;
        NPF_HANDLE      handle;
        NPF_HANDLE      fehandle;
        NPF_HANDLE      ifhandle;
        uint32_t        id;
        time_t          stamp;
        int             edge;
        DList           peerList;
        int             status;
        HWADDR          hwaddr;
        int             refcount;
} PDK_PORT_t
```

PDK_ATM_LINK

```
Definition: typedef struct pdk_atm_link_tag {
        NPF_DATATYPE    type;
        NPF_HANDLE      handle;
        char            desc[256];
        char            name[128];
        int             status;
        int             refcount;
} PDK_ATM_LINK_t
```

As described previously, C&M also defines an additional external object data structure that contains only a subset of the original data structure. For simplification, just rename the PDK to NPF, and remove the fields in the structure that are not in bold font. For example, the external object data structure for PDK_FE would be as follows:

NPF_FE

```
Definition: typedef struct npf_fe_tag {
        char            desc[256];
        char            name[128];
        time_t          stamp;
        uint32_t        id;
        int             num_ports;
        int             status;
        HWADDR          hwaddr;
        IPADDR          ipaddr;
} NPF_FE_t
```

The external data structures are listed in the configuration and management API Reference.

The C&M also provides internal APIs to the namespace component to create and destroy the object in the object list.

cm_newDataByType

Signature: `int cm_newDataByType ( NPF_DATATYPE, NPF_HANDLE * )`

Description: Create a memory space for such data type and assign the default value

Parameters In: `NPF_DATATYPE`

Parameters Out: The pointer to the C&M-generated NPF data handle.

Return Values: NPF_SUCCESS, NPF_OUT_OF_MEMORY or NPF_INVALID_PARAMETERS

cm_delDataByHandle

Signature: `int cm_delDataByHandle ( NPF_HANDLE )`

Description: Free the memory space pointed to by the object handle.

Parameters In: NPF object handle

Parameters Out: None

Return Values: NPF_SUCCESS or NPF_INVALID_PARAMETERS

cm_getFEHandleById

Signature: `int cm_getFEHandleById ( uint32_t, NPF_HANDLE * )`

Description: Get FE data handle from the FE Id.

Parameters In: FE Id

Parameters Out: NPF object handle

Return Values: NPF_SUCCESS or NPF_INVALID_PARAMETERS

2.2 Initialization

C&M initialization should be called before all the other internal components, except B&D and forwarding plane plug-in, that are interested in event notification or API callbacks in order to provide registration service.

In the initialization, C&M must initialize all object lists, register the FP plug-in callback service, and initialize the list of registered FP plug-in callback handles.

Before all lists are initialized, C&M must register the callback service from the FP plug-in for all the NPF APIs. For simplification, we use the enumerator of the FP plug-in supported callback type (such as FPPI_L3SetProperties) as the C&M context to register the FP plug-in callback service. The FP plug-in then returns the callback handle for each registration. C&M must store it in a list so that when the corresponding API is called from the user program, C&M can locate the correct FP plug-in callback handle by matching the type of the API call. The structure is as follows:

```
typedef struct cmRegFppCBId_tag {
    FPPI_CBTYPE  fppapi;      /* Enum for all the FPPI supported
CBTYPE */
    NPF_CBHANDLE fppcbid;     /* Callback handle assigned by FP
plug-in */
} CMREGFPPCBID_t;
```

2.3 Shutdown

With initialization, the C&M shutdown must be called after all the internal components have deregistered the C&M callback service. The C&M shutdown must walk through the `CMREGFPPCBID_t` list to deregister all FP plug-in callback services by passing the FP plug-in callback handle to the FP plug-in deregister call. C&M must then destroy the `CMREGFPPCBID_t` list, as well as every object list.

In the PDK shutdown process, the PDK manager should invoke the C&M All FE Shutdown API first, and then within the success callback, the PDK Manager must shut down each component in the proper order. So, when C&M shutdown occurs there should be no FE/Port/Interface objects in the object list.

2.4 API Calls

The C&M APIs are split into two different sections, one synchronous and the other asynchronous. In the former case synchronous section, the API gets the result immediately without blocking. In the asynchronous section, though the API still returns immediately without blocking, it does not get the result right away. The result is called back some time later by the C&M 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.

We can define the PDK internal data can be defined into four different categories:

1. The data collected from FEs/ports/interfaces that cannot be changed, such as, number of interfaces.
2. The data collected from FEs/ports/interfaces that can be changed, such as, Maximum Transmission Unit (MTU) and IP address.
3. The data that is changed dynamically, such, as port statistics.
4. The data that is created by the application, such, as routing entry).

Since types 1 and 2 do not change as frequently as types 3 and 4, and they are under the control of the PDK in general as well as the user application, the PDK can cache the data of type 1 and 2 in the C&M. Whenever a user application makes a GET request, the C&M then returns the cached data immediately without any callback, so those APIs are synchronous calls.

For the SET request of types 2, 3, and 4, it is necessary for C&M to make the change or query the current snapshot in the forwarding plane, so those APIs are asynchronous calls.

2.4.1 Synchronous APIs

The GET request of types 1 and 2, as well as the callback register, deregister and C&M initialization/shutdown calls are all synchronous calls. The examples are Get FE properties, get interface properties, and Get port properties. When C&M gets these API calls, except register/deregister service and C&M initialization/shutdown, it first validates the input parameters and then searches for the correct information in the cached data. C&M always makes a copy of the cached data for the user program. The user program must pass in the pointer to its data structure when making these API calls. It does not return the pointer of the cached data to the user program directly. By doing this we avoid the user program destroying the internal PDK cached data.

2.4.2 Asynchronous APIs

As describe previously, the SET request of types 2, 3, and 4 are asynchronous calls. Most C&M APIs fall into this category.

2.4.2.1 Callback Registration

A user program must register the callback service for each category in which it is interested, such as, event notification or FE, IP, ICMP or interface management. It is necessary for C&M to keep track of the registered callback subject. In the implementation there is a central callback service table for looking up the correct callback functions. The table is constructed as an array of link lists. The size of the array is fixed to a pre-defined maximum length and the length of each link list can vary. Whenever a new user application registers a callback service, a new list element is added to the correct row in the table, and a list element is removed at de-register time. The callback table is organized by category.

Figure 2: Callback table illustrates the callback table, and descriptions of the structures follow the figure.

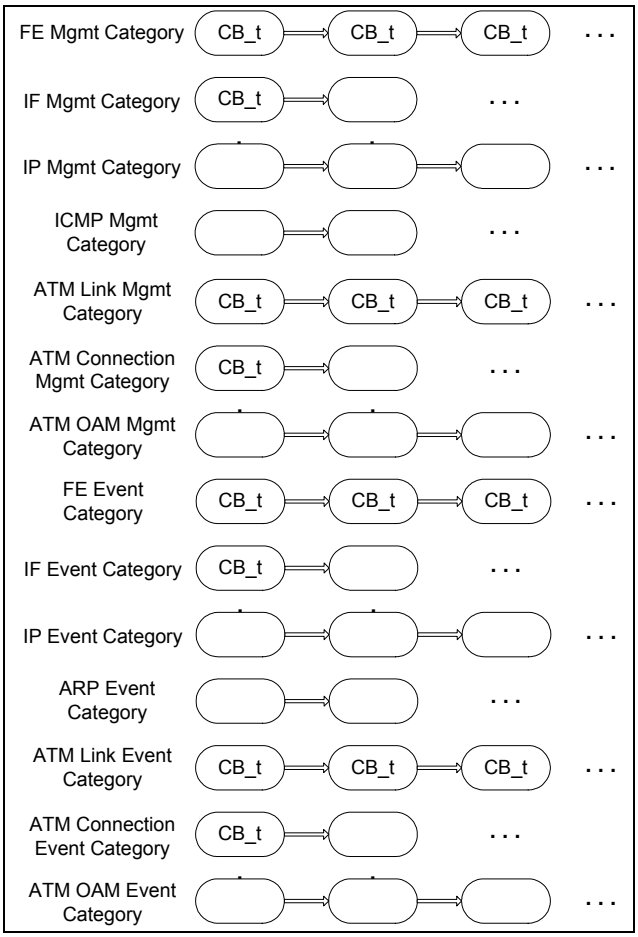

Figure 2: Callback table

```
typedef struct CB_tag {
        NPF_CBHANDLE            cbid;
        NPF_USERCONTEXT         context;
        NPF_CM_CBCAT            category;
        NPF_GENERAL_CBFUNC      func;
} CB_t;
typedef enum {  NPF_CAT_FE = 0,
                NPF_CAT_IF,
                NPF_CAT_IP,
                NPF_CAT_ICMP,
                NPF_CAT_IPv4,
                NPF_CAT_ATM_LINK,
                NPF_CAT_ATM_CONN,
                NPF_CAT_OAM,
                NPF_EVT_FE      } NPF_CM_CBCAT;
```

Whenever the user program registers a category or event notification callback, a new `CB_t` is appended at to the corresponding row in the category callback table. The `CB_t` structure contains the user-provided context, user-provided callback function, and the unique C&M-assigned callback handle. C&M keeps a global handle index number, which always increases by one whenever a new registration occurs.

2.4.2.2 Internal Callback Registration

External user programs need to register the asynchronous API or event service. Some of the internal components may be also interested in some services. For example, all internal components should register the FE unbind event, and binding and & discovery components should must be interested in setting some properties in FE category.

Whenever the callback happens, internal components should be notified first, in a well-defined order, and then the user program should be notified. Most of the internal components do perform the service registration at PDK initialization time, and deregistration at PDK shutdown time. Because Since the callback is in first-in-first-served order and starts from the internal components, the internal component callback is invoked prior to the user program callback.

For this reason, the central callback service provides two tables: one for the external user program and another for internal components. There is an internal category register/deregister API only for internal components. If the internal components want to register the service in runtime after PDK initialization has been called, the internal registration calls insert the `CB_t` into the internal callback table.

2.4.2.3 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. This handle along with the NPF object handle, user correlator, verbosity, and request user data to make the subsequent API calls in the same category. After C&M receives the NPF API calls, it first validates the input arguments, such as callback handle and NPF object handle, and then searches the corresponding FP plug-in callback handle in the FP plug-in registration list. CMREGFPPCBID_t, which stands for C&M registered FP plug-in callback ID list and is constructed at C&M initialization time, based on the type of the API call. C&M then dynamically allocates a C&M correlator, which contains the following information:

```
typedef struct cmCorrelator_tag {
        NPF_HANDLE      handle;     /* Handle to NPF Object */
        NPF_CBHANDLE    cbid;       /* Handle to callback category
*/
        NPF_CORRELATOR  correlator; /* User provided correlator */
        NPF_VERBOSITY   verbosity;  /* User provided verbosity */
        NPF_DATA        data;       /* User provided request data
*/
} *CMCORRELATOR_t;
```

As with synchronous calls, the PDK always makes a copy of the data whenever it needs to communicate with the user program. In this case, the PDK must dynamically construct data and copy the user program requested data to this internal memory space, putting its pointer into the `CMCORRELATOR_t` data structure shown previously.

C&M can call the corresponding FP plug-in API along with the FP plug-in callback handle, random C&M context, and newly constructed `CMCORRELATOR_t`. All the FP plug-in APIs are asynchronous calls, so they return immediately based on the result of the FP plug-in APIs.

Note: The internal data, user program pass-in arguments, and `CMCORRELATOR_t`, do not explicitly store globally in the C&M. They simply dynamically allocate and pass the pointer into the FP plug-in APIs as a correlator without keeping a variable in C&M. C&M can retrieve those hidden memory spaces after the FP plug-in calls back and needs to delete the dynamically allocated memory before it returns to the user program. By doing this, C&M does not need to keep

track of the current pending API calls, simplifying the complexity. The following section provides further detail.

2.4.3 Snapshot Get and Individual Set

For reducing the size of the APIs, C&M uses only one call for GET and multiple calls for SET. For example, in the interface part there is only one Get interface properties instead of multiple calls such as Get MTU and Get SPEED. All the interface information is constructed in an interface structure and passed back to the user program. There are multiple APIs for Set MTU, Set SPEED, and Set Promiscuous.

Another reason to do this is that the user program may want to get the snapshot of the current statistics or properties information instead of getting properties one by one, but want to assign only one value at a time.

2.5 Asynchronous API Completion Callback

After the user program makes asynchronous API calls, the commands are sent to the forwarding plane and a response later returns to the FP plug-in in the control plane. The FP plug-in then calls back. The callback is executed in the separate FE receiving thread, which might be different from the original thread that made the request. There is a need to put the callback tables and object lists into C&M global data and them with a lock.

2.5.1 General GET/SET Algorithm

The general callback algorithm for a GET or SET API is listed as follows, starting with the FP plug-in:

1. The FP plug-in invokes the callback function provided by C&M, registered at C&M initialization time. The function parameter contains FE ID, `CMCONTEXT`, `CMCORRELATOR`, and response data.
2. In the callback functions, C&M first validates the `CMCONTEXT` and `CMCORRELATOR`. Since we use the FP plug-in supported callback type enumerator as the C&M context is used, it is easy to validate by simply comparing the value.
3. If the validation passes, then C&M may need to copy the data in the C&M correlator into the correct data object. For example, if the request was SET Port IP Address, C&M needs to wait for the success message from the FP plug-in to assign the IP address value into the system data before making the FP plug-in API calls. C&M already dynamically allocated a data space in the C&M correlator and copied in the user-requested IP address. When it calls back, C&M must copy the value of the data in the correlator into the real memory location of the Port's IP address, since it was already allocated at the time the port object was created.
4. C&M can now search the category callback table by matching the callback ID stored in the C&M correlator, locating the correct user program callback function, and passing in the user context, user correlator, user verbosity, NPF object handle, and

the response data. The order of the callback functions is more complicated if related events must be triggered.

5. Before exiting the C&M callback function, it is important to free the C&M correlator and data inside the C&M correlator that was dynamically allocated before C&M makes the FP plug-in calls.

Note There is no verbosity level between C&M and the FP plug-in. The FP plug-in always calls back the response whether successful or not. That means the verbosity level is always ACK. By doing this, C&M does not miss any dynamically allocated C&M correlator and data and, avoiding avoids potential memory leaks.

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

2.5.1.1 Events after Completing Callbacks

In the control plane and forwarding plane separation design of CP-PDK, the control plane controls the forwarding plane. Most forwarding plane events should be triggered after a control plane command. Only a few events originally come from forwarding plane even without any trigger from control plane. For example, port MTU change events occur after the control plane makes a SET port MTU command, while forwarding plane-driven events include port up, port down, FE bind, FE unbind, and ARP change.

In the implementation of a C&M completion callback, it must first call back the internal components who have already registered the related events, then call back the user program that made the API call, then the event callback in the user program that registered the related events.

2.5.2 Shutdown Algorithm

This section provides information on the shutdown flow of an FE or port.

2.5.2.1 Single FE/Port Shutdown

Shutting down a single FE or Port is an asynchronous operation. Every component that deals with the FE or Port object should register for an FE un-bind or Port down event. The completion callback triggers the FE un-bind or Port down when the FE or Port has been successfully shut down.

Note There is no FE shutdown event, only FE un-bind event is there. The FE shutdown means the NPF API call to administratively shut down an FE, not the event.

Some notes for shutdown command has following features:

1. An FE/Port can only be administratively shut down after it is up.
2. Once an FE/Port data instance is created, it is never deleted until the entire PDK is shut down. FE/Port shutdown only means that the offline flag is set.
3. If all ports in an FE are shut down, it does not automatically shut down the FE. Shutting down an FE automatically shuts down all the ports that belong to that FE.

4. Components wait for an FE un-bind or Port shutdown event to take proper actions, such as closing all open nodes.
5. The ConfigApp is resposible for FE/Port data persistence, not C&M. Keeping the FE/Port data persistence is not C&M's responsibility, but rather ConfigApp. C&M only does the pure shutdown operation.

The callback procedure for NPF API FE shutdown is as follows:

1. The FP plug-in first calls back all pending operations on the FE with the appropriate return type and, notifies all components and user programs that the previous pending operations have failed.
2. In the C&M FE shutdown callback, it first sets the status of the FE and all the Ports on that FE to offline.
3. After setting the status of FE, it then triggers the Port down event on each port and triggers the FE Unbind event on internal components.
4. In a Port down or FE un-bind callback, internal components must close all open nodes on that Port or FE.
5. After completing all the Port down and FE Unbind event callbacks for internal components one by one to avoid interleaf issues, After finishing all the Port down and FE Unbind event callbacks for internal components (since they are done one by one to avoid interleave problems), C&M triggers the Port down callbacks for each port one by one and then the FE Unbind event to the user program.
6. Finally, C&M calls back to the FE shutdown APIs to finish the shutdown call.
7. The callback procedure for NPF API Port shutdown is similar but easier, since it does not need to do a two-layer shutdown.

2.5.2.2 Shut Down All FEs

All FE shutdown command shuts down all the FEs at the same time. This is only called at PDK shutdown time. ConfigApp must maintain persistency before calling this API. This simply makes a single FE shutdown API call for each FE. Since single FE shutdown is an asynchronous API, the call returns immediately and at the callback of the single FE shutdown, it must check the current pending shutdown status. If it indicates that it is the last FE, the callback thread must call back the All FE shutdown callback to the PDK manager.

Note All FE shutdown is only one step in the PDK shutdown. The PDK manager should shutdown all the internal components to clean up all the internal data structures after successfully completing All FE shutdown.

2.6 Event Notification Callback

The situation is much simpler in the event notification case. Whenever an event notification comes from the FP plug-in along with FE ID, C&M user context, and event data, C&M checks the event type to locate the correct row in the category callback table, starting from the internal callback tables. Then C&M goes through each `CB_t` in the link list of that event category row, invoking all callback functions inside each `CB_t` along with the FE handle, user context, event type, and event data.

Since event callbacks are registered by category, user programs must de-multiplex inside their callback functions. For example, a port MTU change event triggers the Port event callback, even though the user program may only be interested in port line speed changes.

2.7 Helper Functions

In addition to the standard NPF API calls, C&M also provides some helper functions only for C&M itself. External user programs or internal components must not access these helper functions to keep the integrity of the PDK structures. The helper functions are as follows:

cm_isCBIdPresent

Signature: `int cm_isCBIdPresent ( NPF_CBHANDLE )`

Description: Search the callback table to see if a callback handle exists.

Parameters In: Callback handle.

Parameters Out: None.

Return Values: 0 if it exists, -1 otherwise.

cm_getDataByHandle

Signature: `int cm_getDataByHandle ( NPF_HANDLE, NPF_DATA * )`

Description: Search the object lists to get the internal data pointer by NPF object handle.

Parameters In: NPF object handle.

Parameters Out: Pointer to the internal data mapped to that handle.

Return Values: 0 if successful, otherwise –1.

cm_getFEByFEId

Signature: `int cm_getFEByFEId ( uint32_t, FE ** )`

Description: Search the FE list to get the internal FE data pointed to by FE Id.

Parameters In: FE Id, which is assigned by the FP plug-in.

Parameters Out: Pointer to the internal FE data mapped to that FE Id.

Return Values: 0 if successful, otherwise –1.

Part 3: Runtime Interactions

3 Runtime Interactions

Figure 3: Sequence diagram for synchronous APIs

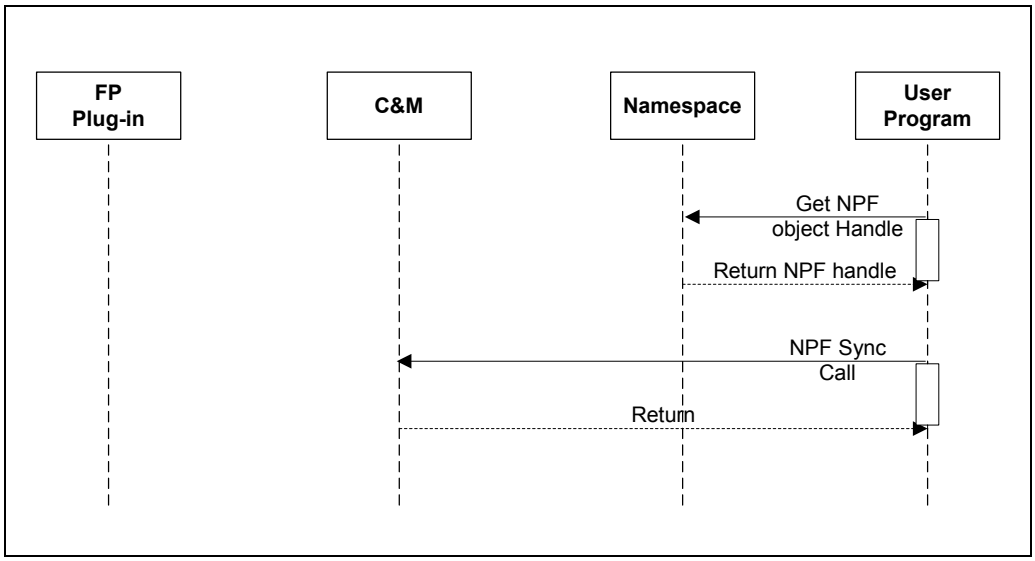

Figure 3: Sequence diagram for synchronous APIs

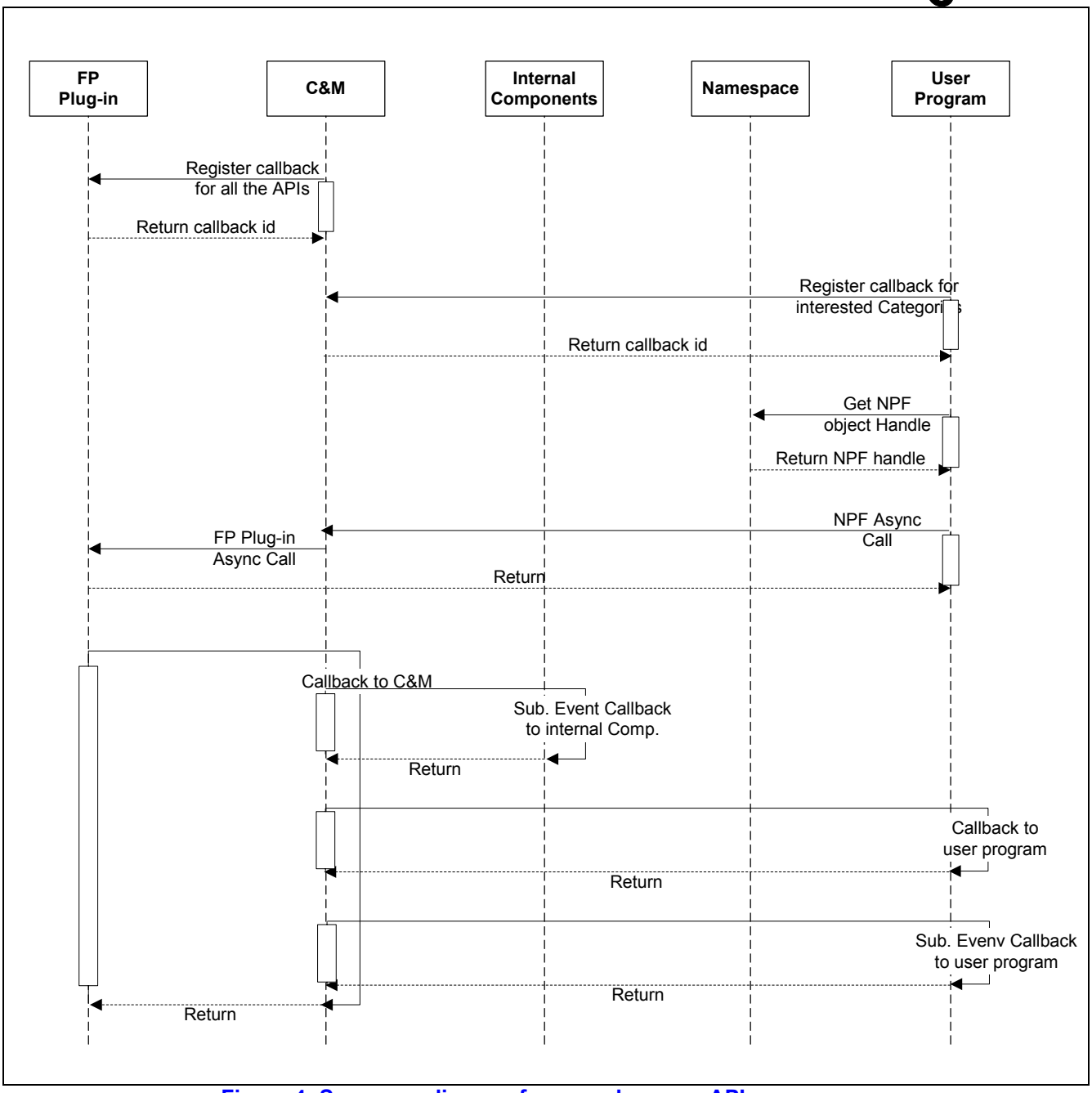

Figure 4: Sequence diagram for asynchronous APIs

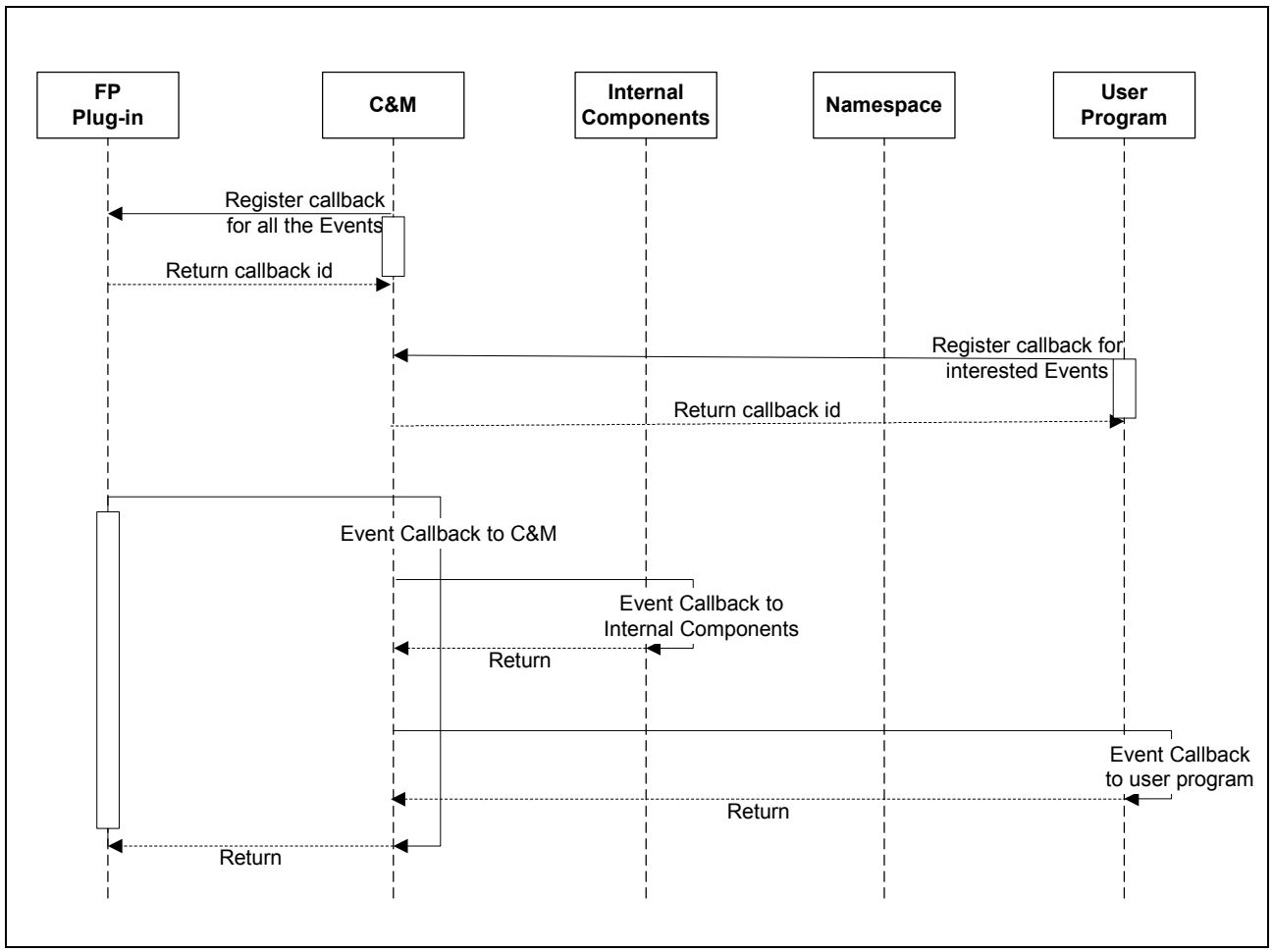

Figure 5: Sequence diagram for event notification

Part 4: Memory Allocation and Locking

4 Memory Allocation and Locking

4.1 Memory Allocation

The PDK C&M implementation always makes a temporary copy of the user program request data, and in some APIs, such as, SET APIs, the same data must be passed down to the FP plug-in. After the callback, C&M copies this data into the system data and if the result is successful, it then deletes the temporary data.

In another sets of APIs, such as, GET APIs, the response data is created by the FP plug-in and called back to the user program. After the callback returns, the FP plug-in deletes the response data immediately. The user program should make a copy of the data if needed.

4.2 Locks

It is important for C&M to protect the global data, such as event and category callback tables and object lists. C&M has a global component lock and also a default `DList` lock for each object list. When C&M traverses the object list or callback tables, or accesses one of the elements in the list, the default list lock provides mutual exclusive protections implicitly, which is the functionality of the `DList`. While a dependency exists between multiple lists, C&M uses the component lock to ensure multiple thread safety between multiple lists. The locks used in C&M and lists are binary semaphore.

Part 5: Pseudo Code

5 Pseudo Code

This section describes pseudo code for some key functions in the C&M API. For simplicity, the pseudo code does not use a lock to protect the global data and does not do some function error checking, such as, malloc.

5.1 Initialization

```
NPF_RET cm_init(void) {
    // init all the object list
    npf_list_init(&SYSTEMList, PIL_FreeMemory);
    npf_list_init(&FEList, PIL_FreeMemory);
    npf_list_init(&PortList, PIL_FreeMemory);
    npf_list_init(&InterfaceList, PIL_FreeMemory);
    // init the registered fp plugin callback list
    npf_list_init(&CMRegFppCBIdList, PIL_FreeMemory);
    // register binding & discovery
    bd_register_();
    // register all the apis with fp plugin
    for each apis fppapi_register_cb();
    for each registered npf_list_pushBack(&CMRegFppCBIdList,
regfppcbid);
    return NPF_SUCCESS;
}
```

5.2 Shutdown

```
NPF_RET cm_destroy(void){
    for each registered cb fppapi_deregister_cb();
    for each deregistered cb
npf_list_popFrontFree(&CMRegFppCBIdList);
    // destroy the registered fp plugin callback list
    npf_list_destroy(&CMRegFppCBIdList);
    // destroy the object list
    npf_list_destroy(&FEList);
    npf_list_destroy(&SYSTEMList);
    npf_list_destroy(&PortList);
    npf_list_destroy(&InterfaceList);
    return NPF_SUCCESS;
}
```

5.3 Registration & Deregistration

```
NPF_RET npf_category_register(  NPF_CBCAT cat,
                                NPF_USERCONTEXT context,
                                NPF_GENERAL_CBFUNC func,
                                NPF_CBHANDLE    *cbid   ) {
    newCB = (CatCB *)malloc(sizeof(CatCB));
    /* assign the unique callback handle */
    *cbid = CallBackCounter++;
    newCB->cbid = *cbid;
```

```
    newCB->context = context;
    newCB->category = cat;
    newCB->func = func;
    /* push to external callback or internal callback table */
    npf_list_pushBack(&CBList[cat], (void *)newCB);
    return NPF_SUCCESS;
}
NPF_RET npf_category_deregister(NPF_CBHANDLE cbid) {
    /* search the category callback table */
    cm_searchCatCBbyCBId(cbid, matchItr);
    npf_list_itrRemoveFree(matchItr);
    return NPF_SUCCESS;
}
```

5.4 Asynchronous API Calls

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

```
NPF_RET npf_ip_set_ipaddr(      NPF_HANDLE      handle,
                                NPF_CBHANDLE    cbid,
                                NPF_CORRELATOR  correlator,
                                NPF_VERBOSITY   verbosity,
                                IPADDR          ipaddr  ) {
    // check if the CB exists, i.e. if registered before
    if (cm_isCBIdExist(cbid) == -1)
        return NPF_INVALID_PARAMETERS;
    // Search Port id from PortList by comparing handle
    // Search fppcbid from CMRegFppCBIdList by comparing client
cookie
    if ((itr = npf_list_itrCreate(&CMRegFppCBIdList)) == NULL)
        return NPF_INVALID_PARAMETERS;
    found = 0;
    npf_list_itrFirst(itr);
    do {
        regfppcbid = (CMREGFPPCBID *)npf_list_itrGetData(itr);
        if ((CMCLIENTCONTEXT)regfppcbid->fppapi ==
CM_CC_IP_SETIPADDR) {
            found = 1;
            break;
        }
    } while (npf_list_itrNext(itr) != -1);
    npf_list_itrDelete(itr);
    if (!found)
        return NPF_INVALID_PARAMETERS;
    // Construct a CM correlator. Note: Should be delete at
onCallBack
    cmcorrelator = (CMCORRELATOR *)malloc(sizeof(CMCORRELATOR));
    cmcorrelator->handle = handle;
    cmcorrelator->cbid = cbid;
    cmcorrelator->correlator = correlator;
    cmcorrelator->verbosity = verbosity;
    newipaddr = (IPADDR *)malloc(sizeof(IPADDR));
    memcpy(newipaddr, &ipaddr, sizeof(IPADDR));
    cmcorrelator->data = newipaddr;
    // call fp plugin
    return fpp_ip_setipaddr(feid, regfppcbid->fppcbid,
cmcorrelator, newipaddr)
}
```

5.5 API Callbacks

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

```
void onCB_fe_set_ipaddr(uint32_t        feid,
                        CMCONTEXT       cmcontext,
                        CMCORRELATOR    cmcorrelator,
                        NPF_DATA        data    ) {
    // check if the return context matches
    if ((CMCLIENTCONTEXT)cmcontext != CM_CC_FE_SETIPADDR)
        return;
    /* assign system data */
    if (cm_getDataByHandle(cmcorrelator->handle, &tmpdata) == -1)
        return;
    fe = (FE *)tmpdata;
    /* make a copy here and will delete data later */
    memcpy(&fe->ipaddr, cmcorrelator->data, sizeof(IPADDR));
    /* trigger the sub. internal event callbacks */
    for each related sub. event
        onCB_xxx_internal_event();
    /* search callback func */
    clientCBFunc =
         (NPF_FE_CBFUNC)pdk_callback_func(cmcontext->cbid,
&context);
    // callback to user program
    clientCBFunc(       context,
                        cmcorrelator->correlator,
                        cmcorrelator->verbosity,
                        cmcorrelator->handle,
                        NPF_FE_SET_IPADDR,
                        data    );
    /* trigger the sub. external event callback */
    for each related sub. Event
        onCB_xxx_external_event();
    // free the temporary data and c&m correlator
    PIL_FreeMemory(cmcorrelator->data);
    PIL_FreeMemory(cmcorrelator);
    return;
}
```