Topology Manager Functional API Implementation Agreement

22 December, 2004

Revision 1.0

Editor(s):

Hormuzd Khosravi, Intel Corporation, hormuzd.m.khosravi@intel.com

The words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", “MAY", and "OPTIONAL" in the remainder of this document are to be interpreted as described in the NPF Software API Conventions Implementation Agreement revision 1.0.

For additional information contact:
The Network Processing Forum, 39355 California Street,
Suite 307, Fremont, CA 94538
+1 510 608-5990 phone ✦ info@npforum.org

Table of Contents

1 Revision History 3
2 Introduction 3
  2.1 Assumptions and External Requirements 3
  2.2 Scope 3
  2.3 Dependencies 3
3 Data Types 4
  3.1 FAPI Topology Types 4
  3.2 Data Structures for Completion Callbacks 8
  3.3 Data Structures for Event Notifications 9
4 Functions 11
  4.1 Completion Callback 11
  4.2 Event Notification Function Calls 12
  4.3 Callback Registration/Deregistration Function Calls 13
  4.4 Event Registration/Deregistration Function Calls 15
  4.5 Topology Discovery APIs 17
  4.6 Examples of Topology Discovery APIs used with LFB APIs 19
5 References 21
6 API Call and Event Capabilities 22
  6.1 Common Function Calls 22
  6.2 Table of Events 22
Appendix A Header file information 23
Appendix B Acknowledgements 27
Appendix C List of companies belonging to NPF DURING APPROVAL PROCESS 28

Table of Figures

Figure 1: A sample LFB graph consisting of Ingress, IPSec Tunnel, IPv4 Forwarder and Egress LFBs 19
Figure 2: A sample LFB graph exposing DiffServ functionality. 20

1 Revision History

1.0	10/27/2004	Created Rev 1.0 of the implementation agreement by taking the Topology Manager FAPI document (npf2002.438.00) and making minor editorial corrections.

2 Introduction

The Functional API (FAPI) is used by vendors to expose board-level functionality. FAPI exposes the functionality specific to each board, as such, it is expected that the set of functions exposed will vary just as the network processing elements on different boards vary. While variance in the set of exposed functions is expected, for each type of function the methods used and the semantics of the function is expected to be vendor agnostic and consistent. Thus, while one board might expose functionality for IPv4 forwarding and NAT, perhaps based on a programmable network processor, and another board might expose functionality for IPv4 forwarding and MPLS using a classification chip and QoS chip, the IPv4 functionality exposed would be the same, at least as to the syntax used. Differences might exist in capabilities (supported numbers of forwarding entries, maximum rate of forwarding, etc.) but not in syntax. The same method would be used to add a forwarding entry, using the same data structures. In order to expose the variation in functions provided by different boards the FAPI model defines two sets of APIs. These are the FAPI Topology Discovery APIs and the FAPI Logical Function Block APIs.

The FAPI Topology Discovery APIs are used to learn the presence of types of functions on a device and acquire handles used to configure instances of those functions. The learning aspects of the FAPI Topology Discovery APIs are expected to be used in scenarios where a blade is “hot plugged” into a system and the control plane must learn the type of blade it is. The handle retrieval methods of the FAPI Topology Discovery APIs are used both in “hot plug” scenarios and in static configuration scenarios, and allow a client program to programmatically acquire handles for use in configuring the tables which control forwarding device behavior.

The FAPI Logical Function Block APIs are used to configure LFB resources and associate resources between LFBs. For example, they can be used to configure Data Path functions such as IPv4 forwarding, MPLS forwarding, tunneling in conjunction with IPv4, etc. The FAPI LFB APIs are specific to different LFBs and will be discussed in separate contributions.

2.1 Assumptions and External Requirements

This API is aligned with the requirements set by the ForCES WG [FORCESREQ] in the IETF.

2.2 Scope

This contribution concentrates on the details of the FAPI Topology Discovery APIs. LFB specific capability APIs are not covered in this contribution. This API does not provide a means to dynamically change the LFB topology.

2.3 Dependencies

This API depends on the SWAPI Software Conventions [SWAPICON] contribution.

3 Data Types

3.1 *FAPI Topology Types*

3.1.1 Handle Types

```
typedef NPF_Uint32_t    NPF_BlockId_t;
```

This is a 32-bit value that is used to identify an LFB or block, is unique per FE and it is obtained using the Topology APIs. This value is transparent to the application or client of the API, however it is a small number, which starts with 1 (0 is reserved) and can be used as an index into an array. It can only be assigned by the Topology FAPI implementation.

```
typedef NPF_Uint32_t    NPF_FE_Handle_t;
```

This is a 32-bit value that is used to identify an FE, is unique per System and it is obtained using the NPF_F_topologyGetFEInfoList() function.

3.1.2 FE Identifier

```
typedef struct NPF_FE_ProductDesignator_s {
    NPF_Uint32_t  enterpriseNo;
    NPF_Char_t    vendorSpecificID[16];
} NPF_FE_ProductDesignator_t;
```

This is a unique FE Identifier structure, which should be unique across FE reboot cycles. The enterpriseNo field is the unique SNMP enterprise number as assigned by IANA (http://www.iana.org/assignments/enterprise-numbers). The vendorSpecificID can contain any vendor specific information. Note that the application cannot alter the FE Product Designator and it is assigned by the Topology FAPI implementation.

3.1.3 FE Information

```
typedef struct NPF_FEInfo_s {
      NPF_FE_Handle_t   feHandle;
      NPF_FE_ProductDesignator_t    feDesignator;
      NPF_Uint32_t   locationID;
} NPF_FEInfo_t;
```

This structure describes the FE. The feHandle field is the FE Handle used to identify an FE, the feDesignator is a unique FE identifier. This identifier is constant across FE reboot cycles. The locationID is used to denote the location or slot of the FE in a chassis.

```
typedef struct NPF_FEInfoList_s {
   NPF_Uint32_t   feCount;
   NPF_FEInfo_t   *feArray;
} NPF_FEInfoList_t;
```

This structure defines an array of NPF_FEInfo_t structures. The feCount field defines the number of entries in the array and feArray is a pointer to the start of the array of NPF_FEInfo_t structures.

3.1.4 Block Type

```
typedef struct NPF_BlockType_s {
      NPF_Uint32_t      blockType;
      NPF_Char_t        *blockDescriptor;
} NPF_BlockType_t;
```

This structure describes the Block type. The blockType field is a pre-defined LFB type, the blockDescriptor is a text description of the LFB. The blockDescriptor is a Null terminated string with maximum length of 256 bytes.

The 32-bit blockType is divided into four types of ranges, NPF-standardized, Proprietary, Experimental and Reserved, as follows:

```
0x00000000-0x07ffffff: Used for block types standardized by NPF.
0x08000000-0x3fffffff: Reserved for future use.
0x40000000-0x47ffffff: Used for exported but proprietary block types. It is
anticipated that a vendor's solution offered to an integrator may need to
include vendor-specific block types in addition to the block types already
standardized by NPF.  This range is reserved for such vendor specific block
types. Allocation of this range must be coordinated among vendors in order to
avoid collision in multi-vendor integrations.
0x48000000-0x77ffffff: Reserved for future use.
0x78000000-0x7fffffff: For experimental use. This is an uncoordinated range
meant for proprietary (intra-company) experimentation.

The corresponding header file definitions are as follows:

#define NPF_LFB_TYPE_STD_MIN 0x00000000
#define NPF_LFB_TYPE_STD_MAX 0x07ffffff
#define NPF_LFB_TYPE_PROP_MIN 0x40000000
#define NPF_LFB_TYPE_PROP_MAX 0x47ffffff
#define NPF_LFB_TYPE_EXP_MIN 0x78000000
#define NPF_LFB_TYPE_EXP_MAX 0x7fffffff

Here are block types for LFBS which have currently been defined:

#define NPF_LFB_TYPE_IPv4_PREFIX 10
#define NPF_LFB_TYPE_IPv4_NEXTHOP 11
#define NPF_LFB_TYPE_DIFFSERV_METER 12
#define NPF_LFB_TYPE_GENERIC_CLASSIFIER 13
#define NPF_LFB_TYPE_MESSAGING 14
```

Note: The definition of the complete set of Block types is out of scope of this document. These should be defined in the corresponding FAPI documents.

3.1.5 LFB Edge Attributes

```
typedef struct NPF_ LFB_Edge_Attribute_s {
   NPF_BlockId_t    lfbId;
   NPF_Uint32_t      lfbInputPortId;
   NPF_Uint32_t      lfbOuputPortId;
} NPF_LFB_Edge_Attribute_t;
```

This structure defines the LFB edge attributes. It consists of the lfbId which identifies the LFB, the lfbInputPortId which identifies the input port on the LFB from which the connection to this LFB originates and the lfbOutputPortId which identifies the output port on this LFB. Since the upstream LFB structure always points to the LFB Edge structure, this structure does not need to identify the upstream LFB.

3.1.6 LFB Instance (Node), LFB Instance List

```
typedef struct NPF_LFBInstance_s {
   NPF_Uint32_t                lfbId;
   NPF_BlockType_t             lfbType;
   NPF_Uint16_t                toLFBCount;
   NPF_ LFB_Edge_Attribute_t  *toLFBArray;
} NPF_LFBInstance_t;
```

This structure describes the LFB node. The lfbId, lfbType identify the LFB or block (Id and Type) from which the connection originates and toLFBArray defines the array of LFB edges (connections to other LFBs) which originate from this LFB.

```
typedef struct NPF_LFBInstanceList_s {
   NPF_FE_Handle_t   feHandle;
   NPF_Uint32_t      nodeCount;
   NPF_LFBInstance_t *nodeArray;
} NPF_LFBInstanceList_t;
```

This structure defines an array of NPF_LFBInstance_t structures. The feHandle refers to the corresponding FE, nodeCount defines the number of entries in the array and nodeArray is a pointer to the start of the array of NPF_LFBInstance_t structures. The order of nodes in the array is not specified i.e. it might be in order of increasing block ID or tree-walk order or some other order.

3.1.7 Error Codes

```
typedef NPF_Uint32_t NPF_F_topologyErrorType_t;
```

This defines the asynchronous error codes returned in the function callbacks.

```
#define NPF_FTOPOLOGY_BASE_ERR 1000 /* Base value of 1000 wrt other NPF codes
*/

/* Invalid FE handle */
#define NPF_FTOPOLOGY_E_INVALID_FE_HANDLE \
((NPF_F_topologyErrorType_t) NPF_FTOPOLOGY_BASE_ERR + 1)
```

3.2 *Data Structures for Completion Callbacks*

A completion callback is defined for each of the functions in this API.

3.2.1 Callback Type

The callback response contains one of the following codes, indicating the function that was called to cause the callback. This code tells the application how to interpret the data included in the union that is part of the response structure.

```
/* completion callback types */
typedef enum NPF_F_topologyCallbackType {
      NPF_F_TOPOLOGY_GET_FE_INFOLIST     = 1,
      NPF_F_TOPOLOGY_GET_GRAPH_NODELIST  = 2
} NPF_F_topologyCallbackType_t;
```

3.2.2 Callback Data

This is the callback response structure which is passed to the caller in the asynchronous response from a function call. It contains an error/success code, the callback type that identifies the function called and a function-specific structure embedded in a union.

```
typedef struct {
      NPF_F_topologyCallbackType_t type;
      NPF_F_topologyErrorType_t error;
      union {
            NPF_FEInfoList_t        feInfoArray;
            NPF_LFBInstanceList_t         lfbNodeArray;
      } u;
} NPF_F_topologyCallbackData_t;
```

Table 2-1. Callback type to Callback data mapping table

Callback Type	Callback Data
NPF_F_TOPOLOGY_GET_FE_INFOLIST	NPF_FEInfoList_t
NPF_F_TOPOLOGY_GET_GRAPH_NODELIST	NPF_LFBInstanceList_t

3.3 Data Structures for Event Notifications

The following sections detail the information related to Topology events. When an event notification routine is invoked, one of the parameters will be a structure of information related to one or more events.

3.3.1 Event Notification Types

The event type indicates the type of event data in the union of event structures returned in NPF_F_topologyEventData_t.

```
/*
 * This structure enumerates the events defined for
 * Topology Manager API.
 */
typedef enum NPF_F_topologyEvent {
   NPF_F_TOPOLOGY_NEW_FE_APPEAR   =  1,
   NPF_F_TOPOLOGY_FE_DISAPPEAR    =  2,
   NPF_F_TOPOLOGY_LFB_TOPO_CHANGE = 3
} NPF_F_topologyEvent_t;

/*
 * Definitions for Topology events to be
 * used in event Mask.
 */
#define NPF_F_TOPOLOGY_EV_NEW_FE_APPEAR   (1 << 1)
#define NPF_F_TOPOLOGY_EV_FE_DISAPPEAR    (1 << 2)
#define NPF_F_TOPOLOGY_LFB_TOPO_CHANGE    (1 << 3)
```

3.3.2 Event Notification Structures

This section describes the various events which MAY be implemented.

It is important to note that even if an implementation does not support any of these events, the implementation still needs to provide the register and deregister event function to enable interoperability.

This structure defines all the possible event definitions for Topology FAPI. An event type field indicates which member of the union is relevant in the specific structure.

```
/*
 * This structure represents a single event in the event array. The
 * type field indicates the specific event in the union.
 */

typedef struct {
   NPF_F_topologyEvent_t  type;
   union {
      NPF_FEInfo_t      feInfo;
   } u;
} NPF_F_topologyEventData_t;
```

This structure represents the data parameter provided when the event notification routine is invoked. It contains a count of events and an array of structures providing event specific information.

```
/*
 * This structure is provided when the event notification handler
 * is invoked. It specifies one or more Topology FAPI events.
 */
typedef struct {
   NPF_uint32_t                       numEvents;
   NPF_F_topologyEventData_t         *eventArray;
} NPF_F_topologyEventArray_t;
```

4 Functions

4.1 Completion Callback

This callback function is for the application to register an asynchronous response handling routine to the Topology FAPI implementation.

4.1.1 Completion Callback Function Signature

```
typedef void (*NPF_F_topologyCallBackFunc_t)(
      NPF_IN NPF_userContext_t             userContext,
      NPF_IN NPF_correlator_t              correlator,
      NPF_IN NPF_F_topologyCallbackData_t ftopologyCallbackData);
```

4.1.1.1 Description

This callback function is for the application to register asynchronous response handling routine to the NPF FAPI Topology API implementation. This callback function is intended to be implemented by the application, and be registered to the NPF FAPI Topology API implementation through NPF_F_topologyRegister( ) function.

4.1.1.2 Input Parameters

- userContext
  The context item that was supplied by the application when the completion callback function was registered.
- correlator
  The correlator item that was supplied by the application when the FAPI Topology API function call was made. The correlator is used by the application mainly to distinguish between multiple invocations of the same function.
- ftopologyCallbackData
  Response information related to the FAPI Topology API function call. Contains information that are common among all functions, as well as information that are specific to a particular function. See NPF_ftopologyCallbackData_t definition for details.

4.1.1.3 Output Parameters

None.

4.1.1.4 Return Value

None.

4.2 Event Notification Function Calls

This event notification function is for the application to register an event handler routine to the Topology FAPI implementation.

4.2.1 NPF_F_topologyEventCallFunc_t

```
typedef void (*NPF_F_topologyEventCallFunc_t) (
   NPF_IN NPF_userContext_t       userContext,
   NPF_IN NPF_F_topologyEventArray_t data);
```

4.2.1.1 Description

This function is a registered event notification routine for handling Topology FAPI events.

4.2.1.2 Input Parameters

- userContext - The context item that was supplied by the application when the event callback routine was registered.
- data – A structure containing an array of event data structures and a count to indicate how many events are present. Each of these NPF_F_topologyEventData_t members contains event specific information and a type field to identify the particular event.

4.2.1.3 Output Parameters

None

4.2.1.4 Return Value

None

4.3 Callback Registration/Deregistration Function Calls

This section defines the registration and de-registration functions used to install and remove an asynchronous response callback routine.

4.3.1 Completion Callback Registration Function

```
NPF_error_t NPF_F_topologyRegister(
      NPF_IN NPF_userContext_t                  userContext,
      NPF_IN NPF_F_topologyCallbackFunc_t ftopologyCallbackFunc,
      NPF_OUT NPF_F_topologyCallbackHandle_t
      *ftopologyCallbackHandle);
```

4.3.1.1 Description

This function is used by an application to register its completion callback function for receiving asynchronous responses related to NPF FAPI Topology API function calls. Application may register multiple callback functions using this function. The callback function is identified by the pair of userContext and ftopologyCallbackFunc, and for each individual pair, a unique ftopologyCallbackHandle will be assigned for future reference. Since the callback function is identified by both userContext and ftopologyCallbackFunc, duplicate registration of same callback function with different userContext is allowed. Also, same userContext can be shared among different callback functions. Duplicate registration of the same userContext and ftopologyCallbackFunc pair has no effect, and will output a handle that is already assigned to the pair, and will return NPF_E_ALREADY_REGISTERED.

Note : NPF_F_topologyRegister( ) is a synchronous function and has no completion callback associated with it.

4.3.1.2 Input Parameters

- userContext
  A context item for uniquely identifying the context of the application registering the completion callback function. The exact value will be provided back to the registered completion callback function as its 1st parameter when it is called. Application can assign any value to the userContext and the value is completely opaque to the NPF FAPI Topology API implementation.
- ftopologyCallbackFunc
  The pointer to the completion callback function to be registered.

4.3.1.3 Out Parameters

- ftopologyCallbackHandle
  A unique identifier assigned for the registered userContext and ftopologyCallbackFunc pair. This handle will be used by the application to specify which callback function to be called when invoking asynchronous NPF FAPI Topology API functions. It will also be used when de-registering the userContext and ftopologyCallbackFunc pair.

4.3.1.4 Return Values

- NPF_NO_ERROR
  The registration completed successfully.
- NPF_E_BAD_CALLBACK_FUNCTION
  ftopologyCallbackFunc is NULL.
- NPF_E_ALREADY_REGISTERED
  No new registration was made since the userContext and ftopologyCallbackFunc pair was already registered.
  Note: Whether this should be treated as an error or not is dependent on the application.

4.3.2 Completion Callback Deregistration

```
NPF_error_t NPF_F_topologyDeregister(
      NPF_IN NPF_F_topologyCallbackHandle_t
      ftopologyCallbackHandle);
```

4.3.2.1 Description

This function is used by an application to de-register a pair of user context and callback function.
Note: If there are any outstanding calls related to the de-registered callback function, the callback function may be called for those outstanding calls even after de-registration.
Note: NPF_F_topologyEventRegister( ) is a synchronous function and has no completion callback associated with it.

4.3.2.2 Input Parameters

- ftopologyCallbackHandle
  The unique identifier representing the pair of user context and callback function to be de-registered.

4.3.2.3 Output Parameters

None.

4.3.2.4 Return Values

- NPF_NO_ERROR
  The de-registration completed successfully.
- NPF_E_BAD_CALLBACK_HANDLE

The API implementation does not recognize the callback handle. There is no effect to the registered callback functions.

4.4 Event Registration/Deregistration Function Calls

This section defines the registration and de-registration functions used to install and remove an event handler routine

4.4.1 NPF_F_topologyEventRegister

```
NPF_error_t NPF_F_topologyEventRegister(
   NPF_IN NPF_userContext_t           userContext,
   NPF_IN NPF_F_topologyEventCallFunc_t eventCallFunc,
   NPF_IN NPF_eventMask_t            eventMask,
   NPF_OUT NPF_callbackHandle_t     *eventCallHandle);
```

4.4.1.1 Description

This function is used by an application to register its event handling routine for receiving notifications of Topology events. Applications MAY register multiple event handling routines using this function. The event handling routine is identified by the pair of userContext and eventCallFunc, and for each individual pair, a unique eventCallHandle will be assigned for future reference.

Since the event handling routine is identified by both userContext and eventCallFunc, duplicate registration of the same event handling routine with a different userContext is allowed. Also, the same userContext can be shared among different event handling routines. Duplicate registration of the same userContext and eventCallFunc pair has no effect, and will output a handle that is already assigned to the pair, and will return NPF_E_ALREADY_REGISTERED.

4.4.1.2 Input Parameters

- userContext – A context item for uniquely identifying the context of the application registering the event handling routine. The exact value will be provided back to the registered event handling routine as its first parameter when it is called. Applications can assign any value to the userContext and the value is completely opaque to the Topology FAPI implementation
- eventCallFunc – The pointer to the event handling routine to be registered.
- eventMask – This is a bit mask of the Topology events. It allows the application to register for those selected events.

4.4.1.3 Output Parameters

- eventCallHandle - A unique identifier assigned for the registered userContext and eventCallFunc pair. This handle will be used when deregistering the userContext and eventCallFunc pair.

4.4.1.4 Return Values

- NPF_NO_ERROR - The registration completed successfully.
- NPF_E_BAD_CALLBACK_FUNCTION – The eventCallFunc is NULL, or otherwise invalid.

- NPF_E_CALLBACK_ALREADY_REGISTERED – No new registration was made since the userContext and eventCallFunc pair was already registered.

4.4.2 NPF_F_topologyEventDeregister

```
NPF_error_t NPF_F_topologyEventDeregister(
   NPF_IN NPF_callbackHandle_t eventCallHandle);
```

4.4.2.1 Description

This function is used by an application to de-register an event handler routine which was previously registered to receive notifications of Topology events. It represents a unique user context and event handling routine pair.

4.4.2.2 Input Parameters

- eventCallHandle - The unique identifier returned to the application when the event callback routine was registered.

4.4.2.3 Output Parameters

None

4.4.2.4 Return Values

- NPF_NO_ERROR - The de-registration completed successfully.
- NPF_E_BAD_CALLBACK_HANDLE – The de-registration did not complete successfully due to problems with the callback handle provided.

4.5 Topology Discovery APIs

The FAPI Topology Discovery APIs are used to determine what logical function blocks are implemented by a device and the sequence in which packets flow through them. These APIs are expected to be used when a blade is "plugged in" to a system or when a blade boots up and an application needs to get FE handles and block ids to use to programmatically configure and control the blade. The graph of block connections that a device supports will be queried by applications.

4.5.1 NPF_F_topologyGetFEInfoList()

```
NPF_error_t    NPF_F_topologyGetFEInfoList(
                    NPF_IN      NPF_callbackHandle_t    cbHandle,
                    NPF_IN      NPF_correlator_t        correlator);
```

4.5.1.1 Description

This function is used to retrieve information about the FEs in the system.

4.5.1.2 Input Parameters

- cbHandle: The callback handle returned by NPF_ftopologyRegister() call.
- correlator: A 32-bit value that will be returned in the callback for this function call.

4.5.1.3 Output Parameters

None.

4.5.1.4 Return Codes

- NPF_NO_ERROR: The function call was accepted, and a callback will occur or has already occured.
- NPF_E_BAD_CALLBACK_HANDLE: The cbHandle parameter is invalid; no callback will occur.

4.5.1.5 Asynchronous Callback Response

- feInfoArray: The NPF_FEInfoList_t struct as part of NPF_ftopologyCallbackData_t is returned to the caller of the API.

4.5.2 NPF_F_topologyGetLFBInstanceList()

```
NPF_error_t    NPF_F_topologyGetLFBInstanceList(
                  NPF_IN      NPF_callbackHandle_t    cbHandle,
                  NPF_IN      NPF_correlator_t        correlator,
                  NPF_IN      NPF_FE_Handle_t         feHandle);
```

4.5.2.1 Description

This function is used to get the array of all LFB nodes in the DG (directed graph) for the particular FE. Note that the LFB topology of a given FE is not necessarily a connected graph, it may be a collection of sub-graphs (each being a connected graph by itself), and it can even contain nodes that are not (yet) connected to any other nodes.

4.5.2.2 Input Parameters

- cbHandle: The callback handle returned by NPF_ftopologyRegister() call.
- correlator: A 32-bit value that will be returned in the callback for this function call.
- feHandle: The FE Handle returned by NPF_ftopologyGetFEInfoList_t() call.

4.5.2.3 Output Parameters

None.

4.5.2.4 Return Codes

- NPF_NO_ERROR: The function call was accepted, and a callback will occur or has already occured.
- NPF_E_BAD_CALLBACK_HANDLE: The cbHandle parameter is invalid; no callback will occur.

4.5.2.5 Asynchronous Callback Response

- `NPF_FTOPOLOGY_E_INVALID_FE_HANDLE`: The feHandle is invalid.
- lfbNodeArray: The NPF_LFBInstanceList_t as part of NPF_ftopologyCallbackData_t is returned to the caller of the API.

4.6 Examples of Topology Discovery APIs used with LFB APIs

This section describes some usage scenarios for the Topology Discovery APIs used in conjunction with LFB APIs to query and configure LFBs and datapath functions. It uses pseudo-code to describe this wherever appropriate. These examples and the LFB APIs used are for illustration purpose only.

A sample example of a LFB graph is shown in Figure 1. It consists of four LFBs: Ingress, IPSec Tunnel, IPv4 Forwarder and Egress, which are connected together and expose the logical functionality of a blade.

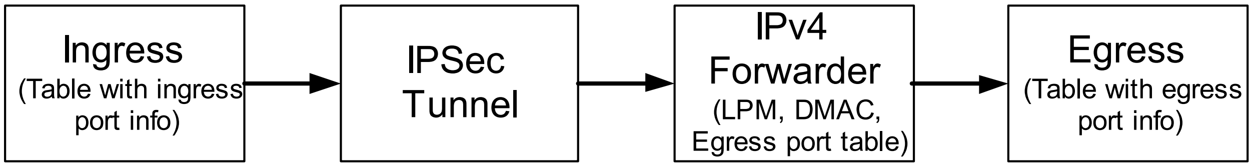
Ingress
(Table with ingress port info)
IPSec Tunnel
IPv4 Forwarder
(LPM, DMAC, Egress port table)
Egress
(Table with egress port info)

Figure 1: A sample LFB graph consisting of Ingress, IPSec Tunnel, IPv4 Forwarder and Egress LFBs

The application uses NPF_F_topologyGetFEInfoList () to find the FEs in the system and NPF_F_topologyGetLFBInstanceList () to find all the LFB connectivity information in the system.

In order to configure any LFB in the above graph, the application would use the specific LFB API. For example, it would configure an IPSec Tunnel using the tunnel specific API below.

```
NPF_F_tunnel_AddEntry (feHandle, blockID, tunnelData);
```

where tunnelData is a specific struct which would have fields such as the Key, security Algorithm, etc.

The tunnelData might also have information needed to associate a Tunnel entry with an entry in the IPv4 Forwarder. Also, note that the blockID used in the call is obtained from the Topology APIs.

Another example of a LFB graph exposing DiffServ functionality is shown in Figure 2. It consists of six LFBs: Ingress, 6-tuple DiffServ Classifier, Meter, Marker, Scheduler and Egress.

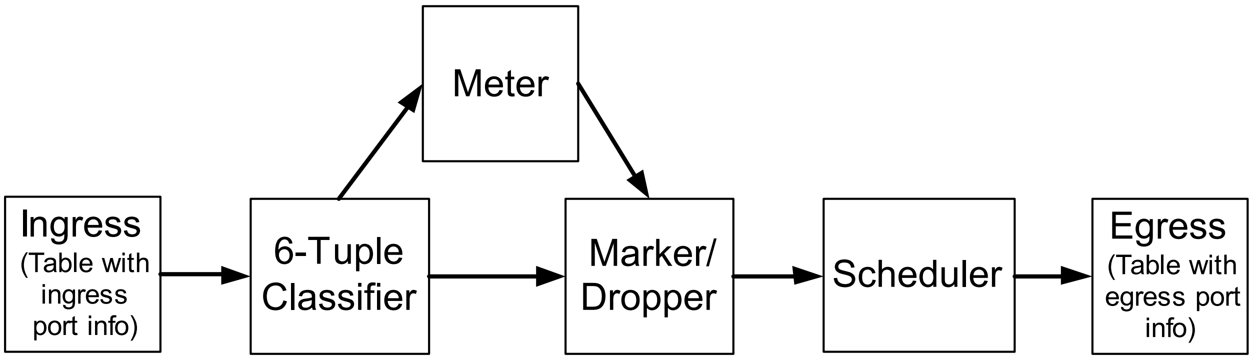

Figure 2: A sample LFB graph exposing DiffServ functionality.

The application would query the LFB information in the same manner as described before.

In order to configure a particular DiffServ policy in the blade, the application would need to configure individual LFBs. For example, to set up some marking action for a 5-tuple filter, the application would make the following API calls.

```
NPF_F_5tupleClfr_AddEntry (feHandle, blockID, clfrData);

NPF_F_meter_AddEntry (feHandle, blockID, meterData);
```

where the clfrData and meterData structs are specific to those LFBs and might have a common field such as FlowId to associate each other’s entries.

5 References

[FORCESREQ] ”Requirements for Separation of IP Control and Forwarding”, H. Khosravi, T. Anderson et al, July 2003. (http://www.ietf.org/rfc/rfc3654.txt)

[DIFFSERV] ”An Informal Management Model for Diffserv Routers”, Y. Bernet et al, May 2002. (http://www.ietf.org/rfc/rfc3290.txt)

[SWAPICON] “SwAPI Software Conventions Implementation Agreement”, Rev 2.0, Network Processing Forum, August 2002

6 API Call and Event Capabilities

These tables are included as a summary for informative purposes.

6.1 Common Function Calls

API function Name	**Function Required**
`NPF_F_topologyCallBackFunc`	Required
`NPF_F_topologyEventCallFunc`	Required
`NPF_F_topologyRegister`	Required
`NPF_F_topologyDeregister`	Required
`NPF_F_topologyEventRegister`	Required
`NPF_F_topologyEventDeregister`	Required
`NPF_F_topologyGetFEInfoList`	Required
`NPF_F_topologyGetLFBInstanceList`	Required

6.2 Table of Events

Event Name	**Event Required**
`NPF_F_TOPOLOGY_NEW_FE_APPEAR`	Optional
`NPF_F_TOPOLOGY_FE_DISAPPEAR`	Optional
`NPF_F_TOPOLOGY_LFB_TOPO_CHANGE`	Optional

APPENDIX A HEADER FILE INFORMATION

```
/*
 * This header file defines typedefs, constants, and functions
 * for the NP Forum Functional Topology Manager API
 */
#ifndef __NPF_F_TOPO_H
#define __NPF_F_TOPO_H

#ifdef __cplusplus
extern "C"      {
#endif

/*---------------------------------------------------------------
 *
 * Common Data Types
 *
 *---------------------------------------------------------------*/

typedef NPF_Uint32_t    NPF_BlockId_t;

typedef NPF_Uint32_t    NPF_FE_Handle_t;

typedef struct NPF_FE_ProductDesignator_s {
    NPF_Uint32_t  enterpriseNo;
    NPF_Char_t    vendorSpecificID[16];
} NPF_FE_ProductDesignator_t;

typedef struct NPF_FEInfo_s {
      NPF_FE_Handle_t   feHandle;
      NPF_FE_ProductDesignator_t    feDesignator;
      NPF_Uint32_t      locationID;
} NPF_FEInfo_t;

typedef struct NPF_FEInfoList_s {
   NPF_Uint32_t   feCount;
   NPF_FEInfo_t   *feArray;
} NPF_FEInfoList_t;

typedef struct NPF_BlockType_s {
      NPF_Uint32_t      blockType;
      NPF_Char_t        *blockDescriptor;
} NPF_BlockType_t;

typedef struct NPF_ LFB_Edge_Attribute_s {
   NPF_BlockId_t    lFBId;
   NPF_Uint32_t      lfbInputPortId;
   NPF_Uint32_t      lfbOuputPortId;
} NPF_LFB_Edge_Attribute_t;

typedef struct NPF_LFBInstance_s {
   NPF_Uint32_t               lfbId;
   NPF_BlockType_t            lFBType;
   NPF_Uint16_t               toLFBCount;
   NPF_ LFB_Edge_Attribute_t  *toLFBArray;
} NPF_LFBInstance_t;


typedef struct NPF_LFBInstanceList_s {
   NPF_FE_Handle_t   feHandle;
   NPF_Uint32_t      nodeCount;
   NPF_LFBInstance_t *nodeArray;
} NPF_LFBInstanceList_t;


typedef NPF_Uint32_t NPF_F_topologyErrorType_t;

#define NPF_FTOPOLOGY_BASE_ERR 1000 /* Base value of 1000 wrt other NPF codes
*/

/* Invalid FE handle */
#define NPF_FTOPOLOGY_E_INVALID_FE_HANDLE \
      ((NPF_F_topologyErrorType_t) NPF_FTOPOLOGY_BASE_ERR + 1)

/*-----------------------------------------------------------------
 *
 * Completion Callback Data Types
 *
 *-----------------------------------------------------------------*/

/* completion callback types */
typedef enum NPF_F_topologyCallbackType {
      NPF_F_TOPOLOGY_GET_FE_INFOLIST     = 1,
      NPF_F_TOPOLOGY_GET_GRAPH_NODELIST  = 2
} NPF_F_topologyCallbackType_t;


typedef struct {
      NPF_F_topologyCallbackType_t type;
      NPF_F_topologyErrorType_t error;
      union {
            NPF_FEInfoList_t        feInfoArray;
            NPF_LFBInstanceList_t         lfbNodeArray;
      } u;
} NPF_F_topologyCallbackData_t;


/*-----------------------------------------------------------------
 *
 * Event Notification Data Types
 *
 *-----------------------------------------------------------------*/

/*
 * This structure enumerates the events defined for
 * Topology Manager API.
 */
typedef enum NPF_F_topologyEvent {
   NPF_F_TOPOLOGY_NEW_FE_APPEAR   =  1,
   NPF_F_TOPOLOGY_FE_DISAPPEAR =  2,
   NPF_F_TOPOLOGY_LFB_TOPO_CHANGE = 3
} NPF_F_topologyEvent_t;


/*
 * Definitions for Topology events to be
 * used in event Mask.
 */
#define NPF_F_TOPOLOGY_EV_NEW_FE_APPEAR   (1 << 1)
#define NPF_F_TOPOLOGY_EV_FE_DISAPPEAR    (1 << 2)
#define NPF_F_TOPOLOGY_LFB_TOPO_CHANGE    (1 << 3)

/*
 * This structure represents a single event in the event array. The
 * type field indicates the specific event in the union.
*/
typedef struct {
   NPF_F_topologyEvent_t  type;
   union {
         NPF_FEInfo_t   feInfo;
   } u;
} NPF_F_topologyEventData_t;


/*
 * This structure is provided when the event notification handler
 * is invoked. It specifies one or more Topology FAPI events.
 */
typedef struct {
   NPF_uint32_t                       numEvents;
   NPF_F_topologyEventData_t         *eventArray;
} NPF_F_topologyEventArray_t;


/*-----------------------------------------------------------------
 *
 * Function Call Prototypes
 *
 *-----------------------------------------------------------------*/

typedef void (*NPF_F_topologyCallBackFunc_t)(
      NPF_IN NPF_userContext_t            userContext,
      NPF_IN NPF_correlator_t             correlator,
      NPF_IN NPF_F_topologyCallbackData_t ftopologyCallbackData);

typedef void (*NPF_F_topologyEventCallFunc_t) (
         NPF_IN NPF_userContext_t       userContext,
         NPF_IN NPF_F_topologyEventArray_t data);

NPF_error_t NPF_F_topologyRegister(
      NPF_IN NPF_userContext_t                  userContext,
      NPF_IN NPF_F_topologyCallbackFunc_t       ftopologyCallbackFunc,
      NPF_OUT NPF_F_topologyCallbackHandle_t    *ftopologyCallbackHandle);

NPF_error_t NPF_F_topologyDeregister(
      NPF_IN NPF_F_topologyCallbackHandle_t     ftopologyCallbackHandle);
```

```
NPF_error_t NPF_F_topologyEventRegister(
         NPF_IN NPF_userContext_t          userContext,
         NPF_IN NPF_F_topologyEventCallFunc_t eventCallFunc,
         NPF_IN NPF_eventMask_t            eventMask,
         NPF_OUT NPF_callbackHandle_t     *eventCallHandle);

NPF_error_t NPF_F_topologyEventDeregister(
         NPF_IN NPF_callbackHandle_t eventCallHandle);

NPF_error_t    NPF_F_topologyGetFEInfoList(
         NPF_IN   NPF_callbackHandle_t    cbHandle,
         NPF_IN   NPF_correlator_t        correlator);

NPF_error_t    NPF_F_topologyGetLFBInstanceList(
         NPF_IN   NPF_callbackHandle_t    cbHandle,
         NPF_IN   NPF_correlator_t        correlator,
         NPF_IN   NPF_FE_Handle_t         feHandle);


#ifdef __cplusplus
}
#endif

#endif /* __NPF_F_TOPO_H */
```

APPENDIX B ACKNOWLEDGEMENTS

Working Group Chair: Alex Conta

Working Group Editor: John Renwick

Task Group Chair: Alistair Munro

The following individuals are acknowledged for their participation in the FAPI TG teleconferences, plenary meetings, mailing list, and/or for their NPF contributions used for the development of this Implementation Agreement. This list may not be all-inclusive since only names supplied by member companies for inclusion here will be listed. The NPF wishes to thank all active participants to this Implementation Agreement, whether listed here or not.

The list is in alphabetical order of last names:

Steven Blake, Modular Networks, Inc.
Gamil Cain, Intel
Jason Goldschmidt, Sun Microsystems
Reda Haddad, Ericsson
Zsolt Haraszti, Modular Networks, Inc.
Hormuzd Khosravi, Intel
Vinoj Kumar, Agere Systems
David Maxwell, IDT
David Putzolu, Intel
John Renwick, Agere Systems
Michael Speer, Sun Microsystems

APPENDIX C LIST OF COMPANIES BELONGING TO NPF DURING APPROVAL PROCESS

Agere Systems	HCL Technologies	Nortel Networks
Altera	Hifn	NTT Electronics
AMCC	IBM	PMC Sierra
Analog Devices	IDT	Seaway Networks
Avici Systems	Infineon Technologies AG	Sensory Networks
Cypress Semiconductor	Intel	Sun Microsystems
Enigma Semiconductor	IP Fabrics	Teja Technologies
Ericsson	IP Infusion	TranSwitch
Erlang Technologies	Kawasaki LSI	U4EA Group
ETRI	Motorola	Xelerated
EZChip	NetLogic	Xilinx
Flextronics	Nokia	