Transport Plug-in

Design Specification

Control Plane-Platform Development Kit 2.11

March 2004

<Document Part # /Release # /Order #: XXXXX

Contents

Figures

Tables

Revision History

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

Part 1: Introduction

1 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 or 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) services 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/.

This document specifies the high-level design for the transport plug-in module of the CP-PDK.

1.1 Terminology

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

Table 1. Terminology table

Term	Description
NPF	Network Processing Forum
CE	Control Element
FE	Forwarding Element
ForCES	Forwarding and Control Element Separation protocol
PDK	Platform Development Kit

Term	Description
COPS	Common Open Policy Service protocol
GSMP	General Switch Management Protocol
PCI	Peripheral Connect Interface
BER	Basic Encoding Rules
XML	Extensible Markup Language
TLV	Type Length Value

1.2 Reference

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

Table 2. Reference table

Reference	Document
[1]	NPF Application Level API Framework; NP Forum, September 2000.
[2]	CP-PDK : Software Architecture Overview
[3]	CP-PDK : Forwarding Plane Plug-in API Reference
[4]	Requirements for Separation of IP Control and Forwarding, IETF draft
[5]	ForCES FE Functional Model, IETF draft
[6]	ForCES Architectural Framework, IETF draft
[7]	COPS Portability Layer Specification

Part 2: Transport Plug-in Architecture

2 Transport Plug-in Architecture

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

The transport plug-in abstracts out the type and the details of the communication mechanisms from the rest of the PDK implementation, at the same time providing the functionality required for separation of the CP and FP. It enables plug-and-play functionality for different communication mechanisms with the rest of the PDK. Different types of transport plug-ins can be placed between the planes such that CPs and FPs communicate transparently. This section describes the architecture for a transport plug-in.

The architecture of a transport plug-in is shown in the figure that follows. The plug-in is composed of four distinct parts:

1. **FP Plug-in API** - The abstraction API that hides the transport plug-in details and presents a uniform API that gets invoked by the NPF API implementation modules on the control plane.
2. **Backend API** - The API exposed by the transport plug-in on the FP, which is used by the FP module of the PDK.

In addition to the two APIs above, the transport plug-in includes the following components:

3. **Transport Protocol** - This is the standard or propriety protocol used to exchange information between the planes and consists of two agents.
    - **Control plane agent** - Part of the transport protocol that resides on the control plane and communicates with the FP agent
    - **Forwarding plane agen**t - Part of the transport protocol that resides on the FP and communicates with the CP agent
4. **Interconnect abstraction layer** - This abstraction layer hides the interconnect details and is used by the transport protocol to send and receive messages

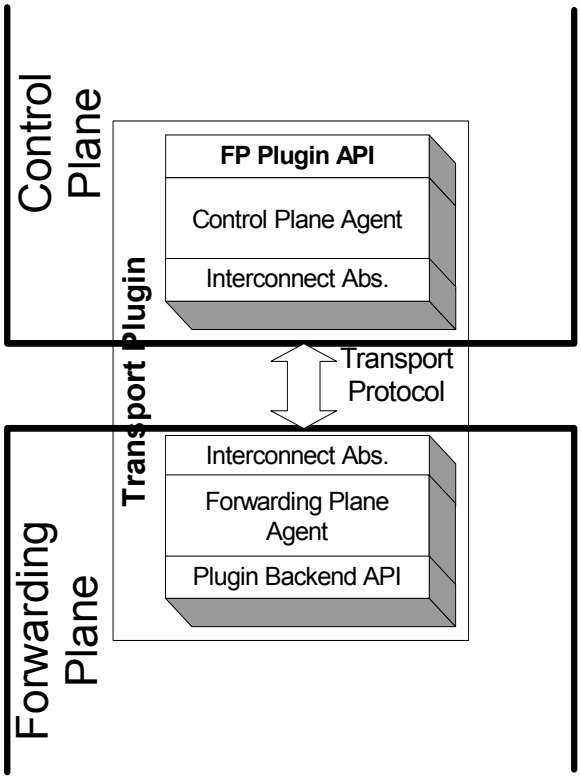


Figure 1: Transport plug-in architecture

Transport plug-in sends NPF API invocations from the CP to the FP. It is used by the FP for sending control data to the control plane for processing as well as data packets.

2.1 Forwarding Plane Plug-in API (FP Plug-in API)

Transport plug-in introduces the concept of a FP plug-in API in order to provide an abstraction to the CP-PDK. This API has been described in detail in [3]. This API allows the CP to send configuration and other control requests to the FP, receive the responses from the FP as well as send and receive data packets to and from the FP.

2.2 Plug-in Backend API

The API exposed by the transport plug-in on the forwarding plane, which is used by the FP module of the PDK. This API has been described in detail in [3]. This API allows the FP module to receive configuration and other requests from the CP, respond to those requests, as well as send and receive data packets, such as, RIP and OSPF to and from the CP.

2.3 Transport Protocol

This can be an IETF standard protocol like ForCES/COPS/GSMP or any other messaging system such as CORBA that can be used for transporting the messages between the control and the forwarding plane. The transport protocol implementation consists of the CP agent and the FP agent.

2.3.1 Control Plane Agent

The control plane agent implements the transport plug-in specific transport protocol and the messaging. It is invoked by the FP plug-in API and converts the API calls to wire format messages, sent to the forwarding plane agent.

2.3.2 Forwarding Plane Agent

This agent sits on the forwarding plane, parses the transport protocol messages and generates well-known messages which are used by the forwarding plane module to invoke the vendor specific API for the forwarding plane.

2.4 Interconnect Abstraction Layer

This provides an abstraction layer that hides the interconnect technology details from the transport protocol. The transport protocol uses this layer to send and receive messages without knowing whether the interconnect is PCI, Infiniband, Ethernet, or some other interconnect.

Part 3: Transport Protocol Design

3 Transport Protocol Design

This section describes the transport protocol design. The protocol can be considered as a preliminary implementation of the ForCES protocol and is based on the requirements [4], framework [6] and FE model [5] being defined in the ForCES working group in the IETF. The protocol described in the sections below is named the FLEX protocol.

3.1 Overview

The ForCES protocol referred to as the FLEX protocol is designed to be a simple, stateless, request-response protocol between the control and forwarding elements in a system. The protocol is designed to be lightweight in terms of low message parsing overhead as well as small message sizes. The protocol has a fixed length header that is 8-bytes long; all messages are 32-bit aligned. The protocol is easily extensible in several ways. It allows for a separate data model [5], which will define the data that needs to be exchanged.

It allows different encapsulation methods, such as, TLV, BER, XML, for both the control messages and the data packets being carried. A separate data channel, such as, GRE tunnel, can be established to exchange only data packets between control and forwarding elements. It encourages the use of TLV encapsulation for control messages since it has the lowest overhead. The protocol supports different interconnect technologies by allowing different encapsulations to be defined for different interconnects.

It assumes a reliable transport mechanism for the control channel. It has been designed to provide message level acknowledgements. The FLEX protocol meets all the requirements for separation of control and forwarding elements defined in [4] including command bundling, message priority, dynamic association and failover support.

3.2 Protocol Operation

The information exchanged between the CE and FE using the FLEX protocol in the CP-PDK can be classified into three phases. First is the binding phase, second is the capability & topology discovery phase, and the third is the configuration/normal operation phase. The following figure shows the information exchange.

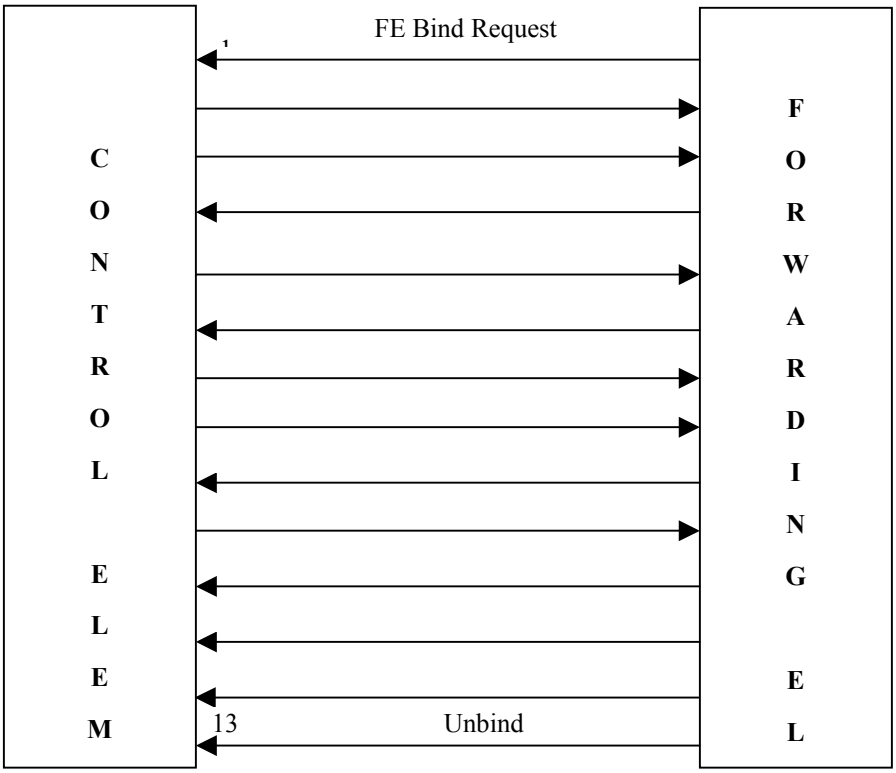


Figure 2: CE-FE information exchange

In the binding phase, the FE sends a bind request to the CE, which sends back a bind response to the FE. The bind response indicates whether the bind was successful or not. During this phase the encapsulation information is exchanged between the CE and FE, which might lead to the creation of a separate data channel, such as GRE tunnel, for the exchange of data packets only between the CE and FE.

In the capability discovery phase, the CE sends a capability request to the FE, which sends back a response with its capability information to the CE. The CE sends a topology request and the FE responds with its topology information relative to other FEs. If the CE is fine with the FE capabilities and topology and is ready to control and configure the FE, it sends a start operation message to the FE. Only after this message is received can the FE report events or send packets to the CE. The heartbeat message exchange starts after this message is sent. If the CE is not capable of controlling or configuring the FE based on the FE's capabilities or topology, it would send an unbind message to the FE at this point.

In the configuration operation phase, configuration and query messages are sent from the CE to the FE. The FE sends back the appropriate responses to the CE. Asynchronous FE events, such as port down event, are reported to the CE. Packet redirection between the CE and the FE takes place that is, control packets such as RIP, OSPF messages are redirected to the CE from the FE and vice-versa. Heartbeat messages are exchanged between the CE and FE according to the interval set during the binding phase.

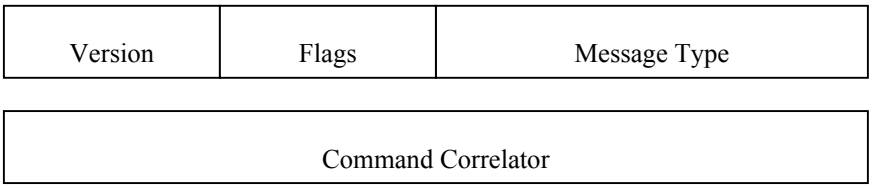
Finally during the shutdown process, the FE or CE send an unbind message to the other which ends their association.

3.3 Protocol Headers and Messages

The ForCES or FLEX protocol headers, commands, and messages are described as follows.

3.3.1 FLEX Protocol Header

The ForCES protocol has a fixed length header, which appears as follows:




```
typedef struct header_tag {
      Uint8_t     version;
      Uint8_t     flags;
Uint16_t    msg_type;
      Uint32_t    cmd_correlator;
} header;
```

The fields in the header are:

`Version : 8 bits`

This field defines the version of the FLEX protocol.

`Flags : 8 bits`

This field defines any flags for the protocol message.

Flags could be used to indicate that protocol reliability or responses to certain messages are not mandatory. This field could be used to indicate the priority of the ForCES message.

The valid values for this field are: normal priority, high priority, low priority, passive message, and no acknowledgment

`Msg_type: 16 bits`

This field defines the message type. The valid values for this field are: FE bind request, FE bind response, capability request, capability response, topology request, topology response, FE start operation, configuration/query request, configuration/query response, FE event/packet notification, unbind, and heartbeat.

`Cmd_correlator: 32 bits`

This field is used to distinguish between responses of multiple outstanding requests of the same type.

A value of 0 is reserved for bind, unbind and capability messages.

3.3.2 FE Binding

The FE binding phase consists of the FE sending a bind request to the CE, which responds with a bind response. The response indicates whether the CE accepts or rejects the bind request. Based on the CE response, any separate data channel for communication between CE and FE would be established after this phase. Communication using this channel would only start after the start operation command is issued by the CE to the FE.

The ForCES bind request appears as follows:

Version = 1	Flags = 0	Message Type = Bind Req
Command Correlator = 0		
FEID		
Control Encapsulation Type = TLV	Data Encapsulation Type = GRE	

```
typedef struct bindinfo_tag {
      Uint32_t    feid;
Uint16_t    control_encapsulation_type;
Uint16_t    data_encapsulation_type;
Uint32_t    bind_status; /* optional */
Uint32_t    heartbeat_interval; /* optional */
} bindinfo;
```

The fields in the bind request are:

`Feid : 32 bits`

This field uniquely identifies an FE.

Control Encapsulation_type : 16 bits

This field defines the encapsulation method for ForCES control messages, which is supported by the FE. The valid values for this field are: TLV, BER, and XML.

Data Encapsulation_type : 16 bits

This field defines the encapsulation method for the data packets, which is supported by the FE.

The valid values for this field are: TLV, BER, XML, GRE protocol, and IP-in-IP protocol.

`Bind Status : 32 bits`

This is an optional field, which defines the status of the FE bind.

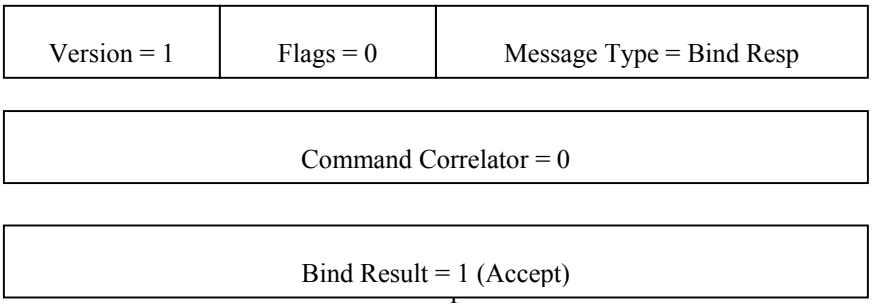
The valid values for this filed are:

`Active`
`Passive`

`Heartbeat Interval : 32 bits`

This is an optional field, which defines the interval in milliseconds at which heartbeat messages should be exchanged between the CE and FE.

The ForCES bind response appears as follows:



`Bind_result : 32 bits`

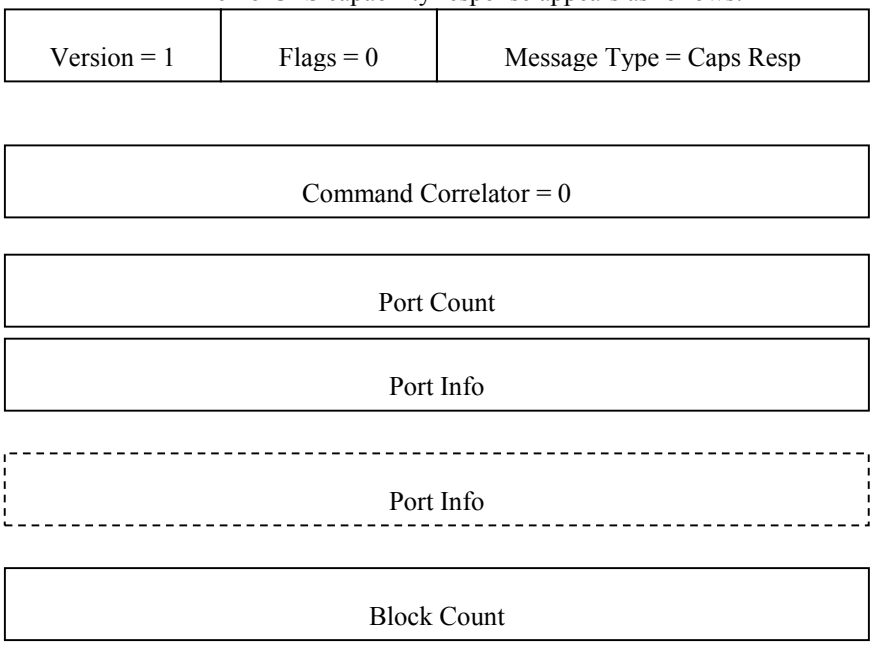
This field defines whether the FE bind request was successful or not. The valid values for this field are:

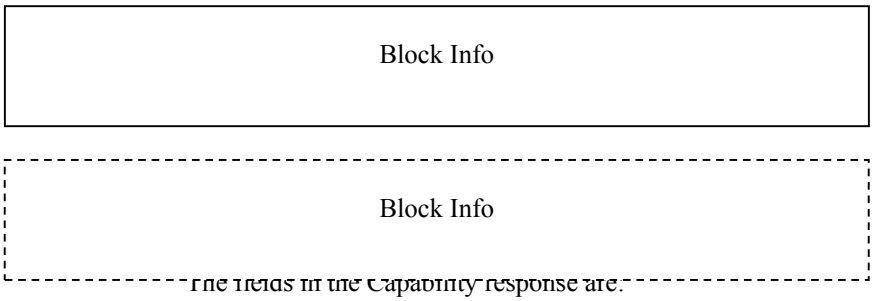
`Accept`
`Reject`

3.3.3 FE Capability Discovery

The FE capability discovery phase consists of the CE sending a capability request message to the FE, which responds with a capability response message. The capability request message consists of the common header with the message type set to capability request. The capability response message consists of the common header along with information about the FE Ports as well as the logical blocks [5]. The port information consists of the number of ports followed by an array of the `port_info structs`. The block information consists of the number of blocks followed by an array of the `block_info` structs.

The ForCES capability response appears as follows:





The fields in the Capability response are:

`Port Count : 32 bits`

This field defines the number of ports on the FE.

`Port Info : 64 bits`

This field defines the port information for each port on the FE and consists of a 32-bit field that defines a unique port identifier followed by a 32-bit field that defines the port type.

```
typedef struct portInfo_tag {
      Uint32_t    port_id;
      Uint32_t    port_type;
} portInfo_t;

typedef struct portlist_tag {
      Uint32_t    port_count;
      portInfo_t  *portArray;
} portlist_t;
```

`Block Count: 32 bits`

This field defines the number of logical blocks that exist on the FE. The blocks represent the logical functionality, or capabilities of the FE, see [5].

`Block Info: variable`

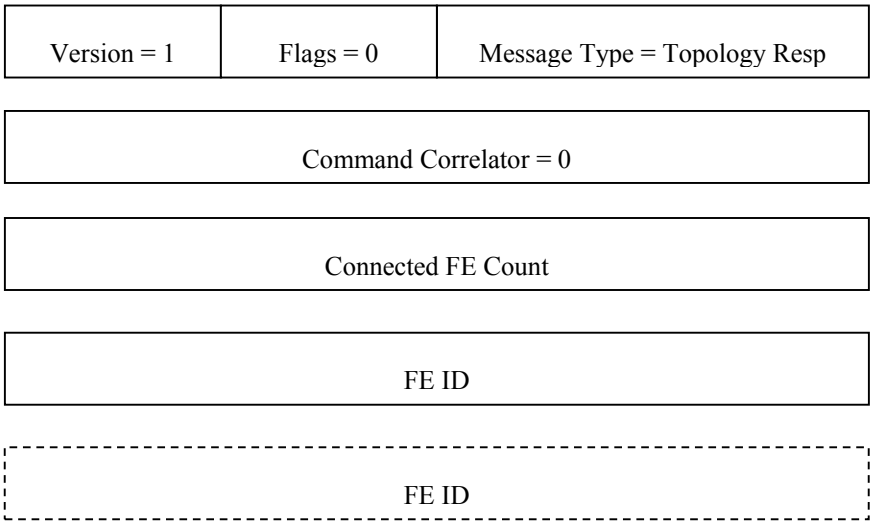
This field defines the block information for each logical functional block on the FE and consists of a 32-bit field that defines the block type followed by a 32-bit field that defines a unique block identifier or handle.

```
typedef struct blockInfo_tag {
      Uint32_t    block_type;
      Uint32_t    block_handle;
      Uint32_t    downstreamBlockCount;
      Uint32_t    *downstreamBlockArray;
} blockInfo_t;
typedef struct blocklist_tag {
      Uint32_t          block_count;
      blockInfo_t       *BlockArray;
} blocklist_t;
```

3.3.4 FE Topology Discovery

The FE topology discovery phase consists of the CE sending a topology request message to the FE, which responds with a topology response message. The topology request message consists of the common header with the message type set to topology request. The Topology response message consists of the common header along with information about the FEs directly connected to the communicating FE. This information consists of the number of directly connected FEs followed by an array of the FE identifiers.

The ForCES topology response appears as follows:




```
typedef struct connected_FElist_tag {
      Uint32_t    fe_count;
      Uint32_t    *feidArray;
} connected_FElist_t;
```

The fields in the topology response are:

`FE Count : 32 bits`

This field defines the number of FEs directly connected to the FE communicating with the CE.

`FE ID : 32 bits`

This field defines the unique FE identifier for each FE directly connected to the communicating FE.

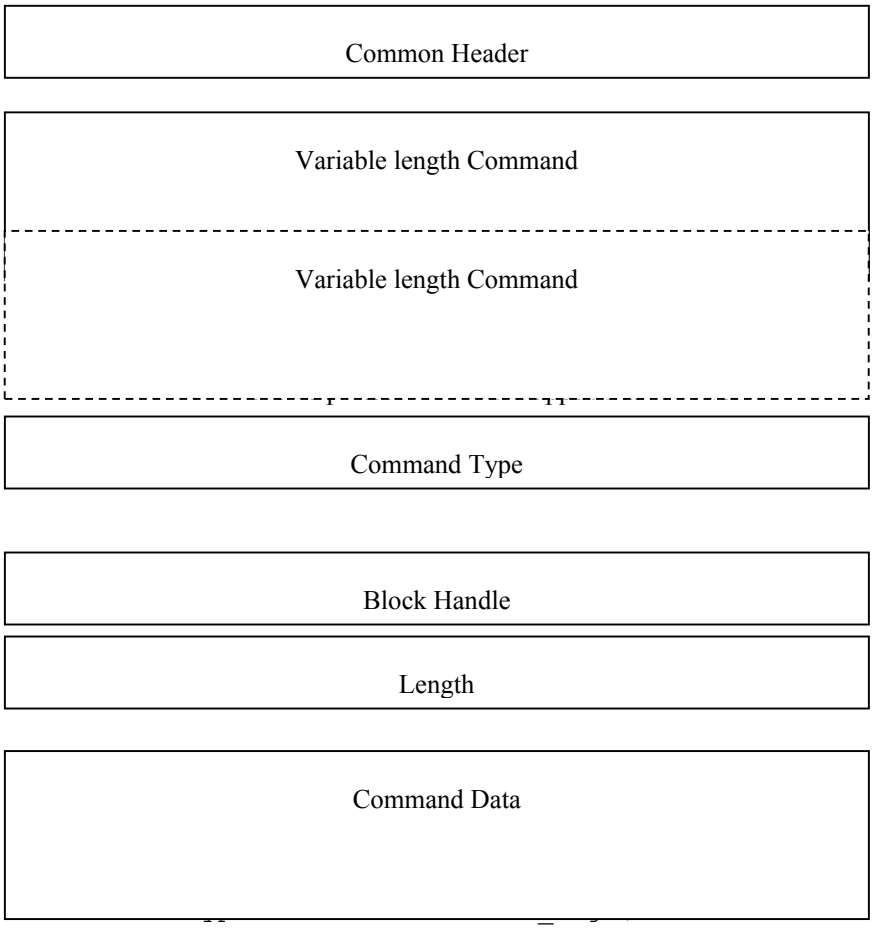
3.3.5 FE Start Operation

Once the capability discovery and topology phase of the protocol is complete, the CE sends the FE start operation message. This message indicates that the CE is fine with the FE capabilities and topology and is ready to control and configure the FE. This indicates that the CE is ready to receive messages from the FE such as event notification and packet redirection. The exchange of the heartbeat messages begins after this. If the CE is not capable of controlling or configuring the FE based on its capabilities or topology, it would send an unbind message to the FE at this point.

3.3.6 FE Configuration/Query Messages

The FE configuration or query messages are exchanged during the operational phase. They are in the form of a request that is sent from the CE to the FE to configure certain blocks/ FE functionality [5] or to query information. The FE sends back a response, which indicates the result of the configuration request or the information requested by the query. They consist of the fixed length header followed by one or more variable length commands. The protocol supports the command bundling requirement.

The ForCES configuration/query messages appear as follows:




```
Uint32_t    cmd_type;
      Uint32_t    block_handle;
      Uint32_t    length;
      Void*       cmd_data;
} command;
```

The fields in the command are:

`Cmd_type : 32 bits`

This field defines the command type. The valid values for this field are: null, add, update, delete, delete all, send packet, query statistics, and query properties.

`Block_handle : 32 bits`

This field defines the block handle or block identifier for which this command is being issued.

`Length : 32 bits`

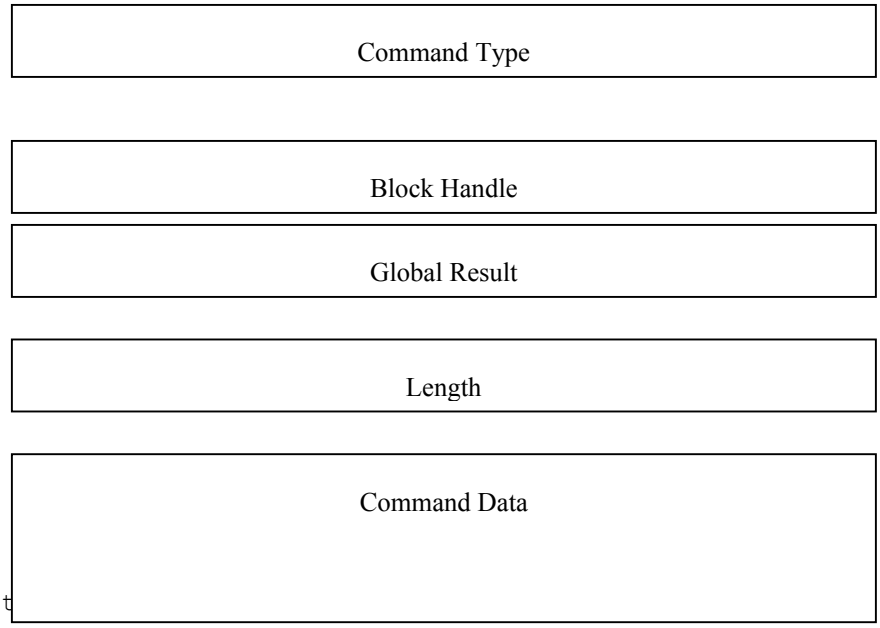
This field defines the length of the command data in bytes that is encapsulated in the command.

`Cmd_data : variable length`

This is the configuration/query data, which is encapsulated using the method negotiated during FE bind phase of the protocol. For the FLEX protocol implementation, the command data is encapsulated as a TLV structure. During the configuration/query request, this structure is essentially the NPF data structure that is passed from the CP module to the FP Plug-in API. For example, in case of a Add_IPv4_NextHop() call, the NPF_IPv4_NextHop structure which is passed in the call will be copied in this field.

In general, this field will contain the Block structures defined in the ForCES FE Model [5]. In the configuration/ query response, the command data will contain the result of the configuration or the query information again in the form of NPF data structures, which are passed by the FP module to the Backend API.

The ForCES protocol command status (response) appears as follows:



t

```
      Uint32_t    cmd_type;
      Uint32_t    block_handle;
      Uint32_t    global_result;
      Uint32_t    length;
      Void*       cmd_data;
} command_resp;
```

Block_handle : 32 bits

This field defines the block handle or block identifier for which this command is being issued.

Length : 32 bits

This field defines the length of the command data in bytes that is encapsulated in the command.

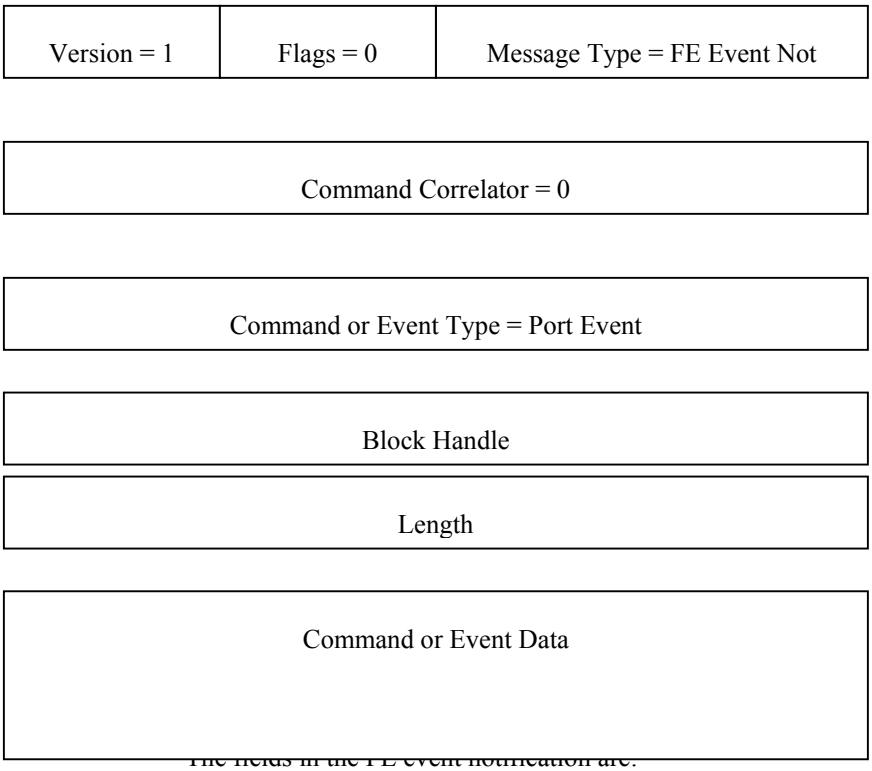
Global_result : 32 bits

This field defines the global result of the command. The individual results will be part of the command data.

3.3.7 FE Events/Packet Redirection

The FE events, such as, port down or change in certain capabilities, are reported to the CE using the FE event notification message. The packets being redirected to the CE from the FE are sent using this message. It is similar to the configuration/query messages as in it consists of the common header followed by one or more variable length commands or events.

The ForCES FE event notification appears as follows:



The fields in the FE event notification are:

Event_type : 32 bits

This field defines the event type. The valid values for this field are

11 = Port Event

12 = Block Specific Event

13 = Packet Redirection

14 = Capability Event

The ForCES FE packet redirection appears as follows:

Version = 1	Flags = 0	Message Type = FE Event Not.

Command Correlator = 0

Event Type = Packet Redirection

Block Handle

Length

Packet + metadata

3.3.8 CE, FE Unbinding

The CE or FE can send an unbind message to the other at any time to end their association.

The unbind message appears as follows:

Version = 1	Flags = 0	Message Type = Unbind

Command Correlator = 0

Reason (optional)

FE Behavior (CE unbind)

FE behavior : 32 bits

This field defines the behavior of the FE after the unbind occurs. The valid values for this field are:

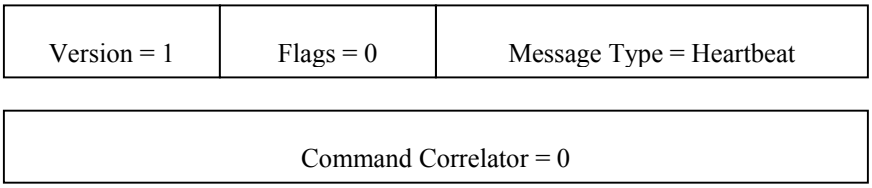
1 = Continue Operation

2 = Stop Operation

3.3.9 Heartbeat

The heartbeat is an optional message, which is exchanged between the CE and FE according to the interval set during FE binding. It is used to detect failure in communication between the CE and FE and helps the fast failover mechanism.

The heartbeat message appears as follows:



3.4 Failover Support

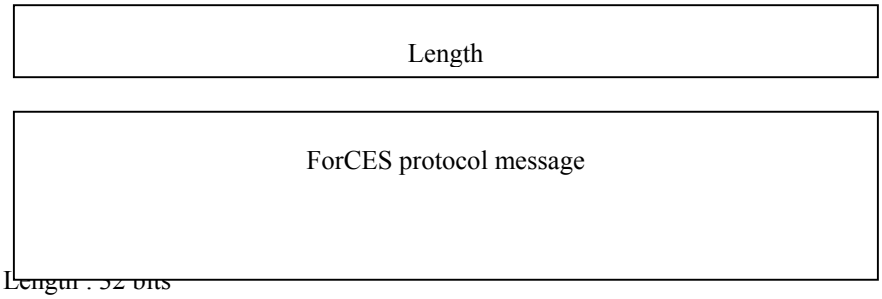
The ForCES protocol provides support for redundant control elements in the CP-PDK architecture and fast failover between primary and secondary CEs in case of failure. In order to provide this support, the protocol provides a failure detection mechanism using heartbeat messages, which can be used to detect any failure in communication between the control and FE.

3.5 Protocol Encapsulations

There are several encapsulations defined for ForCES protocol messages to work over different interconnect technologies. The interconnect technologies can consist or IP-centric technologies such as TCP/IP over Ethernet or non-IP centric such as PCI or Infiniband.

3.5.1 ForCES protocol Encapsulation for TCP

The ForCES protocol encapsulation for TCP appears as follows:



Length : 32 bits

This defines the length of the entire ForCES protocol message in bytes including the header.

Part 4: Interconnect Abstraction Layer Design

4 Interconnect Abstraction Layer Design

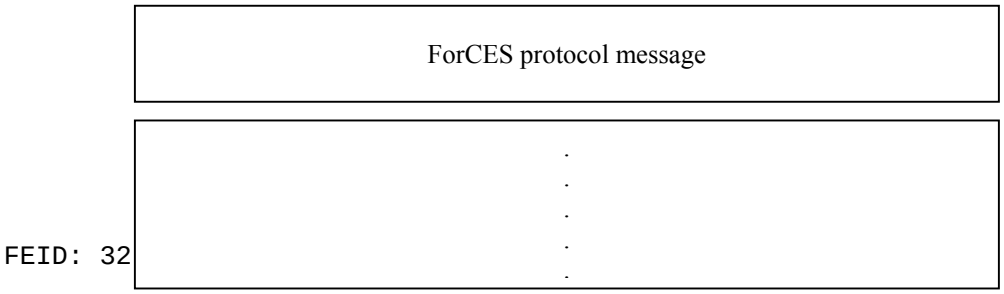
This section describes the design details of the interconnect abstraction layer. It is based on the COPS portability layer specification defined in [7]. It has certain features such as the packet buffer manager, which help in enhancing the performance of the transport plug-in implementation.

4.1 Packet Buffer Management

This module helps in reducing the cost or performance penalty of memory related operations, such as malloc, in the transport plug-in. A chunk of memory is pre-initialized and divided into equal sized buffers. One or more buffers depending on the size requested by the transport plug-in APIs are made available.

```
typedef struct _plBufMem {
    uint32_t total_Q_size;
    uint32_t buffers_used;
    unsigned char *pBufMem;
    uint32_t pBufFreeMem;
    uint32_t pBufFreeTail;
    uint32_t pBufSend;
    uint32_t pSendTail;
    uint32_t pBufRecv;
    uint32_t pRecvTail;
} plBufMem;
typedef struct _plBufHeader {
    uint32_t            connect_id;
    uint32_t            flags;
} plBufHeader;
typedef struct _plHeader {
    uint32_t            cookie;
    uint32_t            length;
} plHeader;
typedef struct _plBuf {
    plBufHeader pbufheader;
    plHeader plheader;
    unsigned  char      pbuf[0];
} plBuf;
```

The following shows the packet buffer encapsulated and ready to be delivered over interconnect:



This defines the identifier the interconnect uses to send and receive to the correct FE. The field is used and filled by the interconnect; this part of the buffer is not delivered across the interconnect.

`FLAGS: 32 bits`

This defines the buffer management flags (FREE/UNCHAINED/CHAINED). CHAINED is used when more than one consecutive buffers have been chained when size requested is more than a single buffer. This part of the buffer is not delivered across the interconnect.

`MAGICCOOKIE: 32 bits`

Reserved or used to identify a valid transport plug-in interconnect message. Delivered across the interconnect

`Length: 32 bits`

This defines the length of the ForCES protocol message. Delivered across the interconnect.

Table 3. Packet buffer encapsulation table

Function	Description
`uint32_t pb_getBuffer ( uint32_t FEID, uint32_t size, uint32_t flags)`	Returns a pointer to the Buffer of size
`uint32_t pb_freeBuffer( Void * buf)`	Free Buffer to be called by the layer that no longer passes the buffer to another layer
`Uint32 pb_PlBufferInit( Uint32 Qsize)`	Initialize Buffer API
`Uint32 pb_PlBufferDeInit( void)`	De-initialize Buffer API

4.2 Datagram API

This module provides a generic API to transport datagram based messages over a reliable connection. The transport plug-in will use the datagram API and the packet buffer management

API to send and receive messages. Datagram API implementation could use TCP/IP for reliable and fast delivery of messages over the networked CE/FE.

```
typedef void (*CB_DatagramReceive)(void *pbuf, uint32_t
connect_id, uint32_t length);
typedef void (*CB_DatagramException)(uint32_t ex, uint32_t
connect_id);

typedef struct _CBServerDatagramParams {
        CB_DatagramRecieve    datagram_receive;
        CB_DatagramException  datagram_exception;
} CBServerDatagramFuncs;

typedef struct _datagram_connect {
                uint32_t                flags;
                union {
                        struct sockaddr_in      ipv4_addr;
                        struct sockaddr_in6     ipv6_addr;
                // Place holder for the other connection
protocols.
                        }               saddr;
                }datagram_connect;

typedef struct _CBClientDatagramParams {
                datagram_connect      connect_info;
                CB_DatagramReceive      datagram_receive;
                CB_DatagramException    datagram_exception;
                } CBClientDatagramFuncs;
```

Table 4. Datagram API function table

Function	Description
`Uint32 InitializeDatagramServer (CBServerDatagramFuncs)`	Initialize Datagram API for server
`Uint32 InitializeDatagramClient (CBClientDatagramFuncs, uint32_t)`	Initialize Datagram API for client
`Void SendDatagram(uint32_t, void* msg, uint32_t size, uint32_t flags)`	Send Client Datagram across the interconnect
`Void DeInitializeDatagram()`	De-initialize Datagram API

Part 5: Transport Plug-in Design

5 Transport Plug-in Design

This section describes the design and implementation details of the transport plug-in.

5.1 Overview

This layer is responsible for the controlling, initialization and shutdown of the protocol implementation. On the control plane, it converts structures from the FP plug-in API format or NPF format to the ForCES message formats. On the forwarding plane, it converts structures from the ForCES message formats to the backend API format or FP plug-in API format, which is understood by the FP module in the forwarding plane.

5.2 Memory Management

The memory management for the transport plug-in is the same as that described in FP Plug-in API specification [3].

5.3 Threading Model

The transport plug-in has many threads. Most of these threads reside in the interconnect layer, there are three which always exist: the listen, send, and receive threads. The listen thread accepts incoming connections and binds from FE's. The send and receive threads which handle buffer flow. There is one thread for every FE connected, which reads packets from the network and places them in the receive thread's queue.

The transport plug-in creates a heartbeat thread for each FE when it binds. The heartbeat thread sends heartbeat messages at a negotiated rate.

These threads are initialized and provided by the portability layer.

5.4 Timeout Mechanism

This layer provides a timeout mechanism to help make the PDK design more robust. This allows the control plane PDK to set a timeout interval for each request sent to the forwarding plane. If the FP does not send a response within a certain time interval, the transport plug-in informs the CP-PDK.

5.5 Data Structures

On the control plane, the transport plug-in maintains a list of connected FEs. It uses the FEList to maintain the mapping of `FEID`, which is the FE identifier generated by the transport plug-in, to connectionid, which is used by the interconnect layers to identify the connection to the FE or client. The FEInfo structure, which is stored in the list is shown below.

```
typedef struct FEInfo_t{
```

```
      FPPI_FEID   feid;
      uint32_t    connectionid;
      forces_bind_info_t      bindinfo;  /* all the information */
      forces_portlist_t*      portlist;  /* about the FE       */
      forces_blocklist_t*     blocklist;
      FPPI_FE_Caps*           fecaps;
      DList       outstandingcommands;   /* commands sent to FE */
      PilThread heartbeatthread;         /* heartbeat info: thread
handle */
      int       msgpipe[2];              /* pipe to shut it down
*/
      int       missedbeats;             /* count of missed beats
from FE */
} *FEInfo;
```

It uses the outstandingcommands list to maintain the callback information required for all queries and commands. The CBInfo structure, which is stored in the list shown below.

```
typedef struct FPPCommand_t {
    FEInfo          feinfo;     /* FE this command belongs to */
    ForCESMessage   msg;        /* handle to message sent */
    uint32_t        cmd_type;   /* ForCES command */
    uint32_t        block_type; /* and block */
    char*           buffer;
    size_t          size;
    uint32_t        forces_correlator;    /* correlator sent with
this
command to FE */
    FPPI_CORRELATOR fppi_correlator;      /* correlator to return
to PDK */
    FPPI_CBHANDLE   fppi_cbhandle;        /* callback to invoke */
    FPPCommandHandler   handler;          /* function to invoke to
decap
any results */
} * FPPCommand;
```

5.6 Pseudo-Code for Control Plane

The Pseudo-Code for calls such as Initialize, Shutdown, Start, Stop, RouteAdd, RouteDel, that are exposed by FP Plug-in API on the control plane, has been shown below.

FPPAPI_Initialize

```
{
    // Initialize all state info lists
    npf_list_init(&ConnectList, PIL_FreeMemory);
    npf_list_init(&CBInfoList, PIL_FreeMemory);
    // Initialize portability layer
    return success;
```

```
}
```

FPPAPI_Shutdown

```
{
    // destroy all lists that were initialized
    npf_list_destroy(&ConnectList);
    npf_list_destroy(&CBInfoList);
    // De-initialize portability layer
    return success;
}
```

FPPAPI_Start

```
{
    // ready to receive any FE bind requests
    return success;
}
```

FPPAPI_Stop

```
{
    //send unbind message to all FEs
    return success;
}
```

FPPAPI_ipv4_unicastRouteAdd

```
{
    // Determine size of buffer needed to send command
    // Create new FPPCommand with:
    //   appropriate ForCES command and block for ipv4 route add,
    //   size required for buffer
    //   callback handle
    //   correlator
    //   and a handler
    // encapsulate parameters into FPPCommand’s buffer
    buffer = encapsulate_uint32_t(route_count, buffer);
    buffer = encapsulate_array_ipv4Route(route_list, route_count,
buffer);
    // Send command
    FPPCommandSend();
    return success;
}
```

5.7 Pseudo-Code for Forwarding Plane

The Pseudo-Code for calls such as Initialize, Shutdown, bindRequest, SendEvent, SendPacket, ReportStatus, which are exposed by Backend API on the forwarding plane, has been shown below.

BENDAPI_Initialize

```
{
    // Initialize all state info lists
    npf_list_init(&ReportList, PIL_FreeMemory);
    // Initialize portability layer
    return success;
}
```

BENDAPI_Shutdown

```
{
    // Destroy all lists that were initialized
    npf_list_destroy(&ReportList);
    // De-initialize portability layer
    return success;
}
```

BENDAPI_bindRequest

```
{
     // send FE bind message
    return success;
}
```

BENDAPI_unbindRequest

```
{
    // send FE unbind message
    return success;
}
```

BENDAPI_sendPacket

```
{
    // encapsulate packet list into ForCES message

    // send message to the CP
    return success;
}
```

BENDAPI_sendEvent

```
{
    // encapsulate event type & event data into ForCES message
    // send message to CP
    return success;
}
```

BENDAPI_Report_Status

```
{
    // search ReportList for cbtype, cbcorrelator

    // initialize Report message
```

```
    // check response_data if cbtype is GetProperties or
GetStatistics
    // otherwise encapsulate a success or failure report
    // send message to CP
    return success;
}
```

Part 6: Transport Plug-in Design

6 Code Generator Design

6.1 Code Generator Introduction

This section describes the design for the transport plug-in code generator, and the reasoning behind it.

The transport plug-in consists of two major parts: the core where all the state is maintained and all the ForCES communication takes place, and the APIs. The APIs expose the functionality of the forwarding plane to the control plane. Each function in an API serializes the command and any data from the control plane, sends them to the correct client, which then de-serializes the data and makes the appropriate calls into the forwarding plane. Then, any results are serialized and sent back to the server, which de-serializes the data and makes the appropriate callback into control plane.

6.2 Code Generator Requirements

The ultimate goal of the code generator is to take a transport plug-in API description, including functions and data types, and generate all the code required to build and send ForCES commands, and serialize and de-serialize all commands, data and results, for both the server and client sides of the transport plug-in. At this stage the focus is on generating the serialization.

The code generator must:

- Take as input a standard C header file, which contains all the data types that will need to be serialized. This C header file must be able to coexist and be used by the rest of the PDK. This will insure that there is no duplication of data type definitions, thereby reducing the chances of synchronization problems later, if the types were to change.
- Be able to do all the same preprocessing on the input file that the compiler will do when building the PDK. This insures that any code that is #ifdef'd is not included, or any macro substitution is performed.
- Generate all serialization, encapsulation/decapsulation routines for all types in the input file, and put them into appropriate `.c files`, and generate appropriate `.h files` for use by other `.c files`.
- Any changes to the input file required by the code generator must easy to write, human readable, and not impact any other `.c files` that may be including the input .h file.
- All output of the code generator must compile without any changes by the user

6.3 Code Generator Design Considerations

Given that the code generator cannot be omniscient, the input file must give it some hints about certain data types and fields.

For example, given this data type:

```
typedef struct {
      uint32_t num;
      uint32_t * arr;
} my_array;
```

What is the meaning of the field arr? Is it a pointer to a single uint32_t, or is an array of length num uint32_t's? In order to instruct the code generator on how to treat array, the code needs to be marked up. For other C compilers, we have elected to use comments to hide our markups. Here is the above example, reworked to tell the code generator that array is an array.

```
typedef struct {
      uint32_t num;
      uint32_t * arr; /* @!array-length:num! */
} my_array;
```

The code generator knows that the `field arr` is an array of length num, and will encapsulate and decapsulate accordingly. Notice that the markup is entirely enclosed in a C style comment, effectively hiding it from the compiler.

Note: The special @!<command>:<value>! notation should prevent normal comments from interfering with the code generator.

The only other markup needed is for unions. Given any union, encapsulation and decapsulation of the appropriate field is the only motive. Here is an example of unions:

```
typedef struct {
      uint32_t type;
      union /* @!union-switch:type! */
      {
            uint32_t a;   /* @!union-case:0! */
            my_array b;   /* @!union-case:1! */
            foo      bar; /* @!union-case:GLOBAL_DEFINE! */
      } u;
} my_array;
```

The resulting encapsulation/decapsulation code produced by the code generator will then switch off of the field type, and if the result is 0 will encapsulate/decapsulate a uint32_t and store it in UA.

To avoid multiple copies of data types in header files and to avoid feeding extraneous information to the code generator, a developer can break existing header files into two parts. The main header file and a sub-header file that holds all the information, should be given to the code generator. For example:

npf_header.h:

```
typedef void* npf_context;
…various other things never encapsulated…
#include npf_header_remote_types.h
…more local stuff…
```

npf_header_remote_types.h:

```
#typedef uint32_t npf_correlator;
#typedef uint8_t npf_array_foo[SIZE_OF_FOO];
```

Only the `npf_header_remote_types.h` would be run through the code generator, or one could put `#ifdef` around only those types that need to have code generated and then make sure that the code generator has defined it.

6.4 Code Generator Design

Much of the design of the code generator is decided by the requirement that it uses existing C header files. The code generator must be able to parse C syntax, so the majority of the logic behind the code generator is focused on that.

6.4.1 Code Generator Parser Design

The code generator parser actually has two parts: the lexical analyzer and the parser. The code generator relies on the Lex and Yacc tools for these parts.

The basic design of the definition file given to Lex to build the lexical analyzer is:

```
%{
%}

ws      [ \t]+                /* white space */
id      [a-zA-Z][a-zA-Z0-9_]* /* identifiers, types etc */
size    [0-9]+                /* hard coded sizes of arrays */
command [a-z][a-z-]*          /* markup commands */
nl      [\n]                  /* newlines */


/* States */
%x COMMENT    /* c style comments */
%x CPPCOMMENT /* c++ style comments */
%x COMMAND    /* markup command */

%%

//            { BEGIN CPPCOMMENT; } /* start c++ comment */
<CPPCOMMENT>.   { }                   /* ignore all */
<CPPCOMMENT>\n  { BEGIN 0; }          /* ends at end of line */

/*            { BEGIN COMMENT; }    /* begin c comment */

<COMMENT>*/   { BEGIN 0; }          /* end comment */
<COMMENT>@!   { BEGIN COMMAND; }    /* begin markup command */
<COMMENT>\n     { }
<COMMENT>.      { }                   /* ignore everything else */
<COMMAND>{command} {                  /* return command */
        return ID;
        }
<COMMAND>{id} {                       /* return id for command */
        return ID;
        }
<COMMAND>{size} {                     /* return size for command */
        return SIZE;
```

```
        }
<COMMAND>!    { BEGIN COMMENT; }    /* end command, resume comment */
<COMMAND>{ws}   ;
<COMMAND>{nl}   { }
<COMMAND>.      { }

{ws}                                  /* ignore all whitespace */
{nl}    { }                           /* ignore newlines */
{id}    { }                           /* return an id */
{size}  { }                           /* return an size */
.       { }


%%
```

Other than some state maintenance for the comments and commands, the lexer is seems to be simpleer. The parser appears to be little complicated. Following is a basic design for the input to Yacc:

```
%{
%}

%start statements

%%

statements: /* statements are : */
        statements statement  /* many statements and a statement */
    |   statement             /* a statement */
    |                         /* nothing */
    ;

statement: /* a statement is : */
        definition            /* a type's definition */
    |   functiondef           /* a functions's definition */
    ;

definition:
        TYPEDEF modifieddeclaration
    ;

functiondef:
        type id '(' arglist ')' ';'
    ;

declaration: /* a declaration can be what is being typedef's or
                might be a field in a struct or union */
```

```
        /* basic type */
        type id ';'

        /* pointer to a basic type */
    |   type '*' id ';'

        /* hard core array.  already allocated within
           the struct. size is a value or define or something */
    |   type id '[' size ']' ';'

        /* a struct, contains multiple fields */
    |   STRUCT optid decllist id optarr ';'

        /* a union, like a struct, but different */
    |   UNION optid '@' id ':' size decllist optid ';'

        /* enums are basically ignored,
           but we need to remember them for later fields
           and encap/decap them as uint32_t's */
    |   ENUM optid '{' enumlist '}' id ';'

        /* functions and function pointers, not used yet
           but may be handy for auto generating api calls */
    |   type '(' '*' id ')' '(' arglist ')' ';'
    |   type '*' '(' id ')' ';'

optarr: /* incase of an array of structs */
        '[' size ']'  /* size of the array */
    |                 /* or nothing as it is optional */
    ;

type:   /* types may be modified, unsigned, long etc */
        modifier id
    |   modifier modifier   /* long long? */
    |   id
      ;

modifier:  /* possible modifiers */
        UNSIGNED
    |   SIGNED
    |   SHORT
    |   LONG
    ;

command:  /* markup commands */
```

```
        '@' id ':' size
    ;

declist:
        '{' declarationlist '}'
    ;

commandchain:  /* in future may allow multiple markup commands */
        commandchain command
    |   command
    |
    ;

modifieddeclaration:  /* markedup declaration */
        declaration commandchain
    ;

declarationlist:      /* a list of declaration is : */
    | declarationlist modifieddeclaration
    | modifieddeclaration
    |
    ;

optval: /* optional value in enum */
      '=' SIZE  /* = number, or id */
    |           /* or nothing */
    ;

enum: /* enum entry */
      id optval
    ;

enumlist:  /* list of enum entries */
      enum ',' enumlist
    | enum
    |
    ;

arglist: /* list of arguments to a function */
      arg
    | arglist ',' arg
    |
    ;

arg: /* argument to a function */
```

```
      type id
    | type '*' id
    ;

size: /* sizes might be hardcoded numbers, or identifiers */
      SIZE
    | id { $$ = $1; }
    ;

id: /* identifier, returned by lexer */
      ID { $$ = strdup($1); }
    ;

optid: /* optional id */
      id
    |
    ;

%%
```

Lex and Yacc can be run on the definitions files described by the above, generating the code that will parse the input files of the code generator. The required output of the parser is a list of type definitions. A type definition is defined as:

```
typedef struct type_t {
      char* name;     /* name of type */
      int typeid;     /* unique id for type */
      int kindoftype; /* regular, pointer, array, etc */
      char* size;     /* if array or pointer array,
                         might be 100 or MAX_SIZE_OF_ARR, etc */
      char* kase;     /* case kase: if union member */

      /* pointer to type this type is based on
         i.e. uint32_t or struct, or FOO */
      struct type_t* basetype;

      /* if this is a struct or union, this is a list of the
         fields comprising it */
      list fields;
} * itype;
```

6.5 Code Generator Code Generation

Once the parser builds the list of type definitions, it is time to start generating code. The code generator emits encapsulation, sizeof, and decapsulation functions for all entries in the list, as well as header files for those functions. All code generation is simplistic, following strict templates, and assumes the existence of encapsulation, decapsulation, or sizeof functions elsewhere for any unknown types. All

functions for basetypes have been handwritten earlier and are part of the core of the transport plug-in. General strategy of all encapsulation, decapsulation and sizeof functions is to reduce all types to their base types, and call the encapsulation, decapsulation, and sizeof functions of those base types. Following is an example of this principle:

Example type:

```
typedef struct {
      uint32_t x;
      char y;
      bar z;
} foo;
```

generated encapsulation function:

```
char* encapsulate_foo(
            foo* a,
            char* buf
            )
{
      buf = encapsulate_uint32_t(&(a->x), buf);
      buf = encapsulate_char(&(a->y),buf);
      buf = encapsulate_bar(&(a->z),buf);
      return buf;
}
```

Notice how the `encapsulate_foo` function calls the encapsulation functions for all the members of a `foo`, irrespective of their types, It assumes the encapsulation function exists and takes care of the encapsulation detail. Notice that the pointer to the buffer that is being encapsulated into is never directly manipulated, except through assigning it to the result of an encapsulation. The only functions that must know how much to move forward, in the buffer to encapsulate the next item, are the very base functions that have been built by hand.

Following is a little complicated example for a decapsulation function:

Example type:

```
typedef struct {
      uint32_t len;
      bar * arr; /* @!array-length:len! */
} foo;
```

generated decapsulation function:

```
char* decapsulate_foo(
            foo* a,
            char* buf
            )
{
      buf = decapsulate_uint32_t(&(a->len),buf);
      a->arr = (buf*)malloc(sizeof(bar)*a->len);
      buf = decapsulate_array_bar(a->bar,a->len,buf);
```

```
        return buf;
}
```

Notice that the decapsulation function needs to allocate space for the bar array, also known as arr. From the markup command array-length, the code generator knows that arr is a pointer to an array of len bar's. Notice that the function decapsulate_array_bar is called to decapsulate that array. The code generator produces not only encapsulation and decapsulation functions for all types, but functions to encapsulate and decapsulate entire arrays for all types.