Internet Draft
Expiration: July 2004
File: draft-ietf-forces-model-02.txt
Working Group: ForCES

L. Yang
Intel Corp.
J. Halpern
Megisto Systems
R. Gopal
Nokia
A. DeKok
IDT Inc.
Z. Haraszti
S. Blake
Ericsson
E. Deleganes
Intel Corp.

February 2004

ForCES Forwarding Element Model

draft-ietf-forces-model-02.txt

Status of this Memo

Abstract

This document defines the forwarding element (FE) model used in the Forwarding and Control Plane Separation (ForCES) protocol. The model represents the capabilities, state and configuration of

forwarding elements within the context of the ForCES protocol, so that control elements (CEs) can control the FEs accordingly. More specifically, the model describes the logical functions that are present in an FE, what capabilities these functions support, and how these functions are or can be interconnected. This FE model is intended to satisfy the model requirements specified in the ForCES requirements draft [1]. A list of the basic logical functional blocks (LFBs) is also defined in the LFB class library to aid the effort in defining individual LFBs.

Table of Contents

Conventions used in this document

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in [RFC-2119].

1. Definitions

A set of terminology associated with the ForCES requirements is defined in [1] and is not copied here. The following list of terminology is relevant to the FE model defined in this document.

FE Model -- The FE model is designed to model the logical processing functions of an FE. The FE model proposed in this document includes three components: the modeling of individual logical functional blocks (LFB model), the logical interconnection between LFBs (LFB topology) and the FE level attributes, including FE capabilities. The FE model provides the basis to define the information elements exchanged between the CE and the FE in the ForCES protocol.

Datapath -- A conceptual path taken by packets within the forwarding plane inside an FE. Note that more than one datapath can exist within an FE.

LFB (Logical Function Block) class (or type) -- A template representing a fine-grained, logically separable and well-defined packet processing operation in the datapath. LFB classes are the basic building blocks of the FE model.

LFB (Logical Function Block) Instance -- As a packet flows through an FE along a datapath, it flows through one or multiple LFB instances, with each implementing an instance of a certain LFB

class. There may be multiple instances of the same LFB in an FE's datapath. Note that we often refer to LFBs without distinguishing between LFB class and LFB instance when we believe the implied reference is obvious for the given context.

LFB Model -- The LFB model describes the content and structures in an LFB, plus the associated data definition. There are four types of information defined in the LFB model. The core part of the LFB model is the LFB class definitions; the other three types define the associated data including common data types, supported frame formats and metadata.

LFB Metadata -- Metadata is used to communicate per-packet state from one LFB to another, but is not sent across the network. The FE model defines how such metadata is identified, produced and consumed by the LFBs, but not how metadata is encoded within an implementation.

LFB Attribute -- Operational parameters of the LFBs that must be visible to the CEs are conceptualized in the FE model as the LFB attributes. The LFB attributes include, for example, flags, single parameter arguments, complex arguments, and tables that the CE can read or/and write via the ForCES protocol.

LFB Topology -- Representation of how the LFB instances are logically interconnected and placed along the datapath within one FE. Sometimes it is also called intra-FE topology, to be distinguished from inter-FE topology. LFB topology is outside of the LFB model, but is part of the FE model.

FE Topology -- A representation of how the multiple FEs within a single NE are interconnected. Sometimes this is called inter-FE topology, to be distinguished from intra-FE topology (i.e., LFB topology). An individual FE might not have the global knowledge of the full FE topology, but the local view of its connectivity with other FEs is considered to be part of the FE model. The FE topology is discovered by the ForCES base protocol or some other means.

Inter-FE Topology -- See FE Topology.

Intra-FE Topology -- See LFB Topology.

LFB class library -- A set of LFB classes that is identified as the most common functions found in most FEs and hence should be defined first by the ForCES Working Group.

2. Introduction

[2] specifies a framework by which control elements (CEs) can configure and manage one or more separate forwarding elements (FEs) within a networking element (NE) using the ForCES protocol. The ForCES architecture allows Forwarding Elements of varying functionality to participate in a ForCES network element. The implication of this varying functionality is that CEs can make only minimal assumptions about the functionality provided by FEs in an NE. Before CEs can configure and control the forwarding behavior of FEs, CEs need to query and discover the capabilities and states of their FEs. [1] mandates that the capabilities, states and configuration information be expressed in the form of an FE model.

RFC 3444 [11] made the observation that information models (IMs) and data models (DMs) are different because they serve different purposes. "The main purpose of an IM is to model managed objects at a conceptual level, independent of any specific implementations or protocols used". "DMs, conversely, are defined at a lower level of abstraction and include many details. They are intended for implementors and include protocol-specific constructs." Sometimes it is difficult to draw a clear line between the two. The FE model described in this document is first and foremost an information model, but it also includes some aspects of a data model, such as explicit definitions of the LFB class schema and FE schema. It is expected that this FE model will be used as the basis to define the payload for information exchange between the CE and FE in the ForCES protocol.

2.1. Requirements on the FE model

[1] defines requirements, which must be satisfied by a ForCES FE model. To summarize, an FE model must define:

- Logically separable and distinct packet forwarding operations in an FE datapath (logical functional blocks or LFBs);
- The possible topological relationships (and hence the sequence of packet forwarding operations) between the various LFBs;
- The possible operational capabilities (e.g., capacity limits, constraints, optional features, granularity of configuration) of each type of LFB;
- The possible configurable parameters (i.e., attributes) of each type of LFB;
- Metadata that may be exchanged between LFBs.

2.2. The FE Model in Relation to FE Implementations

The FE model proposed here is based on an abstraction of distinct logical functional blocks (LFBs), which are interconnected in a

directed graph, and receive, process, modify, and transmit packets along with metadata. Note that a real forwarding datapath implementation should not be constrained by the model. On the contrary, the FE model should be designed such that different implementations of the forwarding datapath can all be logically mapped onto the model with the functionality and sequence of operations correctly captured. However, the model itself does not directly address the issue of how a particular implementation maps to an LFB topology. It is left to the forwarding plane vendors to define how the FE functionality is represented using the FE model. Nevertheless, we do strive to design the FE model such that it is flexible enough to accommodate most common implementations.

The LFB topology model for a particular datapath implementation MUST correctly capture the sequence of operations on the packet. Metadata generation (by certain LFBs) must always precede any use of that metadata (by subsequent LFBs in the topology graph); this is required for logically consistent operation. Further, modifications of packet fields that are subsequently used as inputs for further processing must occur in the order specified in the model for that particular implementation to ensure correctness.

2.3. The FE Model in Relation to the ForCES Protocol

The ForCES base protocol is used by the CEs and FEs to maintain the communication channel between the CEs and FEs. The ForCES protocol may be used to query and discover the inter-FE topology. The details of a particular datapath implementation inside an FE, including the LFB topology, along with the operational capabilities and attributes of each individual LFB, are conveyed to the CE within information elements in the ForCES protocol. The model of an LFB class should define all of the information that would need to be exchanged between an FE and a CE for the proper configuration and management of that LFB.

Definition of the various payloads of ForCES messages (irrespective of the transport protocol ultimately selected) cannot proceed in a systematic fashion until a formal definition of the objects being configured and managed (the FE and the LFBs within) is undertaken. The FE Model document defines a set of classes and attributes for describing and manipulating the state of the LFBs of an FE. These class definitions themselves will generally not appear in the ForCES protocol. Rather, ForCES protocol operations will reference classes defined in this model, including relevant attributes (and operations if such are defined).

Section 8 provides more detailed discussion on how the FE model should be used by the ForCES protocol.

2.4. Modeling Language for FE Model

Even though not absolutely required, it is beneficial to use a formal data modeling language to represent the conceptual FE model described in this document and a full specification will be written using such a data modeling language. Using a formal language can help to enforce consistency and logical compatibility among LFBs. In addition, the formal definition of the LFB classes has the potential to facilitate the eventual automation of some part of the code generation process and the functional validation of arbitrary LFB topologies.

Human readability was the most important factor considered when selecting the specification language. Encoding, decoding and transmission performance was not a selection factor for the language because the encoding method for over the wire transport is an issue independent of the specification language chosen. It is outside the scope of this document and up to the ForCES protocol to define.

XML was chosen as the specification language in this document, because XML has the advantage of being both human and machine readable with widely available tools support.

2.5. Document Structure

Section 3 provides a conceptual overview of the FE model, laying the foundation for the more detailed discussion and specifications in the sections that follow. Section 4 and 5 constitute the core of the FE model, detailing the two major components in the FE model: LFB model and FE level attributes including capability and LFB topology. Section 6 presents a list of LFB classes in the LFB class library that will be further specified in separate documents according to the FE model presented in Sections 4 and 5. Section 7 directly addresses the model requirements imposed by the ForCES requirement draft [1] while Section 8 explains how the FE model should be used in the ForCES protocol.

3. FE Model Concepts

Some of the important concepts used throughout this document are introduced in this section. Section 3.1 explains the difference between a state model and a capability model, and how the two can be combined in the FE model. Section 3.2 introduces the concept of

LFBs (Logical Functional Blocks) as the basic functional building blocks in the FE model. Section 3.3 discusses the logical inter-connection and ordering between LFB instances within an FE, that is, the LFB topology.

The FE model proposed in this document is comprised of two major components: LFB model, and FE level attributes including FE capabilities and LFB topology. The LFB model provides the content and data structures to define each individual LFB class. FE attributes provide information at the FE level and the capabilities about what the FE can or cannot do at a coarse level. Part of the FE level information is the LFB topology which expresses the logical inter-connection between the LFB instances along the datapath(s) within the FE. Details on these components are described in Section 4 and 5. The intention of this section is to discuss these concepts at the high level and lay the foundation for the detailed description in the following sections.

3.1. State Model and Capability Model

The FE capability model describes the capabilities and capacities of an FE in terms of variations of functions supported or limitations contained. Conceptually, the FE capability model presents the many possible states allowed on an FE with capacity information indicating certain quantitative limits or constraints. For example, an FE capability model may describe the FE at a coarse level such as:

- this FE can handle IPv4 and IPv6 forwarding;
- this FE can perform classification on the following fields: source IP address, destination IP address, source port number, destination port number, etc;
- this FE can perform metering;
- this FE can handle up to N queues (capacity);
- this FE can add and remove encapsulating headers of types including IPSec, GRE, L2TP.

On the other hand, an FE state model describes the current state of the FE, that is, the instantaneous values or operational behavior of the FE. The FE state model presents the snapshot view of the FE to the CE. For example, using an FE state model, an FE may be described to its CE as the following:

- on a given port the packets are classified using a given classification filter;
- the given classifier results in packets being metered in a certain way, and then marked in a certain way;

- the packets coming from specific markers are delivered into a shared queue for handling, while other packets are delivered to a different queue;
- a specific scheduler with specific behavior and parameters will service these collected queues.

The information on the capabilities and capacities of the FE helps the CE understand the flexibility and limitations of the FE functions, so that the CE knows at a coarse level which configurations are applicable to the FEs and which ones are not. It gets more complicated for the capability model to cope with the detailed limits, such as the maximum number of the following items: classifiers, queues, buffer pools, and meters the FE can provide.

While one could try to build an object model to fully represent the FE capabilities, other efforts have found this to be a significant undertaking. A middle of the road approach is to define coarse-grained capabilities and simple capacity measures. Then, if the CE attempts to instruct the FE to set up some specific behavior it is not capable of, the FE will return an error indicating the problem. Examples of this approach include Framework Policy Information Base (PIB) [RFC3318) and Differentiated Services QoS Policy Information Base [4]. The capability reporting classes in the DiffServ and Framework PIBs are all meant to allow the device to indicate some general guidelines about what it can or cannot do, but do not necessarily allow it to indicate every possible configuration that it can or cannot support. If a device receives a configuration that it cannot implement, it can reject that configuration by responding with a failure report.

Figure 1 shows the concepts of FE state, capabilities and configuration in the context of CE-FE communication via the ForCES protocol.

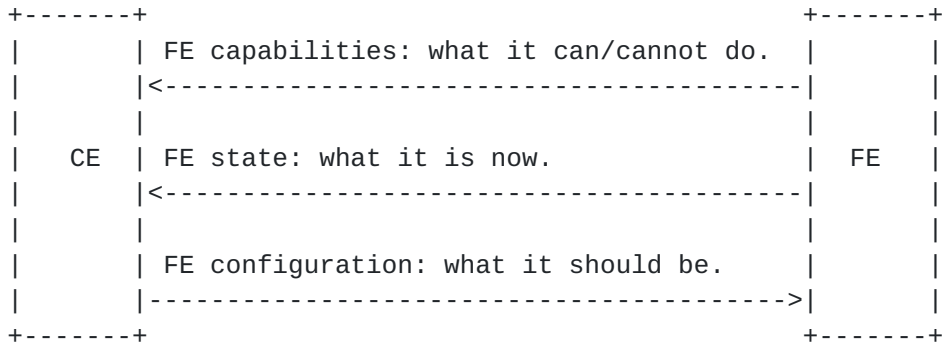


Figure 1. Illustration of FE state, capabilities and configuration exchange in the context of CE-FE communication via ForCES.

The ForCES FE model must include both a state model and a capability model. We believe that a good balance between simplicity and flexibility can be achieved for the FE model by combining the coarse level capability reporting with the error reporting mechanism. Examples of similar approaches include DiffServ PIB [4] and Framework PIB [5].

The concepts of LFB and LFB topology will be discussed in the rest of this section. It will become clear that a capability model is needed at both the FE level and LFB level.

Capability information at the LFB level is an integral part of the LFB model, and is modeled the same way as the other operational parameters inside an LFB. For example, certain features of an LFB class may be optional, in which case it must be possible for the CE to determine whether or not an optional feature is supported by a given LFB instance. Such capability information can be modeled as a read-only attribute in the LFB instance, see Section 4.7.5 for details.

Capability information at the FE level may describe the LFB classes the FE can instantiate; the number of instances of each can be created; the topological (i.e., linkage) limitations between these LFB instances, etc. Section 5 defines the FE level attributes including capability information.

Once the FE capability is described to the CE, the FE state information can be represented by two levels. The first level is the logically separable and distinctive packet processing functions, and we call these individual functions Logical Functional Blocks (LFBs). The second level of information is about how these individual LFBs are ordered and placed along the datapath to deliver a complete forwarding plane service. The interconnection and ordering of the LFBs is called LFB Topology. Section 3.2 discuss high level concepts around LFBs while Section 3.3 discuss issues around LFB topology.

3.2. LFB (Logical Functional Block) Modeling

Each LFB performs a well-defined action or computation on the packets passing through it. Upon completion of such a function, either the packets are modified in certain ways (e.g., decapsulator, marker), or some results are generated and stored, probably in the form of metadata (like a classifier). Each LFB typically does one thing and one thing only. Classifiers, shapers, meters are all examples of LFBs. Modeling LFBs at such a fine

granularity allows us to use a small number of LFBs to create the higher-order FE functions (such as an IPv4 forwarder) precisely, which in turn can describe more complex networking functions and vendor implementations of software and hardware. Section 6 provides a list of useful LFBs with such granularity.

An LFB has one or more inputs, each of which takes a packet P, and optionally metadata M; and produces one or more outputs, each of which carries a packet P', and optionally metadata M'. Metadata is data associated with the packet in the network processing device (router, switch, etc.) and passed from one LFB to the next, but not sent across the network. It is most likely that there are multiple LFBs within one FE, as shown in Figure 2, and all the LFBs share the same ForCES protocol termination point that implements the ForCES protocol logic and maintains the communication channel to and from the CE.

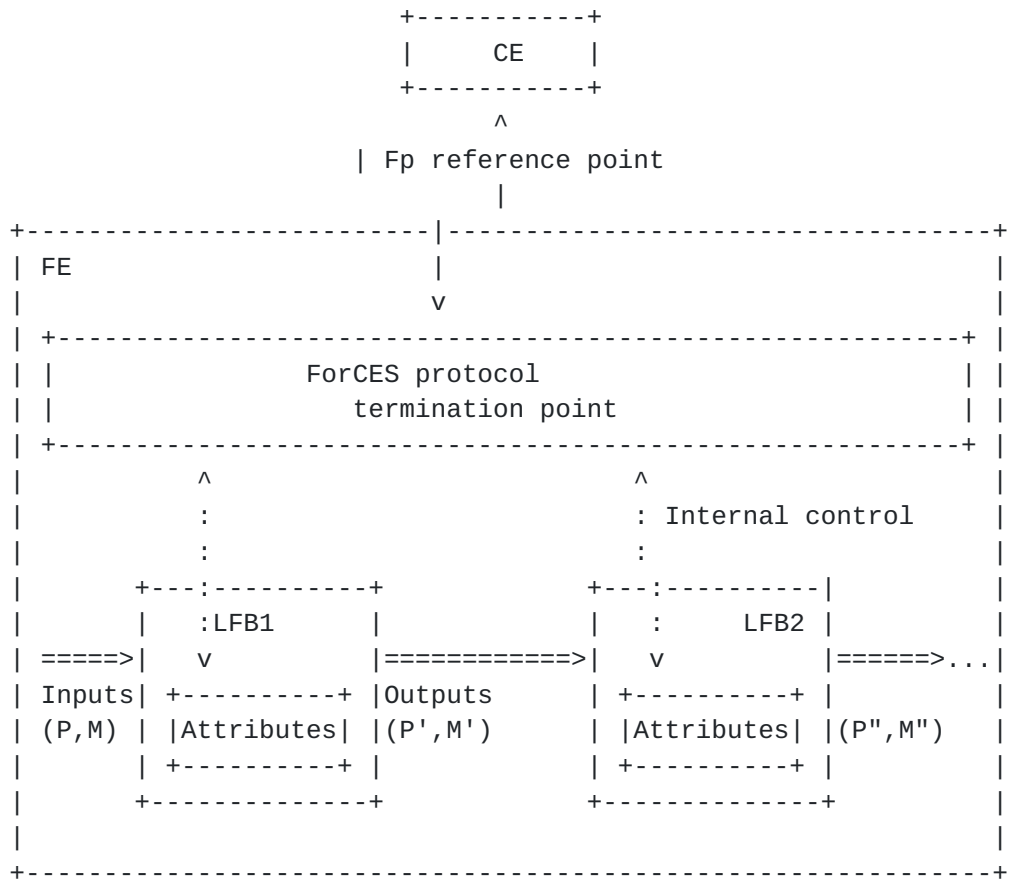


Figure 2. Generic LFB Diagram

An LFB, as shown in Figure 2, has inputs, outputs and attributes that can be queried and manipulated by the CE indirectly via Fp reference point (defined in [2]) and the ForCES protocol termination point. The horizontal axis is in the forwarding plane for connecting the inputs and outputs of LFBs within the same FE. The vertical axis between the CE and the FE denotes the Fp reference point where bidirectional communication between the CE and FE happens: the CE to FE communication is for configuration, control and packet injection while FE to CE communication is used for packet re-direction to the control plane, monitoring and accounting information, errors, etc. Note that the interaction between the CE and the LFB is only abstract and indirect. The result of such interaction is for the CE to indirectly manipulate the attributes of the LFB instances.

A namespace is used to associate a unique name or ID with each LFB class. The namespace must be extensible so that new LFB class can also be added later to accommodate future innovation in the forwarding plane.

LFB operation must be specified in the model to allow the CE to understand the behavior of the forwarding datapath. For instance, the CE must understand at what point in the datapath the IPv4 header TTL is decremented (i.e., it needs to know if a control packet could be delivered to the CE either before or after this point in the datapath). In addition, the CE must understand where and what type of header modifications (e.g., tunnel header append or strip) are performed by the FEs. Further, the CE must verify that various LFBs along a datapath within an FE are compatible to link together.

There is value to vendors if the operation of LFB classes can be expressed in sufficient detail so that physical devices implementing different LFB functions can be integrated easily into an FE design. Therefore, a semi-formal specification is needed; that is, a text description of the LFB operation (human readable), but sufficiently specific and unambiguous to allow conformance testing and efficient design (i.e., eliminate guess-work), so that interoperability between different CEs and FEs can be achieved.

The LFB class model specifies information like:

- number of inputs and outputs (and whether they are configurable)
- metadata read/consumed from inputs;
- metadata produced at the outputs;
- packet type(s) accepted at the inputs and emitted at the outputs;

- packet content modifications (including encapsulation or decapsulation);
- packet routing criteria (when multiple outputs on an LFB are present);
- packet timing modifications;
- packet flow ordering modifications;
- LFB capability information;
- LFB operational attributes, etc.

Section 4 of this document provides a detailed discussion of the LFB model with a formal specification of LFB class schema. The rest of Section 3.2 only intends to provide a conceptual overview of some important issues in LFB modeling, without covering all the specific details.

3.2.1. LFB Input and Input Group

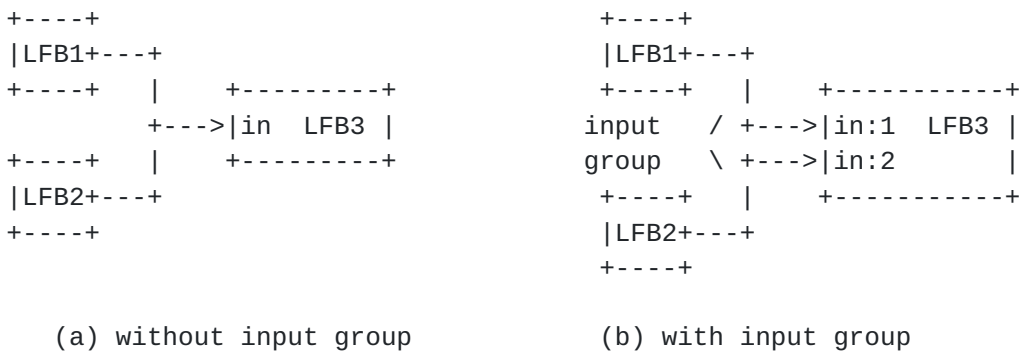
An LFB input is a conceptual port of the LFB where the LFB can receive information from other LFBs. The information is typically a packet (or frame in general) and associated metadata, although in some cases it might consist of only metadata, i.e., with a Null-packet.

It is inevitable that there will be LFB instances that will receive packets from more than one other LFB instances (fan-in). If these fan-in links all carry the same type of information (packet type and set of metadata) and require the same processing within the LFB, then one input should be sufficient. If, however, the LFB class can receive two or more very different types of input, and the processing of these inputs are also very distinct, then that may justify the definition of multiple inputs. But in these cases splitting the LFB class into two LFB classes should always be considered as an alternative. In intermediate cases, e.g., where the inputs are somewhat different but they require very similar processing, the shared input solution should be preferred. For example, if an Ethernet framer LFB is capable of receiving IPv4 and IPv6 packets, these can be served by the same LFB input.

Note that we assume the model allows for connecting more than one LFB output to a single LFB input directly. There is no restriction on the number of up-stream LFBs connecting their outputs to the same input of a single LFB instance. Note that the behavior of the system when multiple packets arrive at such an input simultaneously is not defined by the model. If such behavior needs to be described, it can be done either by separating the single input to become multiple inputs (one per output), or by inserting other

appropriate LFBs (such as Queues and possibly Schedulers) between the multiple outputs and the single input.

If there are multiple inputs with the same input type, we model them as an input group, that is, multiple instances of the same input type. In general, an input group is useful to allow an LFB to differentiate packet treatment based on where the packet came from.



(a) without input group (b) with input group

Figure 3. An example of using input group.

Consider the following two cases in Figure 3(a) and (b). In Figure 3(a), the output from two LFBs are directly connected into one input of LFB3, assuming that it can be guaranteed that no two packets arrive at the same time instance. If LFB3 must do something different based on the source of the packet (LFB1 or LFB2), the only way to model that is to make LFB1 and LFB2 pass some metadata with different values so that LFB3 can make the differentiation based on that metadata. In Figure 3(b), that differentiation can be elegantly expressed within LFB3 using the input group concept where the instance id can server as the differentiating key. For example, a scheduler LFB can potentially use an input group consisting of a variable number of inputs to differentiate the queues from which the packets are coming.

3.2.2. LFB Output and Output Group

An LFB output is a conceptual port of the LFB that can send information to some other LFBs. The information is typically a packet (or frame in general) and associated metadata, although in some cases it might emit only metadata, i.e., with a Null-packet.

We assume that a single LFB output can be connected to only one LFB input (this is required to make the packet flow through the LFB topology unambiguous). Therefore, to allow any non-trivial topology, multiple outputs must be allowed for an LFB class. If there are multiple outputs with the same output type, we model them

as output group, that is, multiple instances of the same output type. For illustration of output group, consider the hypothetical LFB in Figure 4. The LFB has two types of outputs, one of which can be instantiated to form an output group.

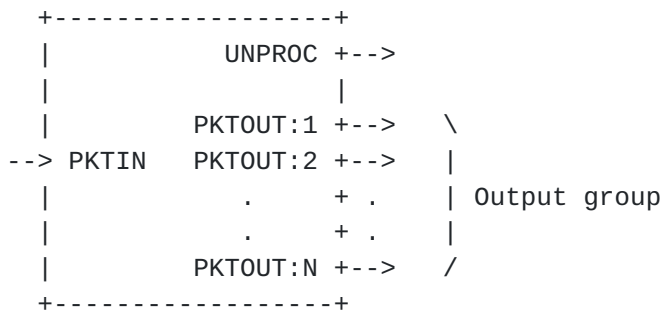


Figure 4. An example of an LFB with output group.

Multiple outputs should mainly be used for functional separation where the outputs are connected to very different types of LFBs. For example, an IPv4 LPM (Longest-Prefix-Matching) LFB may have one default output to send those packets for which look-up was successful (passing a META_ROUTEID as metadata); and have another output for sending packets for which the look-up failed. The former output may be connected to a route handler LFB, while the latter can be connected to an ICMP response generator LFB or to a packet handler LFB that passes the packet up to the CE.

3.2.3. Packet Type

When LFB classes are defined, the input and output packet formats (e.g., IPv4, IPv6, Ethernet, etc.) must be specified: these are the types of packets a given LFB input is capable of receiving and processing, or a given LFB output is capable of producing. This requires that distinct frame types be uniquely labeled with a symbolic name and/or ID.

Note that each LFB has a set of packet types that it operates on, but it does not care about whether the underlying implementation is passing a greater portion of the packets. For example, an IPv4 LFB might only operate on IPv4 packets, but the underlying implementation may or may not be stripping the L2 header before handing it over -- whether that is happening or not is opaque to the CE.

3.2.4. Metadata

Metadata is the per-packet state that is passed from one LFB to another. The metadata is passed with the packet to assist with further processing of that packet. The ForCES model must capture how the per-packet state information is propagated from one LFB to other LFBs. Practically, such metadata propagation can happen within one FE, or cross the FE boundary between two interconnected FEs. We believe that the same metadata model can be used for both situations, however, our focus here is for intra-FE metadata.

Each metadata can be conveniently modeled as a <label, value> pair, where the label identifies the type of information, (e.g., "color"), and its value holds the actual information (e.g., "red"). The tag here is shown as a textual label, but it can be replaced or associated with a unique numeric value (identifier).
The metadata life-cycle is defined in this model using three types of events: "write", "read" and "consume". The first "write" initializes the value of the metadata (implicitly creating and/or initializing the metadata), and hence starts the life-cycle. The explicit "consume" event terminates the life-cycle. Within the life-cycle, that is, after a "write" event, but before the next "consume" event, there can be an arbitrary number of "write" and "read" events. These "read" and "write" events can be mixed in an arbitrary order within the life-cycle. Outside of the life-cycle of the metadata, that is, before the first "write" event, or between a "consume" event and the next "write" event, the metadata should be regarded non-existent or non-initialized. Thus, reading a metadata outside of its life-cycle is considered an error.

To ensure inter-operability between LFBs, the LFB class specification must define what metadata the LFB class "reads" or "consumes" on its input(s) and what metadata it "produces" on its output(s). For maximum extensibility, this definition should not specify which LFBs the metadata is expected to come from for a consumer LFB, or which LFBs are expected to consume metadata for a producer LFB.

While it is important to define the metadata types passing between LFBs, it is not necessary to define the exact encoding mechanism used by LFBs for that metadata. Different implementations are allowed to use different encoding mechanisms for metadata. For example, one implementation may store metadata in registers or shared memory, while another implementation may encode metadata in-band as a preamble in the packets.

At any link between two LFBs, the packet is marked with a finite set of active metadata, where active means the metadata is within its life-cycle. (i.e., the metadata has been properly initialized

and has not been consumed yet.) There are two corollaries of this model:

1. No uninitialized metadata exists in the model.

2. No more than one occurrence of each metadata tag can be associated with a packet at any given time.

3.2.4.1. LFB Operations on Metadata

When the packet is processed by an LFB (i.e., between the time it is received and forwarded by the LFB), the LFB may perform read, write and/or consume operations on any active metadata associated with the packet. If the LFB is considered to be a black box, one of the following operations is performed on each active metadata.

- IGNORE: ignores and forwards the metadata
- READ: reads and forwards the metadata
- READ/RE-WRITE: reads, over-writes and forwards the metadata
- WRITE: writes and forwards the metadata (can also be used to create new metadata)
- READ-AND-CONSUME: reads and consumes the metadata
- CONSUME consumes metadata without reading

The last two operations terminate the life-cycle of the metadata, meaning that the metadata is not forwarded with the packet when the packet is sent to the next LFB.

In our model, a new metadata is generated by an LFB when the LFB applies a WRITE operation into a metadata type that was not present when the packet was received by the LFB. Such implicit creation may be unintentional by the LFB, that is, the LFB may apply the WRITE operation without knowing or caring if the given metadata existed or not. If it existed, the metadata gets over-written; if it did not exist, the metadata gets created.

For source-type LFBs (i.e., an LFB that inserts packets into the model), WRITE is the only meaningful metadata operation.

Sink-type LFBs (i.e., an LFB that removes the packet from the model), may either READ-AND-CONSUME (read) or CONSUME (ignore) each active metadata associated with the packet.

3.2.4.2. Metadata Production and Consumption

For a given metadata on a given packet path, there must be at least one producer LFB that creates that metadata and should be at least one consumer LFB that needs the metadata. In this model, the producer and consumer LFBs of a metadata are not required to be adjacent. There may be multiple consumers for the same metadata and there may be multiple producers of the same metadata. When a packet path involves multiple producers of the same metadata, then the second, third, etc. producers overwrite that metadata value.

The metadata that is produced by an LFB is specified by the LFB class definition on a per output port group basis. A producer may always generate the metadata on the port group, or may generate it only under certain conditions. We call the former an "unconditional" metadata, whereas the latter is a "conditional" metadata. In the case of conditional metadata, it should be possible to determine from the definition of the LFB when a "conditional" metadata is produced.

The consumer behavior of an LFB, that is, the metadata that the LFB needs for its operation, is defined in the LFB class definition on a per input port group basis. An input port group may "require" a given metadata, or may treat it as "optional" information. In the latter case, the LFB class definition must explicitly define what happens if an optional metadata is not provided. One approach is to specify a default value for each optional metadata, and assume that the default value is used if the metadata is not provided with the packet.

When a consumer requires a given metadata, it has dependencies on its up-stream LFBs. That is, the consumer LFB can only function if there is at least one producer of that metadata and no intermediate LFB consumes the metadata.

The model should expose this inter-dependency. Furthermore, it should be possible to take this inter-dependency into consideration when constructing LFB topologies, and also that the dependency can be verified when validating topologies.

For extensibility reasons, the LFB specification should define what metadata the LFB requires without specifying which LFB(s) it expect a certain metadata to come from. Similarly, LFBs should specify what metadata they produce without specifying which LFBs the metadata is meant for.

When specifying the metadata tags, some harmonization effort must be made so that the producer LFB class uses the same tag as its intended consumer(s), or vice versa.

3.2.4.3. Fixed, Variable and Configurable Tag

When the produced metadata is defined for a given LFB class, most metadata will be specified with a fixed tag. For example, a Rate Meter LFB will always produce the "Color" metadata.

A small subset of LFBs need to have the capability to produce one or more of their metadata with tags that are not fixed in the LFB class definition, but instead can be selected per LFB instance. An example of such an LFB class is a Generic Classifier LFB. We call this variable tag metadata production. If an LFB produces metadata with variable tag, a corresponding LFB attribute--called the tag selector--specifies the tag for each such metadata. This mechanism is to improve the versatility of certain multi-purpose LFB classes, since it allows the same LFB class be used in different topologies, producing the right metadata tags according to the needs of the topology.

Depending on the capability of the FE, the tag selector can be a read-only or a read-write attribute. In the former case, the tag cannot be modified by the CE. In the latter case the tag can be configured by the CE, hence we call this "configurable tag metadata production." (Note that in this definition configurable tag metadata production is a subset of variable tag metadata production.)

Similar concepts can be introduced for the consumer LFBs to satisfy the different metadata needs. Most LFB classes will specify their metadata needs using fixed metadata tags. For example, a Next Hop LFB may always require a "NextHopId" metadata; but the Redirector LFB may need to use a "ClassID" metadata in one instance, and a "ProtocolType" metadata in another instance as a basis for selecting the right output port. In this case, an LFB attribute is used to provide the required metadata tag at run-time. This metadata tag selector attribute may be read-only or read-write, depending on the capabilities of the LFB instance and the FE.

3.2.4.4. Metadata Usage Categories

Depending on the role and usage of a metadata, various amount of encoding information must be provided when the metadata is defined,

and some cases offer less flexibility in the value selection than others.

As far as usage of a metadata is concerned, three types of metadata exist:
- Relational (or binding) metadata
- Enumerated metadata
- Explicit/external value metadata

The purpose of the relational metadata is to refer in one LFB instance (producer LFB) to a "thing" in another downstream LFB instance (consumer LFB), where the "thing" is typically an entry in a table attribute of the consumer LFB.

For example, the Prefix Lookup LFB executes an LPM search using its prefix table and resolves to a next-hop reference. This reference needs to be passed as metadata by the Prefix Lookup LFB (producer) to the Next Hop LFB (consumer), and must refer to a specific entry in the next-hop table within the consumer.

Expressing and propagating such binding relationship is probably the most common usage of metadata. One or more objects in the producer LFB are related (bound) to a specific object in the consumer LFB. Such a relation is established by the CE very explicitly, i.e., by properly configuring the attributes in both LFBs. Available methods include the following:

The binding may be expressed by tagging the involved objects in both LFBs with the same unique (but otherwise arbitrary) identifier. The value of the tag is explicitly configured (written by the CE) into both LFBs, and this value is also the value that the metadata carries between the LFBs.

Another way of setting up binding relations is to use a naturally occurring unique identifier of the consumer's object (for example, the array index of a table entry) as a reference (and as a value of the metadata. In this case, the index is obtained (read) or inferred by the CE by communicating with the consumer LFB. Once the CE obtains the index, it needs to plug (write) it into the producer LFB to establish the binding.

Important characteristics of the binding usage of metadata are:

- The value of the metadata shows up in the CE-FE communication for BOTH the consumer and the producer. That is, the metadata value must be carried over the ForCES protocol. Using the tagging technique, the value is WRITTEN to both LFBs. Using the other

technique, the value is WRITTEN to only the producer LFB and may be READ from the consumer LFB.

- The actual value is irrelevant for the CE, the binding is simply expressed by using the SAME value at the consumer and producer LFBs.

- Hence the definition of the metadata does not have to include value assignments. The only exception is when some special value(s) of the metadata must be reserved to convey special events. Even though these special cases must be defined with the metadata specification, their encoded values can be selected arbitrarily. For example, for the Prefix Lookup LFB example, a special value may be reserved to signal the NO-MATCH case, and the value of zero may be assigned for this purpose.

The second class of metadata is the enumerated type. An example is the "Color" metadata that is produced by a Meter LFB and consumed by some other LFBs. As the name suggests, enumerated metadata has a relatively small number of possible values, each with a very specific meaning. All of the possible cases must be enumerated when defining this class of metadata. Although a value encoding must be included in the specification, the actual values can be selected arbitrarily (e.g., <Red=0, Yellow=1, Green=2> and <Red=3, Yellow=2, Green 1> would be both valid encodings, what is important is that an encoding is specified).

The value of the enumerated metadata may or may not be conveyed via the ForCES protocol between the CE and FE.

The third class of metadata is the explicit type. This refers to cases where the value of the metadata is explicitly used by the consumer LFB to change some packet header fields. In other words, its value has a direct and explicit impact on some field and will be visible externally when the packet leaves the NE. Examples are: TTL increment given to a Header Modifier LFB, and DSCP value for a Remarker LFB. For explicit metadata, the value encoding must be explicitly provided in the metadata definition, where the values cannot be selected arbitrarily, but rather they should conform to what is commonly expected. For example, a TTL increment metadata should encode with zero for the no increment case, by one for the single increment case, etc. A DSCP metadata should use 0 to encode DSCP=0, 1 to encode DSCP=1, etc.

3.2.5. LFB Versioning

LFB class versioning is a method to enable incremental evolution of LFB classes. Unlike inheritance (discussed next in Section 3.2.6), where it assumed that an FE datapath model containing an LFB instance of a particular class C could also simultaneously contain an LFB instance of a class C' inherited from class C; with versioning, an FE would not be allowed to contain an LFB instance for more than one version of a particular class.

LFB class versioning is supported by requiring a version string in the class definition. CEs may support backwards compatibility between multiple versions of a particular LFB class, but FEs are not allowed to support more than one single version of a particular class.

3.2.6. LFB Inheritance

LFB class inheritance is supported in the FE model as a means of defining new LFB classes. This also allows FE vendors to add vendor-specific extensions to standardized LFBs. An LFB class specification MUST specify the base class (with version number) it inherits from (with the default being the base LFB class). Multiple-inheritance is not allowed, though, to avoid the unnecessary complexity.

Inheritance should be used only when there is significant reuse of the base LFB class definition. A separate LFB class should be defined if there is not enough reuse between the derived and the base LFB class.

An interesting issue related to class inheritance is backward compatibility (between a descendant and an ancestor class). Consider the following hypothetical scenario where there exists a standardized LFB class "L1". Vendor A builds an FE that implements LFB "L1" and vendors B builds a CE that can recognize and operate on LFB "L1". Suppose that a new LFB class, "L2", is defined based on the existing "L1" class (for example, by extending its capabilities in some incremental way). Lets first examine the FE backward compatibility issue by considering what would happen if vendor B upgrades its FE from "L1" to "L2" while vendor C's CE is not changed. The old L1-based CE can interoperate with the new L2-based FE if the derived LFB class "L2" is indeed backward compatible with the base class "L1".

The reverse scenario is a much less problematic case, i.e., when CE vendor B upgrades to the new LFB class "L2", but the FE is not upgraded. Note that as long as the CE is capable of working with older LFB classes, this problem does not affect the model; hence we

will use the term "backward compatibility" to refer to the first scenario concerning FE backward compatibility.

Inheritance can be designed into the model with backward compatibility support by constraining the LFB inheritance such that the derived class is always a functional superset of the base class, i.e., the derived class can only grow on top of the base class, but not shrink from it. Additionally, the following mechanisms are required to support FE backward compatibility:

1) When detecting an LFB instance of an LFB type that is unknown to the CE, the CE MUST be able to query the base class of such an LFB from the FE.
2) The LFB instance on the FE SHOULD support a backward compatibility mode (meaning the LFB instance reverts itself back to the base class instance), and the CE SHOULD be able to configure the LFB to run in such mode.

3.3. FE Datapath Modeling

Packets coming into the FE from ingress ports generally flow through multiple LFBs before leaving out of the egress ports. How an FE treats a packet depends on many factors, such as type of the packet (e.g., IPv4, IPv6 or MPLS), actual header values, time of arrival, etc. The result of the operation of an LFB may have an impact on how the packet is to be treated in further (downstream) LFBs and this differentiation of packet treatment downstream can be conceptualized as having alternative datapaths in the FE. For example, the result of a 6-tuple classification (performed by a classifier LFB) controls what rate meter is applied to the packet (by a rate meter LFB) in a later stage in the datapath.

LFB topology is a directed graph representation of the logical datapaths within an FE, with the nodes representing the LFB instances and the directed link the packet flow direction from one LFB to the next. Section 3.3.1 discusses how the FE datapaths can be modeled as LFB topology; while Section 3.3.2 focuses on issues around LFB topology reconfiguration.

3.3.1. Alternative Approaches for Modeling FE Datapaths

There are two basic ways to express the differentiation in packet treatment within an FE, one representing the datapath directly and graphically (topological approach) and the other utilizing metadata (the encoded state approach).

. Topological Approach

Using this approach, differential packet treatment is expressed via actually splitting the LFB topology into alternative paths. In other words, if the result of an LFB must control how the packet is further processed, then such an LFB will have separate output ports (one for each alternative treatment) connected to separate sub-graphs (each expressing the respective treatment downstream).

. Encoded State Approach

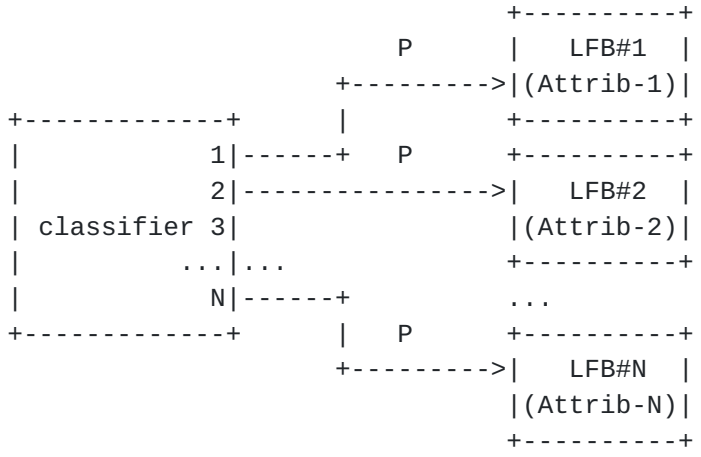
An alternative way of expressing differential treatment is using metadata. The result of the operation of an LFB can be encoded in a metadata which is passed along with the packet to downstream LFBs. A downstream LFB, in turn, can use the metadata (and its value, e.g., as an index into some table) to decide how to treat the packet.

Theoretically, the two approaches can substitute for each other, so one may consider using purely one (or the other) approach to describe all datapaths in an FE. However, neither model by itself is very useful for practically relevant cases. For a given FE with certain logical datapaths, applying the two different modeling approaches would result in very different looking LFB topology graphs. A model using purely the topological approach may require a very large graph with many links (i.e., paths) and nodes (i.e., LFB instances) to express all alternative datapaths. On the other hand, a model using purely the encoded state model would be restricted to a string of LFBs, which would make it very unintuitive to describe very different datapaths (such as MPLS and IPv4). Therefore, a mix of these two approaches will likely be used for a practical model. In fact, as we illustrate it below, the two approaches can be mixed even within the same LFB.

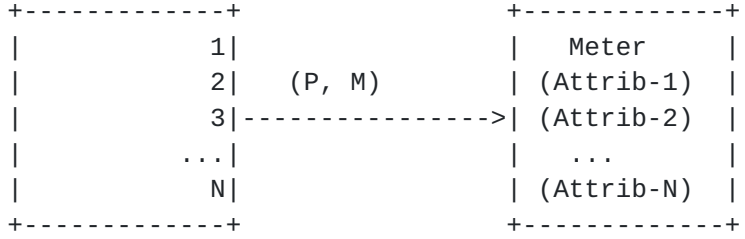
Using a simple example of a classifier with N classification outputs followed by some other LFBs, Figure 5(a) shows what the LFB topology looks like by using the purely topological approach. Each output from the classifier goes to one of the N LFBs followed and no metadata is needed here. The topological approach is simple, straightforward and graphically intuitive. However, if N is large and the N nodes followed the classifier (LFB#1, LFB#2, ..., LFB#N) all belong to the same LFB type (for example, meter) but each with its own independent attributes, the encoded state approach gives a much simpler topology representation, as shown in Figure 5(b). The encoded state approach requires that a table of N rows of meter attributes is provided in the Meter node itself, with each row representing the attributes for one meter instance. A metadata M is also needed to pass along with the packet P from the classifier

to the meter, so that the meter can use M as a look-up key (index) to find the corresponding row of the attributes that should be used for any particular packet P.

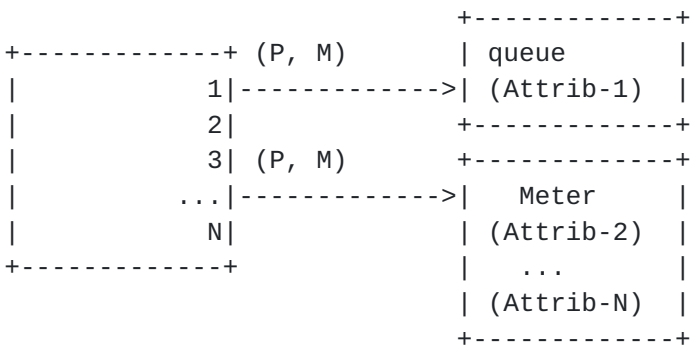
Now what if all the N nodes (LFB#1, LFB#2, ..., LFB#N) are not of the same type? For example, if LFB#1 is a queue while the rest are all meters, what is the best way to represent such datapaths? While it is still possible to use either the pure topological approach or the pure encoded state approach, the natural combination of the two seems the best by representing the two different functional datapaths using topological approach while leaving the N-1 meter instances distinguished by metadata only, as shown in Figure 5(c).



5(a) Using pure topological approach



5(b) Using pure encoded state approach to represent the LFB topology in 5(a), if LFB#1, LFB#2, ..., and LFB#N are of the same type (e.g., meter).

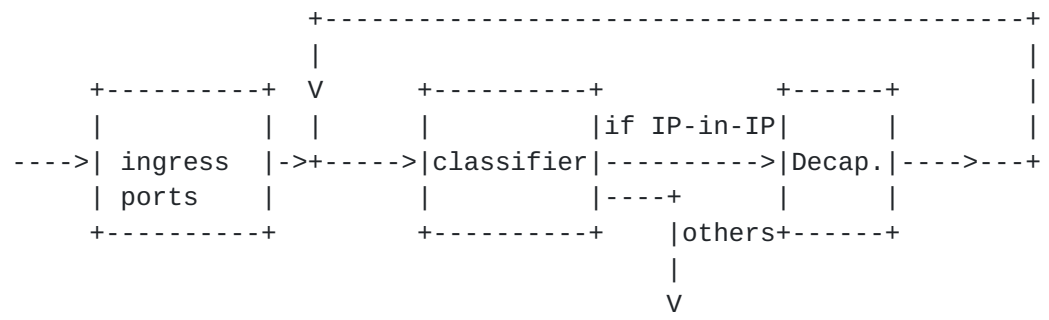


5(c) Using a combination of the two, if LFB#1, LFB#2, ..., and LFB#N are of different types (e.g., queue and meter).

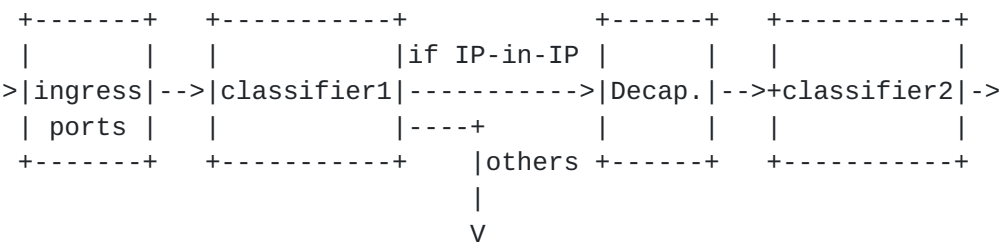
Figure 5. An example of how to model FE datapaths

From this example, we demonstrate that each approach has distinct advantage for different situations. Using the encoded state approach, fewer connections are typically needed between a fan-out node and its next LFB instances of the same type, because each packet carries metadata with it so that the following nodes can interpret and hence invoke a different packet treatment. For those cases, a pure topological approach forces one to build elaborate graphs with a lot more connections and often results in an unwieldy graph. On the other hand, a topological approach is intuitive and most useful for representing functionally very different datapaths.

For complex topologies, a combination of the two is the most useful and flexible. Here we provide a general design guideline as to what approach is best used for what situation. The topological approach should primarily be used when the packet datapath forks into areas with distinct LFB classes (not just distinct parameterizations of the same LFB classes), and when the fan-outs do not require changes (adding/removing LFB outputs) at all or require only very infrequent changes. Configuration information that needs to change frequently should preferably be expressed by the internal attributes of one or more LFBs (and hence using the encoded state approach).



(a) The LFB topology with a logical loop



(b) The LFB topology without the loop utilizing two independent classifier instances.

Figure 6. An LFB topology example.

It is important to point out that the LFB topology here is the logical topology that the packets flow through, not the physical topology as determined by how the FE hardware is laid out. Nevertheless, the actual implementation may still influence how the functionality should be mapped into the LFB topology. Figure 6 shows one simple FE example. In this example, an IP-in-IP packet from an IPSec application like VPN may go to the classifier first and have the classification done based on the outer IP header; upon being classified as an IP-in-IP packet, the packet is then sent to a decapsulator to strip off the outer IP header, followed by a classifier again to perform classification on the inner IP header. If the same classifier hardware or software is used for both outer and inner IP header classification with the same set of filtering rules, a logical loop is naturally present in the LFB topology, as shown in Figure 6(a). However, if the classification is implemented by two different pieces of hardware or software with different filters (i.e., one set of filters for outer IP header while another set for inner IP header), then it is more natural to model them as two different instances of classifier LFB, as shown in Figure 6(b).

To distinguish multiple instances of the same LFB class, each LFB instance has its own LFB instance ID. One way to encode the LFB

instance ID is to encode it as x.y where x is the LFB class ID while y is the instance ID within each LFB class.

3.3.2. Configuring the LFB Topology

While there is little doubt that the individual LFB must be configurable, the configurability question is more complicated for LFB topology. Since LFB topology is really the graphic representation of the datapaths within FE, configuring the LFB topology means dynamically changing the datapaths including changes to the LFBs along the datapaths on an FE, e.g., creating (i.e., instantiating) or deleting LFBs, setting up or deleting interconnections between outputs of upstream LFBs to inputs of downstream LFBs.

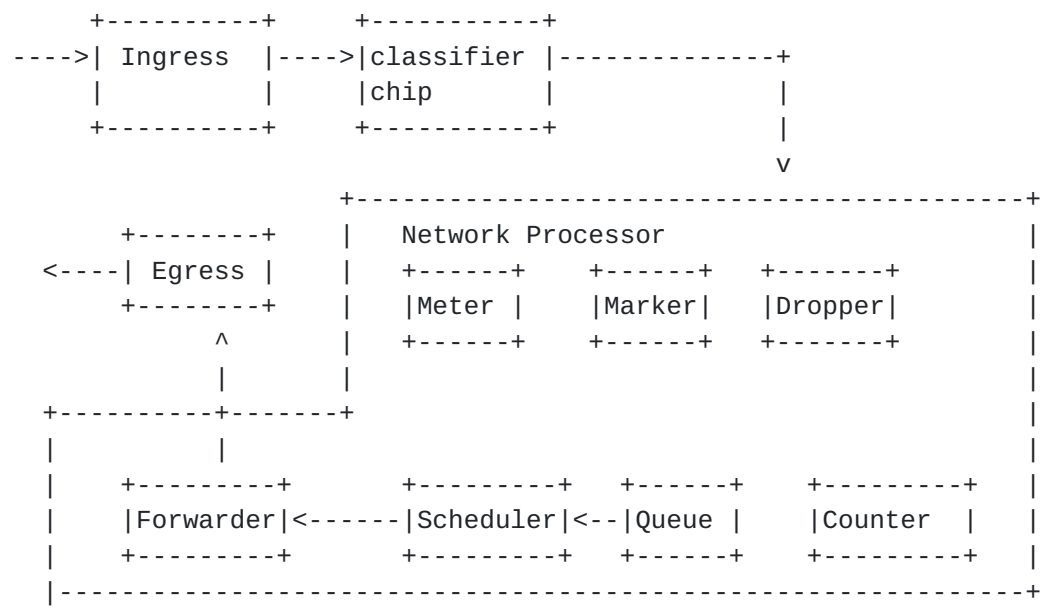
Why would the datapaths on an FE ever change dynamically? The datapaths on an FE is set up by the CE to provide certain data plane services (e.g., DiffServ, VPN, etc.) to the NE's customers. The purpose of reconfiguring the datapaths is to enable the CE to customize the services the NE is delivering at run time. The CE needs to change the datapaths when the service requirements change, e.g., when adding a new customer, or when an existing customer changes their service. However, note that not all datapath changes result in changes in the LFB topology graph, and that is determined by the approach we use to map the datapaths into LFB topology. As discussed in 3.3.1, the topological approach and encoded state approach can result in very different looking LFB topologies for the same datapaths. In general, an LFB topology based on a pure topological approach is likely to experience more frequent topology reconfiguration than one based on an encoded state approach. However, even an LFB topology based entirely on an encoded state approach may have to change the topology at times, for example, to totally bypass some LFBs or insert new LFBs. Since a mix of these two approaches is used to model the datapaths, LFB topology reconfiguration is considered an important aspect of the FE model.

We want to point out that allowing a configurable LFB topology in the FE model does not mandate that all FEs must have such capability. Even if an FE supports configurable LFB topology, it is expected that there will be FE-specific limitations on what can actually be configured. Performance-optimized hardware implementation may have zero or very limited configurability, while FE implementations running on network processors may provide more flexibility and configurability. It is entirely up to the FE designers to decide whether or not the FE actually implements such reconfiguration and how much. Whether it is a simple runtime switch to enable or disable (i.e., bypass) certain LFBs, or more

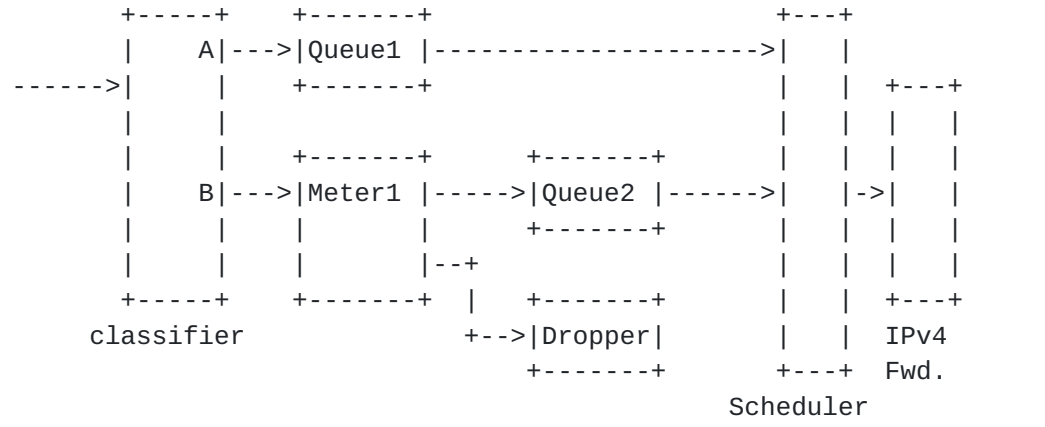
flexible software reconfiguration is all implementation detail internal to the FE and outside of the scope of FE model. In either case, the CE(s) must be able to learn the FE's configuration capabilities. Therefore, the FE model must provide a mechanism for describing the LFB topology configuration capabilities of an FE. These capabilities may include (see Section 5 for full details):

- What LFB classes can the FE instantiate?
- How many instances of the same LFB class can be created?
- What are the topological limitations? For example:
  - How many instances of the same class or any class can be created on any given branch of the graph?
  - Ordering restrictions on LFBs (e.g., any instance of LFB class A must be always downstream of any instance of LFB class B).

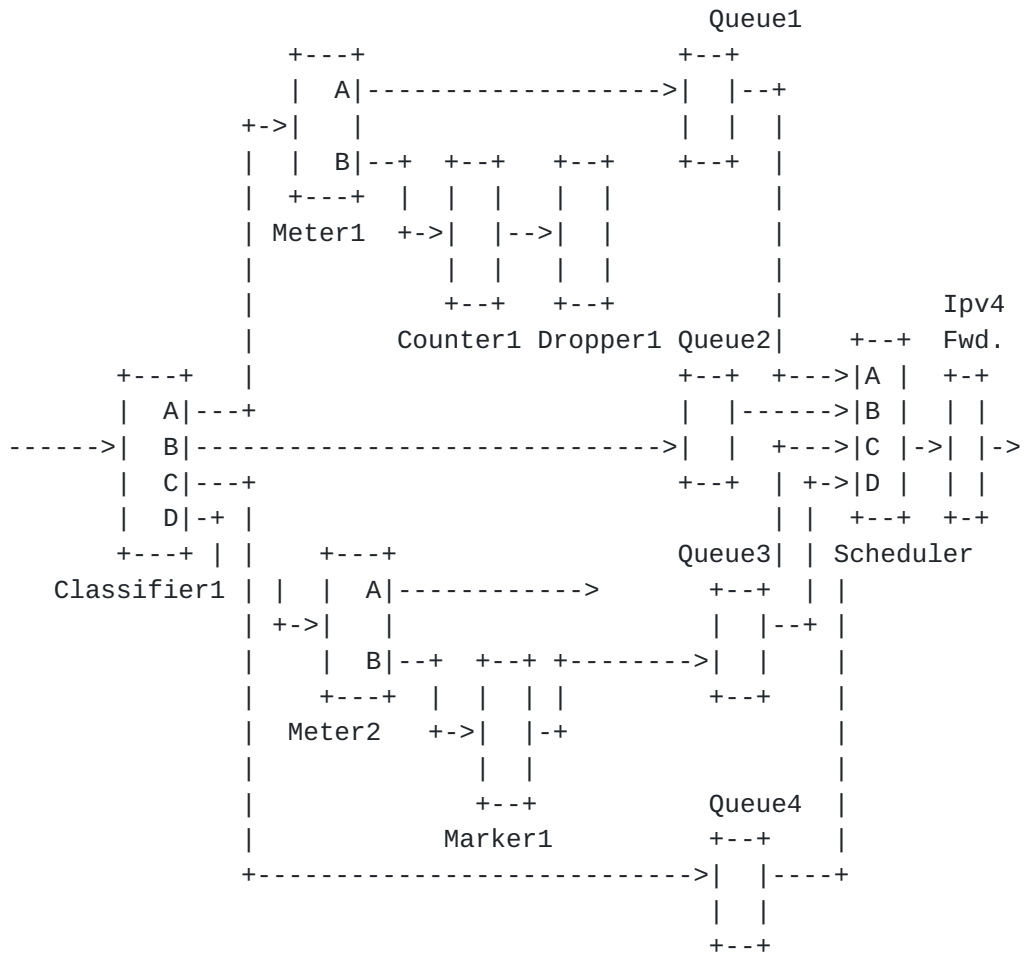
Even if the CE is allowed to configure LFB topology for an FE, how can the CE interpret an arbitrary LFB topology (presented to the CE by the FE) and know what to do with it? In other words, how does the CE know the mapping between an LFB topology and a particular NE service or application (e.g., VPN, DiffServ, etc.)? We argue that first of all, it is unlikely that an FE can support any arbitrary LFB topology; secondly, once the CE understands the coarse capability of an FE, it is up to the CE to configure the LFB topology according to the network service the NE is supposed to provide. So the more important mapping that the CE has to understand is from the high level NE service to a specific LFB topology, not the other way around. Do we expect the CE has the ultimate intelligence to translate any high level service policy into the configuration data for the FEs? No, but it is conceivable that within a given network service domain (like DiffServ), a certain amount of intelligence can be programmed into the CE such that the CE has a general understanding of the LFBs involved and so the translation from a high level service policy to the low level FE configuration can be done automatically. In any event, this is considered an implementation issue internal to the control plane and outside the scope of the FE model. Therefore, it is not discussed any further in this draft.



(a) The Capability of the FE, reported to the CE



(b) One LFB topology as configured by the CE and accepted by the FE



(c) Another LFB topology as configured by the CE and accepted by the FE

Figure 7. An example of configuring LFB topology.

Figure 7 shows an example where a QoS-enabled router has several line cards that have a few ingress ports and egress ports, a specialized classification chip, a network processor containing codes for FE blocks like meter, marker, dropper, counter, queue, scheduler and Ipv4 forwarder. Some of the LFB topology is already fixed and has to remain static due to the physical layout of the line cards. For example, all the ingress ports might be already hard wired into the classification chip and so all packets must follow from the ingress port into the classification engine. On the other hand, the LFBs on the network processor and their execution order are programmable, even though there might exist certain capacity limits and linkage constraints between these LFBs. Examples of the capacity limits might be: there can be no more than 8 meters; there can be no more than 16 queues in one FE; the scheduler can handle at most up to 16 queues; etc. The linkage

constraints might dictate that classification engine may be followed by a meter, marker, dropper, counter, queue or IPv4 forwarder, but not scheduler; queues can only be followed by a scheduler; a scheduler must be followed by the IPv4 forwarder; the last LFB in the datapath before going into the egress ports must be the IPv4 forwarder, etc.

Once the FE reports such capability and capacity to the CE, it is now up to the CE to translate the QoS policy into the desirable configuration for the FE. Figure 7(a) depicts the FE capability while 7(b) and 7(c) depict two different topologies that the FE might be asked to configure to. Note that both the ingress and egress are omitted in (b) and (c) for simple representation. The topology in 7(c) is considerably more complex than 7(b) but both are feasible within the FE capabilities, and so the FE should accept either configuration request from the CE.

4. Model and Schema for LFB Classes

The main goal of the FE model is to provide an abstract, generic, modular, implementation-independent representation of the FEs. This is facilitated using the concept of LFBs which are instantiated from LFB classes. LFB classes and associated definitions will be provided in a collection of XML documents. The collection of these XML documents is called a LFB class library, and each document is called an LFB class library document (or library document, for short). Each of the library documents will conform to the schema presented in this section. The root element of the library document is the <LFBLibrary> element.

It is not expected that library documents will be exchanged between FEs and CEs "over-the-wire". But the model will serve as an important reference for the design and development of the CEs (software) and FEs (mostly the software part). It will also serve as a design input when specifying the ForCES protocol elements for CE-FE communication.

4.1. Namespace

The LFBLibrary element and all of its sub-elements are defined in the following namespace:

http://ietf.org/forces/1.0/lfbmodel

4.2. <LFBLibrary> Element

The <LFBLibrary> element serves as a root element of all library documents. It contains one or more of the following main blocks:

. <frameTypeDefs> for the frame declarations;
. <dataTypeDefs> for defining common data types;
. <metadataDefs> for defining metadata, and
. <LFBClassDefs> for defining LFB classes.

Each block is optional, that is, one library may contain only metadata defintions, another may contain only LFB class definitions, yet another may contain all of the above.

In addition to the above main blocks, a library document can import other library documents if it needs to refer to definitions contained in the included document. This concept is similar to the "#include" directive in C. Importing is expressed by the <load> elements, which must precede all the above elements in the document. For unique referencing, each LFBLibrary instance document has a unique label defined in the "provide" attribute of the LFBLibrary element.

The <LFBLibrary> element also includes an optional <description> element, which can be used to provide textual description about the library.

Following is a skeleton of a library document:

```
<?xml version="1.0" encoding="UTF-8"?>
<LFBLibrary xmlns="http://ietf.org/forces/1.0/lfbmodel"
  provides="this_library">

  <description>
    ...
  </description>

  <!-- Loading external libraries (optional) -->
  <load library="another_library"/>
  ...

  <!-- FRAME TYPE DEFINITIONS (optional) -->
  <frameTypeDefs>
    ...
  </frameTypeDefs>

  <!-- DATA TYPE DEFINITIONS (optional) -->
  <dataTypeDefs>
    ...
```

```
    </dataTypeDefs>

    <!-- METADATA DEFINITIONS (optional) -->
    <metadataDefs>
      ...
    </metadataDefs>

    <! LFB CLASS DEFINITIONS (optional) -->
    <LFBCLassDefs>
      ...
    </LFBCLassDefs>
  </LFBLibrary>
```

4.3. <load> Element

This element is used to refer to another LFB library document. Similar to the "include" directive in C, this makes the objects (metadata types, data types, etc.) defined in the referred library available for referencing in the current document.

The load element must contain the label of the library to be included and may contain a URL to specify where the library can be retrieved. The load element can be repeated unlimited times. Three examples for the <load> elements:

```
<load library="a_library"/>
<load library="another_library" location="another_lib.xml"/>
<load library="yetanother_library"
  location="http://www.petrimeat.com/forces/1.0/lfbmodel/lpm.xml"/>
```

4.4. <frameDefs> Element for Frame Type Declarations

Frame names are used in the LFB definition to define what types of frames the LFB expects at its input port(s) and emits at its output port(s). The <frameDefs> optional element in the library document contains one or more <frameDef> elements, each declaring one frame type.

Each frame definition contains a unique name (NMTOKEN) and a brief synopsis. In addition, an optional detailed description may be provided.

Uniqueness of frame types must be ensured among frame types defined in the same library document and in all directly or indirectly included library documents.

The following example defines two frame types:

```
<frameDefs>
  <frameDef>
    <name>ipv4</name>
    <synopsis>IPv4 packet</synopsis>
    <description>
      This frame type refers to an IPv4 packet.
    </description>
  </frameDef>
    <frameDef>
    <name>ipv6</name>
    <synopsis>IPv6 packet</synopsis>
    <description>
      This frame type refers to an IPv6 packet.
    </description>
  </frameDef>
  ...
</frameDefs>
```

4.5. <dataTypeDefs> Element for Data Type Definitions

The (optional) <dataTypeDefs> element can be used to define commonly used data types. It contains one or more <dataTypeDef> elements, each defining a data type with a unique name. Such data types can be used in several places in the library documents, including:

- Defining other data types
- Defining metadata
- Defining attributes of LFB classes

This is similar to the concept of having a common header file for shared data types.

Each <dataTypeDef> element contains a unique name (NMTOKEN), a brief
synopsis, an optional longer description, and a type definition

element. The name must be unique among all data types defined in the same library document and in any directly or indirectly included library documents. For example:

```
<dataTypeDefs>
  <dataTypeDef>
    <name>ieeemacaddr</name>
    <synopsis>48-bit IEEE MAC address</synopsis>
    ... type definition ...
  </dataTypeDef>
  <dataTypeDef>
    <name>ipv4addr</name>
    <synopsis>IPv4 address</synopsis>
    ... type definition ...
  </dataTypeDef>
  ...
</dataTypeDefs>
```

There are two kinds of data types: atomic and compound. Atomic data types are appropriate for single-value variables (e.g. integer, ASCII string, byte array).

The following built-in atomic data types are provided, but additional atomic data types can be defined with the <typeRef> and <atomic> elements:

<name>	Meaning
char	8-bit signed integer
uchar	8-bit unsigned integer
int16	16-bit signed integer
uint16	16-bit unsigned integer
int32	32-bit signed integer
uint32	32-bit unsigned integer
int64	64-bit signed integer
uint64	64-bit unisgned integer
string[N]	ASCII null-terminated string with buffer of N characters (string max length is N-1)
byte[N]	A byte array of N bytes
float16	16-bit floating point number
float32	32-bit IEEE floating point number
float64	64-bit IEEE floating point number

These built-in data types can be readily used to define metadata or LFB attributes, but can also be used as building blocks when defining new data types.

Compound data types can build on atomic data types and other compound data types. There are four ways that compound data types can be defined. They may be defined as an array of elements of some compound or atomic data type. They may be a structure of named elements of compound or atomic data types (ala C structures). They may be a union of named elements of compound or atomic data types (ala C unions). They may also be defined as augmentations (explained below in 4.5.6) of existing compound data types.

Given that the FORCES protocol will be getting and setting attribute values, all atomic data types used here must be able to be conveyed in the FORCES protocol. Further, the FORCES protocol will need a mechanism to convey compound data types. However, the details of such representations are for the protocol document, not the model documents.

For the definition of the actual type in the <dataTypeDef> element, the following elements are available: <typeRef>, <atomic>, <array>, <struct>, and <union>.

[EDITOR: How to support augmentation is for further study.]

4.5.1. <typeRef> Element for Aliasing Existing Data Types

The <typeRef> element refers to an existing data type by its name. The referred data type must be defined either in the same library document, or in one of the included library documents. If the referred data type is an atomic data type, the newly defined type will also be regarded as atomic. If the referred data type is a compound type, the new type will also be a compound. Some usage examples:

```
<dataTypeDef>
  <name>short</name>
  <synopsis>Alias to int16</synopsis>
  <typeRef>int16</typeRef>
</dataTypeDef>
<dataTypeDef>
  <name><name>ieeemacaddr</name>
  <synopsis>48-bit IEEE MAC address</synopsis>
  <typeRef>byte[6]</typeRef>
</dataTypeDef>
```

4.5.2. <atomic> Element for Deriving New Atomic Types

The <atomic> element allows the definition of a new atomic type from an existing atomic type, applying range restrictions and/or providing special enumerated values. Note that the <atomic> element can only use atomic types as base types, and its result is always another atomic type.

For example, the following snippet defines a new "dscp" data type:

```
<dataTypeDef>
  <name>dscp</name>
  <synopsis>Diffserv code point.</synopsis>
  <atomic>
    <baseType>uchar</baseType>
    <rangeRestriction>
      <allowedRange min="0" max="63"/>
    </rangeRestriction>
    <specialValues>
      <specialValue value="0">
        <name>DSCP-BE</name>
        <synopsis>Best Effort</synopsis>
      </specialValue>
      ...
    </specialValues>
  </atomic>
</dataTypeDef>
```

4.5.3. <array> Element to Define Arrays

The <array> element can be used to create a new compound data type as an array of a compound or an atomic data type. The type of the array entry can be specified either by referring to an existing type (using the <typeRef> element) or defining an unnamed type inside the <array> element using any of the <atomic>, <array>, <struct>, or <union> elements.

The array can be "fixed-size" or "variable-size", which is specified by the "type" attribute of the <array> element. The default is "variable-size". For variable size arrays an optional "max-length" attribute can specify the maximum allowed length. This attribute should be used to encode semantic limitations, and not implementation limitations. The latter should be handled by

capability attributes of LFB classes, and should never be included in data type definitions. If the "max-length" attribute is not provided, the array is regarded as of unlimited-size.

For fixed-size arrays a "length" attribute must be provided which specifies the constant size of the array.

The result of this construct is always a compound type, even if the array has a fixed size of 1.

Arrays can only be subscripted by integers, and will be presumed to start with index 0.

The following example shows the definition of a fixed size array with pre-defined data type as array elements:

```
<dataTypeDef>
  <name>dscp-mapping-table</name>
  <synopsys>
    A table of 64 DSCP values, used to re-map code space.
  </synopsis>
  <array type="fixed-size" length="64">
      <typeRef>dscp</typeRef>
  </array>
</dataTypeDef>
```

The following example defines a variable size array with an upper limit on its size:

```
<dataTypeDef>
  <name>mac-alias-table </name>
  <synopsys>A table with up to 8 IEEE MAC addresses</synopsis>
  <array type="variable-size" max-length="8">
      <typeRef>ieeemacaddr</typeRef>
  </array>
</dataTypeDef>
```

The following example shows the definition of an array with local (unnamed) type definition:

```
<dataTypeDef>
  <name>classification-table</name>
  <synopsys>
    A table of classification rules and result opcodes.
  </synopsis>
  <array type="variable-size">
    <struct>
```

```
        <element>
          <name>rule</name>
          <synopsis>The rule to match</synopsis>
          <typeRef>classrule</typeRef>
        </element>
        <element>
          <name>opcode</name>
          <synopsis>The result code</synopsis>
          <typeRef>opcode</typeRef>
        </element>
      </struct>
    </array>
  </dataTypeDef>
```

In the above example each entry of the array is a <struct> of two fileds ("rule" and "opcode").

4.5.4. <struct> Element to Define Structures

A structure is comprised of a collection of data elements. Each data element has a data type (either an atomic type or an existing compound type) and is assigned a name unique within the scope of the compound data type being defined. These serve the same function as "struct" in C, etc.

The actual type of the field can be defined by referring to an existing type (using the <typeDef> element), or can be a locally defined (unnamed) type created by any of the <atomic>, <array>, <struct>, or <union> elements.

The result of this construct is always regarded a compound type, even if the <struct> contains only one field.

An example:

```
<dataTypeDef>
  <name>ipv4prefix</name>
  <synopsis>
    IPv4 prefix defined by an address and a prefix length
  </synopsis>
  <struct>
    <element>
      <name>address</name>
      <synopsis>Address part</synopsis>
      <typeRef>ipv4addr</typeRef>
    </element>
    <element>
```

```
        <name>prefixlen</name>
        <synopsis>Prefix length part</synopsis>
        <atomic>
          <baseType>uchar</baseType>
          <rangeRestriction>
            <allowedRange min="0" max="32"/>
          </rangeRestriction>
        </atomic>
      </element>
    </struct>
  </dataTypeDef>
```

4.5.5. <union> Element to Define Union Types

Similar to the union declaration in C, this construct allows the definition of overlay types. Its format is identical to the <struct> element.

The result of this construct is always regarded a compound type, even if the union contains only one element.

4.5.6. Augmentations

Compound types can also be defined as augmentations of existing compound types. If the existing compound type is a structure, augmentation may add new elements to the type. They may replace the type of an existing element with an augmentation derived from the current type. They may not delete an existing element, nor may they replace the type of an existing element with one that is not an augmentation of the type that the element has in the basis for the augmentation. If the existing compound type is an array, augmentation means augmentation of the array element type.

One consequence of this is that augmentations are compatible with the compound type from which they are derived. As such, augmentations are useful in defining attributes for LFB subclasses with backward compatibility. In addition to adding new attributes to a class, the data type of an existing attribute may be replaced by an augmentation of that attribute, and still meet the compatibility rules for subclasses.

For example, consider a simple base LFB class A that has only one attribute (attr1) of type X. One way to derive class A1 from A can be by simply adding a second attribute (of any type). Another way to derive a class A2 from A can be by replacing the original attribute (attr1) in A of type X with one of type Y, where Y is an

augmentation of X. Both classes A1 and A2 are backward compatible with class A.

[EDITOR: How to support the concept of augmentation in the XML schema is for further study.]

4.6. <metadataDefs> Element for Metadata Definitions

The (optional) <metadataDefs> element in the library document contains one or more <metadataDef> elements. Each <metadataDef> element defines a metadata.

Each <metadataDef> element contains a unique name (NMTOKEN). Uniqueness is defined over all metadata defined in this library document and in all directly or indirectly included library documents. The <metadataDef> element also contains a brief synopsis, an optional detailed description, and a compulsory type definition information. Only atomic data types can be used as value types for metadata.

Two forms of type definitions are allowed. The first form uses the <typeRef> element to refer to an existing atomic data type defined in the <dataTypeDefs> element of the same library document or in one of the included library documents. The usage of the <typeRef> element is identical to how it is used in the <dataTypeDef> elements, except here it can only refer to atomic types.

[EDITOR: The latter restriction is not yet enforced by the XML schema.]

The second form is an explicit type definition using the <atomic> element. This element is used here in the same way as in the <dataTypeDef> elements.

The following example shows both usages:

```
<metadataDefs>
  <metadataDef>
    <name>NEXTHOPID</name>
    <synopsis>Refers to a Next Hop entry in NH LFB</synopsis>
    <typeRef>int32</typeRef>
  </metadataDef>
  <metadataDef>
    <name>CLASSID</name>
    <synopsis>
      Result of classification (0 means no match).
```

```
      </synopsis>
      <atomic>
        <baseType>int32</baseType>
        <specialValues>
          <specialValue value="0">
            <name>NOMATCH</name>
            <synopsis>
              Classification didn t result in match.
            </synopsis>
          </specialValue>
        </specialValues>
      </atomic>
    </metadataDef>
  </metadataDefs>
```

4.7. <LFBClassDefs> Element for LFB Class Definitions

The (optional) <LFBClassDefs> element can be used to define one or more LFB classes using <LFBClassDef> elements. Each <LFBClassDef> element defines an LFB class and includes the following elements:

- <name> provides the symbolic name of the LFB class. Example: "ipv4lpm"
- <synopsis> provides a short synopsis of the LFB class. Example: "IPv4 Longest Prefix Match Lookup LFB"
- <version> is the version indicator
- <derivedFrom> is the inheritance indicator
- <inputPorts> lists the input ports and their specifications
- <outputPorts> lists the output ports and their specifications
- <attributes> defines the operational attributes of the LFB
- <capabilities> defines the capability attributes of the LFB
- <description> contains the operational specification of the LFB

[EDITOR: LFB class names should be unique not only among classes defined in this document and in all included documents, but also unique across a large collection of libraries. Obviously some global control is needed to ensure such uniqueness. This subject requires further study.]

Here is a skeleton of an example LFB class definition:

```
<LFBClassDefs>
  <LFBClassDef>
    <name>ipv4lpm</name>
    <synopsis>IPv4 Longest Prefix Match Lookup LFB</synopsis>
    <version>1.0</version>
```

```
      <derivedFrom>baseclass</derivedFrom>

      <inputPorts>
        ...
      </inputPorts>

      <outputPorts>
        ...
      </outputPorts>

      <attributes>
        ...
      </attributes>

      <capabilities>
        ...
      </capabilities>

      <description>
        This LFB represents the IPv4 longest prefix match lookup
        operation.
        The modeled behavior is as follows:
           Blah-blah-blah.
      </description>

    </LFBClassDef>
    ...
  </LFBClassDefs>
```

Except the <name>, <synopsis>, and <version> elements, all other elements are optional in <LFBClassDef>, though when they are present, they must occur in the above order.

4.7.1. <derivedFrom> Element to Express LFB Inheritance

The optional <derivedFrom> element can be used to indicate that this class is a derivative of some other class. The content of this element must be the unique name (<name>) of another LFB class. The referred LFB class must be defined in the same library document or in one of the included library documents.

[EDITOR: The <derivedFrom> element will likely need to specify the version of the ancestor, which is not included in the schema yet. The process and rules of class derivation are still being studied.]

It is assumed that the derived class is backwards compatible with the base class.

4.7.2. <inputPorts> Element to Define LFB Inputs

The optional <inputPorts> element is used to define input ports. An LFB class may have zero, one, or more inputs. If the LFB class has no input ports, the <inputPorts> elements must be omitted. The <inputPorts> element can contain one or more <inputPort> elements, one for each port or port-group. We assume that most LFBs will have exactly one input. Multiple inputs with the same input type are modeled as one input group. Input groups are defined the same way as input ports by the <inputPort> element, differentiated only by an optional "group" attribute.

Multiple inputs with different input types should be avoided if possible (see discussion in Section 3.2.1). Some special LFBs will have no inputs at all. For example, a packet generator LFB does not need an input.

Single input ports and input port groups are both defined by the <inputPort> element, they are differentiated by only an optional "group" attribute.

The <inputPort> element contains the following elements:

. <name> provides the symbolic name of the input. Example: "in". Note that this symbolic name must be unique only within the scope of the LFB class.

. <synopsis> contains a brief description of the input. Example: "Normal packet input".

. <expectation> lists all allowed frame formats. Example: {"ipv4" and "ipv6"}. Note that this list should refer to names specified in the <frameDefs> element of the same library document or in any included library documents. The <expectation> element can also provide a list of required metadata. Example: {"classid", "vifid"}. This list should refer to names of metadata defined in the <metadataDefs> element in the same library document or in any included library documents. For each metadata it must be specified whether the metadata is required or optional. For each optional metadata a default value must be specified, which is used by the LFB if the metadata is not provided with a packet.

In addition, the optional "group" attribute of the <inputPort> element can specify if the port can behave as a port group, i.e., it is allowed to be instantiated. This is indicated by a "yes" value (the default value is "no").

An example <inputPorts> element, defining two input ports, the second one being an input port group:

```
<inputPorts>
  <inputPort>
    <name>in</name>
    <synopsis>Normal input</synopsis>
    <expectation>
      <frameExpected>
        <ref>ipv4</ref>
        <ref>ipv6</ref>
      </frameExpected>
      <metadataExpected>
        <ref>classid</ref>
        <ref>vifid</ref>
        <ref dependency="optional" defaultValue="0">vrfid</ref>
      </metadataExpected>
    </expectation>
  </inputPort>
  <inputPort group="yes">
    ... another input port ...
  </inputPort>
</inputPorts>
```

For each <inputPort>, the frame type expectations are defined by the <frameExpected> element using one or more <ref> elements (see example above). When multiple frame types are listed, it means that "one of these" frame types are expected. A packet of any other frame type is regarded as incompatible with this input port of the LFB class. The above example list two frames as expected frame types: "ipv4" and "ipv6".

Metadata expectations are specified by the <metadataExpected> element. In its simplest form this element can contain a list of <ref> elements, each referring to a metadata. When multiple instances of metadata are listed by <ref> elements, it means that "all of these" metadata must be received with each packet (except metadata that are marked as "optional" by the "dependency" attribute of the corresponding <ref> element). For a metadata that is specified "optional", a default value must be provided using the "defaultValue" attribute. The above example lists three metadata as expected metadata, two of which are mandatory ("classid" and "vifid"), and one being optional ("vrfid").

[EDITOR: How to express default values for byte[N] atomic types is yet to be defined.]

The schema also allows for more complex definitions of metadata expectations. For example, using the <one-of> element, a list of metadata can be specified to express that at least one of the specified metadata must be present with any packet. For example:

```
<metadataExpected>
  <one-of>
    <ref>prefixmask</ref>
    <ref>prefixlen</ref>
  </one-of>
</metadataExpected>
```

The above example specifies that either the "prefixmask" or the "prefixlen" metadata must be provided with any packet.

The two forms can also be combined, as it is shown in the following example:

```
<metadataExpected>
  <ref>classid</ref>
  <ref>vifid</ref>
  <ref dependency="optional" defaultValue="0">vrfid</ref>
  <one-of>
    <ref>prefixmask</ref>
    <ref>prefixlen</ref>
  </one-of>
</metadataExpected>
```

Although the schema is constructed to allow even more complex definition of metadata expectations, we do not discuss these here.

4.7.3. <outputPorts> Element to Define LFB Outputs

The optional <outputPorts> element is used to define output ports. An LFB class may have zero, one, or more outputs. If the LFB class has no output ports, the <outputPorts> element must be omitted. The <outputPorts> element can contain one or more <outputPort> elements, one for each port or port-group. If there are multiple outputs with the same output type, we model them as an output port group. Some special LFBs may have no outputs at all (e.g., Dropper).

Single output ports and output port groups are both defined by the <outputPort> element, they are differentiated by only an optional "group" attribute.

The <outputPort> element contains the following elements:

. <name> provides the symbolic name of the output. Example: "out". Note that the symbolic name must be unique only within the scope of the LFB class.

. <synopsis> contains a brief description of the output port. Example: "Normal packet output".

. <product> lists the allowed frame formats. Example: {"ipv4", "ipv6"}. Note that this list should refer to symbols specified in the <frameDefs> element in the same library document or in any included library documents. The <product> element may also contain the list of emitted (generated) metadata. Example: {"classid", "color"}. This list should refer to names of metadata specified in the <metadataDefs> element in the same library document or in any included library documents. For each generated metadata, it should be specified whether the metadata is always generated or generated only in certain conditions. This information is important when assessing compatibility between LFBs.

In addition, the optional "group" attribute of the <outputPort> element can specify if the port can behave as a port group, i.e., it is allowed to be instantiated. This is indicated by a "yes" value (the default value is "no").

The following example specifies two output ports, the second being an output port group:

```
<outputPorts>
  <outputPort>
    <name>out</name>
    <synopsis>Normal output</synopsis>
    <product>
      <frameProduced>
        <ref>ipv4</ref>
        <ref>ipv4bis</ref>
      </frameProduced>
      <metadataProduced>
        <ref>nhid</ref>
        <ref>nhtabid</ref>
      </metadataProduced>
    </product>
  </outputPort>
  <outputPort group="yes">
    <name>exc</name>
    <synopsis>Exception output port group</synopsis>
    <product>
      <frameProduced>
```

```
          <ref>ipv4</ref>
          <ref>ipv4bis</ref>
        </frameProduced>
        <metadataProduced>
          <ref availability="conditional">errorid</ref>
        </metadataProduced>
      </product>
    </outputPort>
  </outputPorts>
```

What types of frames and metadata the port produces are defined inside the <product> element in each <outputPort>. Within the <product> element, the list of frame types the port produces is listed in the <frameProduced> element. When more than one frame is listed, it means that "one of" these frames will be produced.

The list of metadata that is produced with each packet is listed in the optional <metadataProduced> element of the <product>. In its simplest form, this element can contain a list of <ref> elements, each referring to a metadata type. The meaning of such a list is that "all of" these metadata are provided with each packet, except those that are listed with the optional "availability" attribute set to "conditional." Similar to the <metadataExpected> element of the <inputPort>, the <metadataProduced> element supports more complex forms, which we do not discuss here further.

4.7.4. <attributes> Element to Define LFB Operational Attributes

Operational parameters of the LFBs that must be visible to the CEs are conceptualized in the model as the LFB attributes. These include, for example, flags, single parameter arguments, complex arguments, and tables. Note that the attributes here refer to only those operational parameters of the LFBs that must be visible to the CEs. Other variables that are internal to LFB implementation are not regarded as LFB attributes and hence are not covered.

Some examples for LFB attributes are:

- Configurable flags and switches selecting between operational modes of the LFB
- Number of inputs or ouputs in a port group
- Metadata CONSUME vs. PROPAGATE mode selectors
- Various configurable lookup tables, including interface tables, prefix tables, classification tables, DSCP mapping tables, MAC address tables, etc.
- Packet and byte counters
- Various event counters

- Number of current inputs or outputs for each input or output group
- Metadata CONSUME/PROPAGATE mode selector

There may be various access permission restrictions on what the CE can do with an LFB attribute. The following categories may be supported:

- No-access attributes. This is useful when multiple access modes maybe defined for a given attribute to allow some flexibility for different implementations.
- Read-only attributes.
- Read-write attributes.
- Write-only attributes. This could be any configurable data for which read capability is not provided to the CEs. (e.g., the security key information)
- Read-reset attributes. The CE can read and reset this resource, but cannot set it to an arbitrary value. Example: Counters.
- Firing-only attributes. A write attempt to this resource will trigger some specific actions in the LFB, but the actual value written is ignored.

The LFB class may define more than one possible access mode for a given attribute (for example, "write-only" and "read-write"), in which case it is left to the actual implementation to pick one of the modes. In such cases a corresponding capability attribute must inform the CE about the access mode the actual LFB instance supports (see next subsection on capability attributes).

The attributes of the LFB class are listed in the <attributes> element. Each attribute is defined by an <attribute> element. An <attribute> element contains the following elements:

- <name> defines the name of the attribute. This name must be unique among the attributes of the LFB class. Example: "version".
- <synopsis> should provide a brief description of the purpose of the attribute.
- The data type of the attribute can be defined either via a reference to a predefined data type or providing a local definition of the type. The former is provided by using the <typeRef> element, which must refer to the unique name of an existing data type defined in the <dataTypeDefs> element in the same library document or in any of the included library documents. When the data type is defined locally (unnamed type), one of the following elements can be used: <atomic>, <array>, <struct>, and <union>. Their usage is identical to

how they are used inside <dataTypeDef> elements (see Section 4.5).

. The optional <defaultValue> element can specify a default value for the attribute, which is applied when the LFB is initialized or reset. [EDITOR: A convention to define default values for compound data types and byte[N] atomic types is yet to be defined.]

In addition to the above elements, the <attribute> element includes an optional "access" attribute, which can take any of the following values or even a list of these values: "read-only", "read-write", "write-only", "read-reset", and "trigger-only". The default access mode is "read-write".

The following example defines two attributes for an LFB:

```
<attributes>
  <attribute access="read-only">
    <name>foo</name>
    <synopsis>number of things</synopsis>
    <typeRef>uint32</typeRef>
  </attribute>
  <attribute access="read-write write-only">
    <name>bar</name>
    <synopsis>number of this other thing</synopsis>
    <atomic>
      <baseType>uint32</baseType>
      <rangeRestriction>
        <allowedRange min="10" max="2000"/>
      </rangeRestriction>
    </atomic>
    <defaultValue>10</defaultValue>
  </attribute>
</attributes>
```

The first attribute ("foo") is a read-only 32-bit unsigned integer, defined by referring to the built-in "uint32" atomic type. The second attribute ("bar") is also an integer, but uses the <atomic> element to provide additional range restrictions. This attribute has two possible access modes, "read-write" or "write-only". A default value of 10 is provided.

Note that not all attributes are likely to exist at all times in a particular implementation. While the capabilities will frequently indicate this non-existence, CEs may attempt to reference non-existent or non-permitted attributes anyway. The FORCES protocol

mechanisms should include appropriate error indicators for this case.

The mechanism defined above for non-supported attributes can also apply to attempts to reference non-existent array elements or to set read-only elements.

4.7.5. <capabilities> Element to Define LFB Capability Attributes

The LFB class specification will provide some flexibility for the FE implementation regarding how the LFB class is implemented. For example the class may define some features optional, in which case the actual implementation may or may not provide the given feature. In these cases the CE must be able to query the LFB instance about the availability of the feature. In addition, the instance may have some limitations that are not inherent from the class definition, but rather the result of some implementation limitations. For example, an array attribute may be defined in the class definition as "unlimited" size, but the physical implementation may impose a hard limit on the size of the array.

Such capability related information is expressed by the capability attributes of the LFB class. The capability attributes are always read-only attributes, and they are listed in a separate <capabilities> element in the <LFBClassDef>. The <capabilities> element contains one or more <capability> elements, each defining one capability attribute. The format of the <capability> element is almost the same as the <attribute> element, it differs in two aspects: it lacks the access mode attribute (because it is always read-only), and it lacks the <defaultValue> element (because default value is not applicable to read-only attributes).

Some examples of capability attributes:

- The version of the LFB class that this LFB instance complies with;
- Supported optional features of the LFB class;
- Maximum number of configurable outputs for an output group;
- Metadata pass-through limitations of the LFB;
- Maximum size of configurable attribute tables;
- Additional range restriction on operational attributes;
- Supported access modes of certain attributes (if the access mode of an operational attribute is specified as a list of two or mode modes).

The following example lists two capability attributes:

```
<capabilities>
  <capability>
    <name>version</name>
    <synopsis>
      LFB class version this instance is compliant with.
    </synopsis>
    <typeRef>version</typeRef>
  </capability>
  <capability>
    <name>limitBar</name>
    <synopsis>
      Maximum value of the "bar" attribute.
    </synopsis>
    <typeRef>uint16</typeRef>
  </capability>
</capabilities>
```

4.7.6. <description> Element for LFB Operational Specification

The <description> element of the <LFBClass> provides unstructured text (in XML sense) to verbally describe what the LFB does.

4.8. XML Schema for LFB Class Library Documents

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
 xmlns="http://ietf.org/forces/1.0/lfbmodel"
 xmlns:lfb="http://ietf.org/forces/1.0/lfbmodel"
 targetNamespace="http://ietf.org/forces/1.0/lfbmodel"
 attributeFormDefault="unqualified"
 elementFormDefault="qualified">
<xsd:annotation>
  <xsd:documentation xml:lang="en">
  Schema for Defining LFB Classes and associated types (frames,
  data types for LFB attributes, and metadata).
  </xsd:documentation>
</xsd:annotation>
<xsd:element name="description" type="xsd:string"/>
<xsd:element name="synopsis" type="xsd:string"/>
<!-- Document root element: LFBLibrary -->
<xsd:element name="LFBLibrary">
  <xsd:complexType>
    <xsd:sequence>
      <xsd:element ref="description" minOccurs="0"/>
      <xsd:element name="load" type="loadType" minOccurs="0"
                   maxOccurs="unbounded"/>
      <xsd:element name="frameDefs" type="frameDefsType"
```

```
                    minOccurs="0"/>
        <xsd:element name="dataTypeDefs" type="dataTypeDefsType"
                    minOccurs="0"/>
        <xsd:element name="metadataDefs" type="metadataDefsType"
                    minOccurs="0"/>
        <xsd:element name="LFBClassDefs" type="LFBClassDefsType"
                    minOccurs="0"/>
      </xsd:sequence>
      <xsd:attribute name="provides" type="xsd:Name" use="required"/>
    </xsd:complexType>
    <!-- Uniqueness constraints -->
    <xsd:key name="frame">
      <xsd:selector xpath="lfb:frameDefs/lfb:frameDef"/>
      <xsd:field xpath="lfb:name"/>
    </xsd:key>
    <xsd:key name="dataType">
      <xsd:selector xpath="lfb:dataTypeDefs/lfb:dataTypeDef"/>
      <xsd:field xpath="lfb:name"/>
    </xsd:key>
    <xsd:key name="metadataDef">
      <xsd:selector xpath="lfb:metadataDefs/lfb:metadataDef"/>
      <xsd:field xpath="lfb:name"/>
    </xsd:key>
    <xsd:key name="LFBClassDef">
      <xsd:selector xpath="lfb:LFBClassDefs/lfb:LFBClassDef"/>
      <xsd:field xpath="lfb:name"/>
    </xsd:key>
  </xsd:element>
  <xsd:complexType name="loadType">
    <xsd:attribute name="library" type="xsd:Name" use="required"/>
    <xsd:attribute name="location" type="xsd:anyURI" use="optional"/>
  </xsd:complexType>
  <xsd:complexType name="frameDefsType">
    <xsd:sequence>
      <xsd:element name="frameDef" maxOccurs="unbounded">
        <xsd:complexType>
          <xsd:sequence>
            <xsd:element name="name" type="xsd:NMTOKEN"/>
            <xsd:element ref="synopsis"/>
            <xsd:element ref="description" minOccurs="0"/>
          </xsd:sequence>
        </xsd:complexType>
      </xsd:element>
    </xsd:sequence>
  </xsd:complexType>
  <xsd:complexType name="dataTypeDefsType">
    <xsd:sequence>
```

```
      <xsd:element name="dataTypeDef" maxOccurs="unbounded">
        <xsd:complexType>
          <xsd:sequence>
            <xsd:element name="name" type="xsd:NMTOKEN"/>
            <xsd:element ref="synopsis"/>
            <xsd:element ref="description" minOccurs="0"/>
            <xsd:group ref="typeDeclarationGroup"/>
          </xsd:sequence>
        </xsd:complexType>
      </xsd:element>
    </xsd:sequence>
  </xsd:complexType>
  <!--
     Predefined (built-in) atomic data-types are:
         char, uchar, int16, uint16, int32, uint32, int64, uint64,
         string[N], byte[N],
         float16, float32, float64
  -->
  <xsd:group name="typeDeclarationGroup">
    <xsd:choice>
      <xsd:element name="typeRef" type="typeRefNMTOKEN"/>
      <xsd:element name="atomic" type="atomicType"/>
      <xsd:element name="array" type="arrayType"/>
      <xsd:element name="struct" type="structType"/>
      <xsd:element name="union" type="structType"/>
    </xsd:choice>
  </xsd:group>
  <xsd:simpleType name="typeRefNMTOKEN">
    <xsd:restriction base="xsd:token">
      <xsd:pattern value="\c+"/>
      <xsd:pattern value="string\[\d+\]"/>
      <xsd:pattern value="byte\[\d+\]"/>
    </xsd:restriction>
  </xsd:simpleType>
  <xsd:complexType name="atomicType">
    <xsd:sequence>
      <xsd:element name="baseType" type="typeRefNMTOKEN"/>
      <xsd:element name="rangeRestriction"
                      type="rangeRestrictionType minOccurs="0"/>
      <xsd:element name="specialValues" type="specialValuesType"
                   minOccurs="0"/>
    </xsd:sequence>
  </xsd:complexType>
  <xsd:complexType name="rangeRestrictionType">
    <xsd:sequence>
      <xsd:element name="allowedRange" maxOccurs="unbounded">
        <xsd:complexType>
```

```
          <xsd:attribute name="min" type="xsd:integer"
   use="required"/>
          <xsd:attribute name="max" type="xsd:integer"
   use="required"/>
        </xsd:complexType>
      </xsd:element>
    </xsd:sequence>
   </xsd:complexType>
   <xsd:complexType name="specialValuesType">
    <xsd:sequence>
      <xsd:element name="specialValue" maxOccurs="unbounded">
        <xsd:complexType>
          <xsd:sequence>
            <xsd:element name="name" type="xsd:NMTOKEN"/>
            <xsd:element ref="synopsis"/>
          </xsd:sequence>
          <xsd:attribute name="value" type="xsd:token"/>
        </xsd:complexType>
      </xsd:element>
    </xsd:sequence>
   </xsd:complexType>
   <xsd:complexType name="arrayType">
    <xsd:sequence>
      <xsd:group ref="typeDeclarationGroup"/>
    </xsd:sequence>
    <xsd:attribute name="type" use="optional"
                   default="variable-size">
      <xsd:simpleType>
        <xsd:restriction base="xsd:string">
          <xsd:enumeration value="fixed-size"/>
          <xsd:enumeration value="variable-size"/>
        </xsd:restriction>
      </xsd:simpleType>
    </xsd:attribute>
    <xsd:attribute name="length" type="xsd:integer" use="optional"/>
    <xsd:attribute name="maxLength" type="xsd:integer"
                   use="optional"/>
   </xsd:complexType>
   <xsd:complexType name="structType">
    <xsd:sequence>
      <xsd:element name="element" maxOccurs="unbounded">
        <xsd:complexType>
          <xsd:sequence>
            <xsd:element name="name" type="xsd:NMTOKEN"/>
            <xsd:element ref="synopsis"/>
            <xsd:group ref="typeDeclarationGroup"/>
          </xsd:sequence>
```

```
        </xsd:complexType>
      </xsd:element>
    </xsd:sequence>
  </xsd:complexType>
  <xsd:complexType name="metadataDefsType">
    <xsd:sequence>
      <xsd:element name="metadataDef" maxOccurs="unbounded">
        <xsd:complexType>
          <xsd:sequence>
            <xsd:element name="name" type="xsd:NMTOKEN"/>
            <xsd:element ref="synopsis"/>
            <xsd:element ref="description" minOccurs="0"/>
            <xsd:choice>
              <xsd:element name="typeRef" type="typeRefNMTOKEN"/>
              <xsd:element name="atomic" type="atomicType"/>
            </xsd:choice>
          </xsd:sequence>
        </xsd:complexType>
      </xsd:element>
    </xsd:sequence>
  </xsd:complexType>
  <xsd:complexType name="LFBClassDefsType">
    <xsd:sequence>
      <xsd:element name="LFBClassDef" maxOccurs="unbounded">
        <xsd:complexType>
          <xsd:sequence>
            <xsd:element name="name" type="xsd:NMTOKEN"/>
            <xsd:element ref="synopsis"/>
            <xsd:element name="version" type="versionType"/>
            <xsd:element name="derivedFrom" type="xsd:NMTOKEN"
                         minOccurs="0"/>
            <xsd:element name="inputPorts" type="inputPortsType"
                         minOccurs="0"/>
            <xsd:element name="outputPorts" type="outputPortsType"
                         minOccurs="0"/>
            <xsd:element name="attributes" type="LFBAttributesType"
                         minOccurs="0"/>
            <xsd:element name="capabilities"
                            type="LFBCapabilitiesType"
minOccurs="0"/>
            <xsd:element ref="description" minOccurs="0"/>
          </xsd:sequence>
        </xsd:complexType>
        <!-- Key constraint to ensure unique attribute names within
             a class:
        -->
        <xsd:key name="attributes">
```

```
          <xsd:selector xpath="lfb:attributes/lfb:attribute"/>
          <xsd:field xpath="lfb:name"/>
        </xsd:key>
        <xsd:key name="capabilities">
          <xsd:selector xpath="lfb:capabilities/lfb:capability"/>
          <xsd:field xpath="lfb:name"/>
        </xsd:key>
      </xsd:element>
    </xsd:sequence>
  </xsd:complexType>
  <xsd:simpleType name="versionType">
    <xsd:restriction base="xsd:NMTOKEN">
      <xsd:pattern value="[1-9][0-9]*\.([1-9][0-9]*|0)"/>
    </xsd:restriction>
  </xsd:simpleType>
  <xsd:complexType name="inputPortsType">
    <xsd:sequence>
      <xsd:element name="inputPort" type="inputPortType"
                   maxOccurs="unbounded"/>
    </xsd:sequence>
  </xsd:complexType>
  <xsd:complexType name="inputPortType">
    <xsd:sequence>
      <xsd:element name="name" type="xsd:NMTOKEN"/>
      <xsd:element ref="synopsis"/>
      <xsd:element name="expectation" type="portExpectationType"/>
      <xsd:element ref="description" minOccurs="0"/>
    </xsd:sequence>
    <xsd:attribute name="group" type="booleanType" use="optional"
                   default="no"/>
  </xsd:complexType>
  <xsd:complexType name="portExpectationType">
    <xsd:sequence>
      <xsd:element name="frameExpected" minOccurs="0">
        <xsd:complexType>
          <xsd:sequence>
            <!-- ref must refer to a name of a defined frame type -->
            <xsd:element name="ref" type="xsd:string"
                         maxOccurs="unbounded"/>
          </xsd:sequence>
        </xsd:complexType>
      </xsd:element>
      <xsd:element name="metadataExpected" minOccurs="0">
        <xsd:complexType>
          <xsd:choice maxOccurs="unbounded">
            <!-- ref must refer to a name of a defined metadata -->
            <xsd:element name="ref" type="metadataInputRefType"/>
```

```
            <xsd:element name="one-of"
                         type="metadataInputChoiceType"/>
          </xsd:choice>
        </xsd:complexType>
      </xsd:element>
    </xsd:sequence>
  </xsd:complexType>
  <xsd:complexType name="metadataInputChoiceType">
    <xsd:choice minOccurs="2" maxOccurs="unbounded">
      <!-- ref must refer to a name of a defined metadata -->
      <xsd:element name="ref" type="xsd:NMTOKEN"/>
      <xsd:element name="one-of" type="metadataInputChoiceType"/>
      <xsd:element name="metadataSet" type="metadataInputSetType"/>
    </xsd:choice>
  </xsd:complexType>
  <xsd:complexType name="metadataInputSetType">
    <xsd:choice minOccurs="2" maxOccurs="unbounded">
      <!-- ref must refer to a name of a defined metadata -->
      <xsd:element name="ref" type="metadataInputRefType"/>
      <xsd:element name="one-of" type="metadataInputChoiceType"/>
    </xsd:choice>
  </xsd:complexType>
  <xsd:complexType name="metadataInputRefType">
    <xsd:simpleContent>
      <xsd:extension base="xsd:NMTOKEN">
        <xsd:attribute name="dependency" use="optional"
                       default="required">
          <xsd:simpleType>
            <xsd:restriction base="xsd:string">
              <xsd:enumeration value="required"/>
              <xsd:enumeration value="optional"/>
            </xsd:restriction>
          </xsd:simpleType>
        </xsd:attribute>
        <xsd:attribute name="defaultValue" type="xsd:token"
                       use="optional"/>
      </xsd:extension>
    </xsd:simpleContent>
  </xsd:complexType>
  <xsd:complexType name="outputPortsType">
    <xsd:sequence>
      <xsd:element name="outputPort" type="outputPortType"
                   maxOccurs="unbounded"/>
    </xsd:sequence>
  </xsd:complexType>
  <xsd:complexType name="outputPortType">
    <xsd:sequence>
```

```
      <xsd:element name="name" type="xsd:NMTOKEN"/>
      <xsd:element ref="synopsis"/>
      <xsd:element name="product" type="portProductType"/>
      <xsd:element ref="description" minOccurs="0"/>
    </xsd:sequence>
    <xsd:attribute name="group" type="booleanType" use="optional"
                   default="no"/>
  </xsd:complexType>
  <xsd:complexType name="portProductType">
    <xsd:sequence>
      <xsd:element name="frameProduced">
        <xsd:complexType>
          <xsd:sequence>
            <!-- ref must refer to a name of a defined frame type -->
            <xsd:element name="ref" type="xsd:NMTOKEN"
                         maxOccurs="unbounded"/>
          </xsd:sequence>
        </xsd:complexType>
      </xsd:element>
      <xsd:element name="metadataProduced" minOccurs="0">
        <xsd:complexType>
          <xsd:choice maxOccurs="unbounded">
            <!-- ref must refer to a name of a defined metadata -->
            <xsd:element name="ref" type="metadataOutputRefType"/>
            <xsd:element name="one-of"
                         type="metadataOutputChoiceType"/>
          </xsd:choice>
        </xsd:complexType>
      </xsd:element>
    </xsd:sequence>
  </xsd:complexType>
  <xsd:complexType name="metadataOutputChoiceType">
    <xsd:choice minOccurs="2" maxOccurs="unbounded">
      <!-- ref must refer to a name of a defined metadata -->
      <xsd:element name="ref" type="xsd:NMTOKEN"/>
      <xsd:element name="one-of" type="metadataOutputChoiceType"/>
      <xsd:element name="metadataSet" type="metadataOutputSetType"/>
    </xsd:choice>
  </xsd:complexType>
  <xsd:complexType name="metadataOutputSetType">
    <xsd:choice minOccurs="2" maxOccurs="unbounded">
      <!-- ref must refer to a name of a defined metadata -->
      <xsd:element name="ref" type="metadataOutputRefType"/>
      <xsd:element name="one-of" type="metadataOutputChoiceType"/>
    </xsd:choice>
  </xsd:complexType>
  <xsd:complexType name="metadataOutputRefType">
```

```
    <xsd:simpleContent>
      <xsd:extension base="xsd:NMTOKEN">
        <xsd:attribute name="availability" use="optional"
                       default="unconditional">
          <xsd:simpleType>
            <xsd:restriction base="xsd:string">
              <xsd:enumeration value="unconditional"/>
              <xsd:enumeration value="conditional"/>
            </xsd:restriction>
          </xsd:simpleType>
        </xsd:attribute>
      </xsd:extension>
    </xsd:simpleContent>
  </xsd:complexType>
  <xsd:complexType name="LFBAttributesType">
    <xsd:sequence>
      <xsd:element name="attribute" maxOccurs="unbounded">
        <xsd:complexType>
          <xsd:sequence>
            <xsd:element name="name" type="xsd:NMTOKEN"/>
            <xsd:element ref="synopsis"/>
            <xsd:element ref="description" minOccurs="0"/>
            <xsd:group ref="typeDeclarationGroup"/>
            <xsd:element name="defaultValue" type="xsd:token"
                         minOccurs="0"/>
          </xsd:sequence>
          <xsd:attribute name="access" use="optional"
                         default="read-write">
            <xsd:simpleType>
              <xsd:list itemType="accessModeType"/>
            </xsd:simpleType>
          </xsd:attribute>
        </xsd:complexType>
      </xsd:element>
    </xsd:sequence>
  </xsd:complexType>
  <xsd:simpleType name="accessModeType">
    <xsd:restriction base="xsd:NMTOKEN">
      <xsd:enumeration value="read-only"/>
      <xsd:enumeration value="read-write"/>
      <xsd:enumeration value="write-only"/>
      <xsd:enumeration value="read-reset"/>
      <xsd:enumeration value="trigger-only"/>
    </xsd:restriction>
  </xsd:simpleType>
  <xsd:complexType name="LFBCapabilitiesType">
    <xsd:sequence>
```

```
      <xsd:element name="capability" maxOccurs="unbounded">
        <xsd:complexType>
          <xsd:sequence>
            <xsd:element name="name" type="xsd:NMTOKEN"/>
            <xsd:element ref="synopsis"/>
            <xsd:element ref="description" minOccurs="0"/>
            <xsd:group ref="typeDeclarationGroup"/>
          </xsd:sequence>
        </xsd:complexType>
      </xsd:element>
    </xsd:sequence>
  </xsd:complexType>
  <xsd:simpleType name="booleanType">
    <xsd:restriction base="xsd:string">
      <xsd:enumeration value="yes"/>
      <xsd:enumeration value="no"/>
    </xsd:restriction>
  </xsd:simpleType>
  </xsd:schema>
```

5. FE Attributes and Capabilities

A ForCES forwarding element handles traffic on behalf of a ForCES control element. While the standards will describe the protocol and mechanisms for this control, different implementations and different instances will have different capabilities. The CE needs to be able to determine what each instance it is responsible for is actually capable of doing. As stated previously, this is an approximation. The CE is expected to be prepared to cope with errors in requests and variations in detail not captured by the capabilities information about an FE.

In addition to its capabilities, an FE will have some information (attributes) that can be used in understanding and controlling the forwarding operations. Some of the attributes will be read only, while others will also be writeable.

The ForCES protocol will define the actual mechanism for getting and setting attribute information. This model defines the starting set of information that will be available. This definition includes the semantics and the structuring of the information. It also provides for extensions to this information.

In order to crisply define the attribute information and structure, this document describes the attributes as information in an abstract XML document. Conceptually, each FE contains such a

document. The document structure is defined by the XML Schema contained in this model. Operationally, the ForCES protocol refers to information contained in that document in order to read or write FE attributes and capabilities. This document is an abstract representation of the information. There is no requirement that such a document actually exist in memory. Unless the ForCES protocol calls for transfer of the information in XML, the information is not required to ever be represented in the FE in XML. The XML schema serves only to identify the elements and structure of the information.

The subsections in this part of the document provide the details on this aspect of the FE model. 5.1 gives the XML schema for the abstract FE attribute document. 5.2 elaborates on each of the defined attributes of the FE, following the hierarchy of the schema. 5.3 provides an example XML FE attribute document to clarify the meaning of 5.1 and 5.2.

5.1. XML Schema for FE Attribute Documents

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">

 <xsd:annotation>
  <xsd:documentation xml:lang="en">
   Schema for the Abstract FE Attributes and Capabilities Document
  </xsd:documentation>
 </xsd:annotation>

 <xsd:element name="FEDocument">
  <xsd:complexType>
   <xsd:sequence>
    <xsd:element name="FECapabilities" type="FECapabilitiesType"
         minOccurs="0" maxOccurs="1"/>
    <xsd:element name="FEAttributes" type="FEAttributesType"
         minOccurs="0" maxOccurs="1"/>
   </xsd:sequence>
  </xsd:complexType>
 </xsd:element>

 <xsd:complexType name="FECapabilitiesType">
  <xsd:sequence>
   <xsd:element name="ModifiableLFBTopology" type="xsd:boolean"
        minOccurs="0" maxOccurs="1"/>
   <xsd:element name="SupportedLFBs" minOccurs="0" maxOccurs="1">
    <xsd:complexType>
     <xsd:sequence>
```

```
        <xsd:element name="SupportedLFB" type="SupportedLFBType"
             minOccurs="1" maxOccurs="unbounded"/>
       </xsd:sequence>
      </xsd:complexType>
     </xsd:element>
     <xsd:element name="SupportedAttributes"
          type="SupportedAttributesType"
          minOccurs="0" maxOccurs="1"/>
    </xsd:sequence>
   </xsd:complexType>

   <xsd:complexType name="SupportedLFBType">
    <xsd:sequence>
     <!-- the name of a supported LFB -->
     <xsd:element name="LFBName" type="xsd:NMTOKEN"/>
     <!-- how many of this LFB class can exist -->
     <xsd:element name="LFBOccurrenceLimit"
          type="xsd:nonNegativeInteger" minOccurs="0" maxOccurs="1"/>
     <!-- For each port group, how many ports can exist -->
     <xsd:element name="PortGroupLimits" minOccurs="0" maxOccurs="1">
      <xsd:complexType>
       <xsd:sequence>
        <xsd:element name="PortGroupLimit" minOccurs="0"
             maxOccurs="unbounded">
         <xsd:complexType>
          <xsd:sequence>
           <xsd:element name="PortGroupName" type="xsd:NMTOKEN"/>
           <xsd:element name="MinPortCount"
                type="xsd:nonNegativeInteger"
                minOccurs="0" maxOccurs="1"/>
           <xsd:element name="MaxPortCount"
                type="xsd:nonNegativeInteger"
                minOccurs="0" maxOccurs="1"/>
          </xsd:sequence>
         </xsd:complexType>
        </xsd:element>
       </xsd:sequence>
      </xsd:complexType>
     </xsd:element>
     <!-- for the named LFB Class, the LFB Classes it may follow -->
     <xsd:element name="CanOccurAfters" minOccurs="0" maxOccurs="1">
      <xsd:complexType>
       <xsd:sequence>
        <xsd:element name="CanOccurAfter"
             type="LFBAdjacencyLimitType"
             minOccurs="0" maxOccurs="unbounded"/>
       </xsd:sequence>
```

```
       </xsd:complexType>
      </xsd:element>

      <!-- for the named LFB Class, which LFB Classes may follow -->
      <xsd:element name="CanOccurBefores" minOccurs="0" maxOccurs="1">
       <xsd:complexType>
        <xsd:sequence>
         <xsd:element name="CanOccurBefore"
              type="LFBAdjacencyLimitType"
              minOccurs="0" maxOccurs="unbounded"/>
        </xsd:sequence>
       </xsd:complexType>
      </xsd:element>
      <!-- information defined by the Class Definition -->
      <xsd:element name="LFBClassCapabilities" type="xsd:anyType"
           minOccurs="0" maxOccurs="1"/>
     </xsd:sequence>
    </xsd:complexType>

    <xsd:complexType name="LFBAdjacencyLimitType">
     <xsd:sequence>
      <xsd:element name="NeighborLFB" type="xsd:NMTOKEN"/>
      <xsd:element name="viaPort" type="xsd:NMTOKEN"
           minOccurs="0" maxOccurs="unbounded"/>
     </xsd:sequence>
    </xsd:complexType>

    <xsd:complexType name="SupportedAttributesType">
     <xsd:sequence>
      <xsd:element name="SupportedAttribute"
           minOccurs="0" maxOccurs="unbounded">
       <xsd:complexType>
        <xsd:sequence>
         <xsd:element name="AttributeName" type="xsd:NMTOKEN"/>
         <xsd:element name="AccessModes" type="xsd:NMTOKEN"/>
        </xsd:sequence>
       </xsd:complexType>
      </xsd:element>
     </xsd:sequence>
    </xsd:complexType>

    <xsd:complexType name="FEAttributesType">
     <xsd:sequence>
      <xsd:element name="Vendor" type="xsd:string" minOccurs="0"/>
      <xsd:element name="Model" type="xsd:string" minOccurs="0"/>
      <xsd:element name="FEStatus" type="FEStateType" minOccurs="0"/>
      <xsd:element name="LFBInstances" minOccurs="0" maxOccurs="1">
```

```
       <xsd:complexType>
        <xsd:sequence>
         <xsd:element name="LFBInstance" minOccurs="0"
              maxOccurs="unbounded">
          <xsd:complexType>
           <xsd:sequence>
            <xsd:element name="LFBClassName" type="xsd:NMTOKEN"/>">
            <xsd:element name="LFBInstanceID" type="xsd:NMTOKEN"/>">
           </xsd:sequence>
          </xsd:complexType>
         </xsd:element>
        </xsd:sequence>
       </xsd:complexType>
      </xsd:element>
      <xsd:element name="LFBTopology" type="LFBTopologyType"
           minOccurs="0" maxOccurs="1"/>
      <xsd:element name="FEConfiguredNeighbors" minOccurs="0"
           maxOccurs="1">
       <xsd:complexType>
        <xsd:sequence>
         <xsd:element name="FEConfiguredNeighbor"
              type="FEConfiguredNeighborType"
              minOccurs="0" maxOccurs="unbounded"/>
        </xsd:sequence>
       </xsd:complexType>
      </xsd:element>
     </xsd:sequence>
    </xsd:complexType>

    <xsd:complexType name="LFBTopologyType">
     <xsd:sequence>
      <xsd:element name="LFBLink" minOccurs="0" maxOccurs="unbounded">
       <xsd:complexType>
        <xsd:sequence>
         <xsd:element name="FromLFBID" type="xsd:NMTOKEN"/>
         <xsd:element name="FromPortGroup" type="xsd:NMTOKEN"/>
         <xsd:element name="FromPortIndex"
              type="xsd:nonNegativeInteger"/>
         <xsd:element name="ToLFBID" type="xsd:NMTOKEN"/>
         <xsd:element name="ToPortGroup" type="xsd:NMTOKEN"/>
         <xsd:element name="ToPortIndex"
              type="xsd:nonNegativeInteger"/>
        </xsd:sequence>
       </xsd:complexType>
      </xsd:element>
     </xsd:sequence>
    </xsd:complexType>
```

```
    <xsd:complexType name="FEConfiguredNeighborType">
     <xsd:sequence>
      <xsd:element name="NeighborID" type="xsd:anyType"/>
      <xsd:element name="NeighborInterface" type="xsd:anyType"/>
      <xsd:element name="NeighborNetworkAddress" type="xsd:anyType"
           minOccurs="0" maxOccurs="1"/>
      <xsd:element name="NeighborMACAddress" type="xsd:anyType"
           minOccurs="0" maxOccurs="1"/>
     </xsd:sequence>
    </xsd:complexType>

    <!-- The values for the simple state attribute -->
    <!-- These should probably be directly encodable in the -->
    <!-- protocol so they may end up numeric instead of strings -->
    <xsd:simpleType name="FEStateType">
     <xsd:restriction base="xsd:NMTOKEN">
      <xsd:enumeration value="AdminDisable"/>
      <xsd:enumeration value="OperDisable"/>
      <xsd:enumeration value="OperEnable"/>
     </xsd:restriction>
    </xsd:simpleType>

   </xsd:schema>
```

5.2. FEDocument

An instance of this document captures the capabilities and FE level attribute / state information about a given FE. Currently, two elements are allowed in the FEDocument, FECapabilities and FEAttributes.

At the moment, all capability and attribute information in this abstract document is defined as optional. We may wish to mandate support for some capability and/or attribute information.

If a protocol using binary encoding of this information is adopted by the ForCES working group, then each relevant element defined in the schema will have a "ProtocolEncoding" attribute added, with a "Fixed" value providing the value that is used in the protocol for that element, so that the XML and the on the wire protocol can be correlated.

5.2.1. FECapabilities

This element, which if it occurs must occur only once, contains all the capability related information about the FE. Capability information is always considered to be read-only.

The currently defined elements allowed within the FECapabilities element are ModifiableLFBTopology, LFBsSupported, WriteableAttributes and ReadableAttributes.

5.2.1.1. ModifiableLFBTopology

This element has a boolean value. This element indicates whether the LFB topology of the FE may be changed by the CE. If the element is absent, the default value is assumed to be true, and the CE presumes the LFB topology may be changed. If the value is present and set to false, the LFB topology of the FE is fixed. In that case, the LFBs supported clause may be omitted, and the list of supported LFBs is inferred by the CE from the LFB topology information. If the list of supported LFBs is provided when ModifiableLFBTopology is false, the CanOccurBefore and CanOccurAfter information should be omitted.

5.2.1.2. SupportedLFBs and SupportedLFB

One capability that the FE should include is the list of supported LFB classes. The SupportedLFBs element, which occurs at most once, serves as a wrapper for the list of LFB classes supported. Each class is described in a SupportedLFB element.

Each occurrence of the SupportedLFB element describes an LFB class that the FE supports. In addition to indicating that the FE supports the class, FEs with modifiable LFB topology should include information about how LFBs of the specified class may be connected to other LFBs. This information should describe which LFB classes the specified LFB class may succeed or precede in the LFB topology. The FE should include information as to which port groups may be connected to the given adjacent LFB class. If port group information is omitted, it is assumed that all port groups may be used.

5.2.1.2.1. LFBName

This element has as its value the name of the LFB being described.

5.2.1.2.2. LFBOccurrenceLimit

This element, if present, indicates the largest number of instances of this LFB class the FE can support. For FEs that do not have the

capability to create or destroy LFB instances, this can either be omitted or be the same as the number of LFB instances of this class contained in the LFB list attribute.

5.2.1.2.3. PortGroupLimits and PortGroupLimit

The PortGroupLimits element is the wrapper to hold information about the port groups supported by the LFB class. It holds multiple occurrences of the PortGroupLimit element.

Each occurrence of the PortGroupLimit element contains the port occurrence information for a single port group of the LFB class. Each occurrence has the name of the port group in the PortGroupName element, the fewest number of ports that can exist in the group in the MinPortCount element, and the largest number of ports that can exist in the group in the MaxPortCount element.

5.2.1.2.4.CanOccurAfters and CanOccurAfter

The CanOccurAfters element is a wrapper to hold the multiple occurrences of the CanOccurAfter permissible placement information.

The CanOccurAfter element describes a permissible positioning of the SupportedLFB. Specifically, it names an LFB that can topologically precede the SupportedLFB. That is, the SupportedLFB can have an input port connected to an output port of the LFB that it CanOccurAfter. The LFB class that the SupportedLFB can follow is identified by the NeighborLFB element of the CanOccurAfter element. If this neighbor can only be connected to a specific set of input port groups, then the viaPort element is included. This element occurs once for each input port group of the SupportedLFB that can be connected to an output port of the NeighborLFB.

[e.g., Within a SupportedLFB element, each CanOccurAfter element must have a unique NeighborLFB, and within each CanOccurAfter element each viaPort must represent a unique and valid input port group of the SupportedLFB. The "unique" clauses for this have not yet been added to the schema.]

5.2.1.2.5. CanOccurBefores and CanOccurBefore

The CanOccurBefores element is a wrapper to hold the multiple occurrences of the CanOccurBefore permissible placement information.

The CanOccurBefore element similarly lists those LFB classes that the SupportedLFB may precede in the topology. In this element, the

viaPort element represents the output port group of the SupportedLFB that may be connected to the NeighborLFB. As with CanOccurAfter, viaPort may occur multiple times if multiple output ports may legitimately connect to the given NeighborLFB class.

[And a similar set of uniqueness constraints apply to the CanOccurBefore clauses, even though an LFB may occur both in CanOccurAfter and CanOccurBefore.]

5.2.1.2.6. LFBClassCapabilities

This element contains capability information about the subject LFB class whose structure and semantics are defined by the LFB class definition.

5.2.1.3. SupportedAttributes

This element serves as a wrapper to hold the information about attributed related capabilities. Specifically, attributes should be described in this element if:

a) they are optional elements in the standard and are supported by the FE, or
b) the standard allows for a range of access permissions (for example, read-only or read-write).

Each attribute so described is contained in the SupportedAttributes element. That element contains an AttributeName element whose value is the name of the element being described and an AccessModes element, whose value is the list of permissions.

5.2.2. FEAttributes

The FEAttributes element contains the attributes of the FE that are not considered "capabilities". Some of these attributes are writeable, and some are read-only, which should be indicated by the capability information. At the moment, the set of attributes is woefully incomplete. Each attribute is identified by a unique element tag, and the value of the element is the value of the attribute.

5.2.2.1. FEStatus

This attribute carries the overall state of the FE. For now, it is restricted to the strings AdminDisable, OperDisable and OperEnable.

5.2.2.2.LFBInstances and LFBInstance

The LFBInstances element serves as a wrapper to hold the multiple occurrences of the LFBInstance information about individual LFB instances on the FE.

Each occurrence of the LFBInstance element describes a single LFB instance. Each element contains an LFBClassName indicating what class this instance has, and an LFBInstanceID indicating the ID used for referring to this instance. For now, the ID uses the NMTOKEN construction. Further protocol work is likely to replace this with a range restricted integer.

5.2.2.3. LFBTopology and LFBLink

This optional element contains the information about each inter-LFB link inside the FE. Each link is described in an LFBLink element. This element contains sufficient information to identify precisely the end points of a link. The FromLFBID and ToLFBID fields indicate the LFB instances at each end of the link, and must reference LFBs in the LFB instance table. The FromPortGroup and ToPortGroup must identify output and input port groups defined in the LFB classes of the LFB instances identified by the FromLFBID and ToLFBID. The FromPortIndex and ToPortIndex fields select the elements from the port groups that this link connects. All links are uniquely identified by the FromLFBID, FromPortGroup, and FromPortIndex fields. Multiple links may have the same ToLFBID, ToPortGroup, and ToPortIndex as this model supports fan in of inter-LFB links but not fan out.

5.2.2.4. FEConfiguredNeighbors an FEConfiguredNeighbor

The FEConfiguredNeighbors element is a wrapper to hold the configuration information that one or more FEConfiguredNeighbor elements convey about the configured FE topology.

The FEConfiguredNeighbor element occurs once for each configured FE neighbor the FE knows about. It should not be filled in based on FE level protocol operations. In general, neighbor discovery operation on the FE should be represented and manipulated as an LFB. However, for FEs that include neighbor discovery and do not have such an LFB, it is permitted to fill in the information in this table based on such protocols.

Similarly, the MAC address information in the table is intended to be used in situations where neighbors are configured by MAC address. Resolution of network layer to MAC address information should be captured in ARP LFBs, not duplicated in this table. Note that the same neighbor may be reached through multiple interfaces

or at multiple addresses. There is no uniqueness requirement of any sort on occurrences of the FEConfiguredNeighbor element.

Information about the intended forms of exchange with a given neighbor is not captured here, only the adjacency information is included.

5.2.2.4.1.NeighborID

This is the ID in some space meaningful to the CE for the neighbor. If this table remains, we probably should add an FEID from the same space as an attribute of the FE.

5.2.2.4.2.NeighborInterface

This identifies the interface through which the neighbor is reached.

[Editors note: As the port structures become better defined, the type for this should be filled in with the types necessary to reference the various possible neighbor interfaces, include physical interfaces, logical tunnels, virtual circuits, etc.]

5.2.2.4.3. NeighborNetworkAddress

Neighbor configuration is frequently done on the basis of a network layer address. For neighbors configured in that fashion, this is where that address is stored.

5.2.2.4.4.NeighborMacAddress

Neighbors are sometimes configured using MAC level addresses (Ethernet MAC address, circuit identifiers, etc.) If such addresses are used to configure the adjacency, then that information is stored here. Note that over some ports such as physical point to point links or virtual circuits considered as individual interfaces, there is no need for either form of address.

5.3. Sample FE Attribute Document

```
<?xml version="1.0">
<fm:FEDocument xmlns:fm="http://www.ietf.org/...theschema...">

 <fm:FECapabilities>

  <fm:ModifiableLFBTopology> true </fm:ModifiableLFBTopology>
```

```
<fm:SupportedLFBs>
 <fm:SupportedLFB>
  <!-- A simple single-input multi-output classifier -->
  <fm:LFBName> Classifier </fm:LFBName>
  <fm:LFBOccurrenceLimit> 3 </fm:LFBOccurrenceLimit>

  <fm:PortGroupLimits>
   <fm:PortGroupLimit>
    <!-- The input port -->
    <fm:PortGroupName> InputPortGroup </fm:PortGroupName>
    <fm:MinPortCount> 1 </fm:MinPortCount>
    <fm:MaxPortCount> 1 </fm:MaxPortCount>
   </fm:PortGroupLimit>
   <fm:PortGroupLimit>
   <!--The normal output ports -->
   <fm:PortGroupName> OutputPortGroup </fm:PortGroupName>
    <fm:MinPortCount> 0 </fm:MinPortCount>
    <fm:MaxPortCount> 32 </fm:MaxPortCount>
   </fm:PortGroupLimit>
   <fm:PortGroupLimit>
    <!-- The optional error port -->
    <fm:PortGroupName> ErrorPortGroup </fm:PortGroupName>
    <fm:MinPortCount> 0 </fm:MinPortCount>
    <fm:MaxPortCount> 1 </fm:MaxPortCount>
   </fm:PortGroupLimit>
  </fm:PortGroupLimits>
  <fm:CanOccurAfters>
   <fm:CanOccurAfter>
    <fm:NeighborLFB> Port </fm:NeighborLFB>
    <!-- omitted viaPort -->
   </fm:CanOccurAfter>
   <fm:CanOccurAfter
    <fm:NeighborLFB> InternalSource </fm:NeighborLFB>
    <!-- omitted viaPort -->
   </fm:CanOccurAfter>
  </fm:CanOccurAfters>
  <fm:CanOccurBefores>
   <fm:CanOccurBefore>
    <fm:NeighborLFB> Marker </fm:NeighborLFB>
    <!-- omitted viaPort -->
   </fm:CanOccurBefore>
  </fm:CanOccurBefores>
 </fm:SupportedLFB>
 <!-- then Supported LFB elements for Port, InternalSource -->
 <!--      Marker, ... -->
</fm:SupportedLFBs>
```

```
     <fm:SupportedAttributes>
      <fm:SupportedAttribute>
       <fm:AttributeName> FEStatus  </fm:AttributeName>
       <fm:AccessModes>  read write </fm:AccessModes>
      </fm:SupportedAttribute>
      <fm:SupportedAttribute>
       <fm:AttributeName>  Vendor   </fm:AttributeName>
       <fm:AccessModes>    read     </fm:AccessModes>
      </fm:SupportedAttribute
      <fm:SupportedAttribute>
       <fm:AttributeName>  Model    </fm:AttributeName>
       <fm:AccessModes>    read     </fm:AccessModes>
      </fm:SupportedAttribute>
     </fm:SupportedAttributes>
    </fm:FECapabilities>

    <fm:FEAttributes>
     <fm:Vendor> World Wide Widgets </fm:Vendor>
     <fm:Model> Foo Forward Model 6 </fm:Model>
     <fm:FEStatus>       OperEnable </fm:FEStatus>
     <fm:LFBInstances>
      <fm:LFBInstance>
       <fm:LFBClassName> Classifier </fm:LFBClassName>
       <fm:LFBInstanceID> Inst5     </fm:LFBInstanceID>
      </fm:LFBInstance>
      <fm:LFBInstance>
       <fm:LFBClassName> Interface  </fm:LFBClassName>
       <fm:LFBInstanceID> Inst11    </fm:LFBInstanceID>
      </fm:LFBInstance>
      <fm:LFBInstance>
       <fm:LFBClassName> Meter      </fm:LFBClassName>
       <fm:LFBInstanceID> Inst17    </fm:LFBInstanceID>
      </fm:LFBInstance>
     </fm:LFBIntances>
     <fm:LFBTopology>
      <fm:LFBLink>
       <fm:FromLFBID>   Inst11      </fm:fromLFBID>
       <fm:FromPortGroup> IFOnwardGroup </fm:FromPortGroup>
       <fm:FromPortIndex> 1         </fm:FromPortIndex>
       <fm:ToLFBID>     Inst5       </fm:ToLFBID>
       <fm:ToPortGroup> InputPortGroup </fm:ToPortGroup>
       <fm:ToPortIndex> 1           </fm:ToPortIndex>
      </fm:LFBLink>
      <fm:LFBLink>
       <fm:FromLFBID>   Inst5       </fm:fromLFBID>
       <fm:FromPortGroup> OutputGroup </fm:FromPortGroup>
       <fm:FromPortIndex> 1         </fm:FromPortIndex>
```

```
      <fm:ToLFBID>     Inst17       </fm:ToLFBID>
      <fm:ToPortGroup> InMeterGroup </fm:ToPortGroup>
      <fm:ToPortIndex> 1           </fm:ToPortIndex>
     </fm:LFBLink>
    </fm:LFBTopology>
   </fm:FEAttributes>
  </fm:FEDocument>
```

6. LFB Class Library

A set of initial LFB classes are identified here in the LFB class library as necessary to build common FE functions. Some of the LFB classes described here are abstract base classes from which specific LFB sub-classes will be derived. Hence, the base classes may not be used directly in a particular FE's model, but the sub-classes (yet to be defined) could be. This initial list attempts to describe LFB classes at the expected level of granularity. This list is neither exhaustive nor sufficiently detailed.

Several working groups in the IETF have already done some relevant work in modeling the provisioning policy data for some of the functions we are interested in, for example, the DiffServ (Differentiated Services) PIB [4] and IPSec PIB [8]. Whenever possible, we have tried to reuse the work done elsewhere instead of reinventing the wheel.

6.1. Port LFB

A Port LFB is used to model physical I/O ports on the FE. It is both a source of data "received" by the FE and a sink of data "transmitted" by the FE. The Port LFB contains a number of static attributes, which may include, but are not limited to, the following items:

- the number of physical ports on this LFB
- physical port type
- physical port link speed (may be variable; e.g., 10/100/1000 Ethernet).

In addition, the Port LFB contains a number of configurable attributes, including:

- physical port current status (up or down)
- physical port loopback
- physical port mapping to L2 interface.

The Port LFB can be sub-classed into technology specific LFB classes, with additional static and configurable attributes. Examples of possible sub-classes include:

- Ethernet

- . Packet-over-SONET OC-N
- . ATM-over-SONET/SDN OC-N
- . T3
- . E3
- . T1
- . E1
- . CSIX-L1 switching fabric port (Fi interface)
- . CE-FE port (for Fp interface).

LFB class inheritance can be used to sub-class derived LFB classes with additional properties, such as TDM channelization.

The Port LFB "receives" (sources) and "transmits" (sinks) frames in technology specific formats (described in the respective LFB class definition but not otherwise modeled) into/out of the FE. Packets "received" from a physical port are sourced on (one of) the LFB's output port(s), while packets to be "transmitted" on a physical port are sinked on (one of) the LFB's input port(s). The Port LFB is unique among LFB classes in that packets accepted on a LFB input port are not emitted back out on an LFB output port (except in the case of physical port loopback operation).

The Port LFB transmits technology specific L2 frames to topologically adjacent LFB instances (i.e., no frame decapsulation/encapsulation is modeled in this LFB class). When transmitting a frame to an adjacent downstream LFB, the Port LFB provides two items of metadata: the frame length and the L2 interface identifier. When receiving frames from an adjacent upstream LFB, the frame is accompanied by two items of metadata: frame length and outgoing port identifier.

Statistics are not maintained by the Port LFB; statistics associated with a particular port may be maintained by an L2 interface LFB (see Section 6.2).

6.2. L2 Interface LFB

The L2 Interface LFB models an L2 protocol termination. The L2 Interface LFB performs two sets of functions: decapsulation and demultiplexing as needed on the receive side of an FE, and encapsulation and multiplexing as needed on the transmit side. Hence the LFB has two distinct sets of inputs and outputs tailored for these separate functions. The L2 Interface LFB is not modeled as two separate (receive/transmit) LFBs because there are shared attributes between the decapsulation and encapsulation functions.

On the decapsulation input(s), the LFB accepts an L2 protocol specific frame, along with frame length and L2 interface metadata. The LFB decapsulates the L2 frame by removing any L2 header/trailers (while simultaneously applying any checksum/CRC functions), determines the L2 or L3 protocol type of the next-layer packet (based on a PID or Ethertype within the L2 frame header), adjusts the frame length metadata, and uses the L2 interface metadata to select an L2 interface attribute. The L2 interface attribute supports a number of additional attributes, including:

- L2 MTU
- supported next-layer L2 or L3 protocols
- L2-specific receive counters (byte, packet)
- counting mode
- L2 or L3 interface metadata for next-layer packet
- LFB output port.

The LFB may support multiple decapsulation output ports within two output groups; one for normal forwarding, and one for exception packets. The LFB emits the decapsulated packet along with the modified frame length metadata, an L2 or L3 protocol type metadata, and an L2 or L3 interface metadata.

On the encapsulation input(s), the LFB accepts a packet along with frame length, protocol type, and L2 interface metadata. The L2 interface metadata is used to select an L2 interface attribute which supports a number of additional attributes, including:

- L2-specific transmit counters (byte, packet)
- counting mode (may be taken from receive counters mode)
- L2 or L3 interface metadata for next-layer frame (we assume that L2
- protocols could be layered on top of an L3 protocol; e.g., L2TP or
- PWE3), or port metadata.
- LFB output port.

The LFB encapsulates the packet using the appropriate L2 header/trailer and protocol type information (calculating checksums/CRCs as necessary), and provides the frame to the next LFB along with incremented frame length metadata, updated protocol type metadata, and updated interface (or port) metadata, on a configurable LFB encapsulation output.

As in the case of the Port LFB, technology specific variants of the L2 interface LFB will be sub-classes of the L2 Interface LFB. Example sub-classes include:

- Ethernet/802.1Q
- PPP
- ATM AAL5.

Each sub-class will likely support static and configurable attributes specific to the L2 technology; for example the Ethernet/802.1Q Interface LFB will support a per-interface MAC address attribute. Note that each technology specific sub-class may require additional metadata. For example, the Ethernet/802.1Q Interface LFB requires an outgoing MAC destination address to generate an outgoing Ethernet header.

The L2 interface management function is separated into a distinct LFB from the Port LFB because L2 encapsulations can be nested within frames; e.g., PPP-over-Ethernet-over-ATM AAL5 (PPPoEoA).

6.3. IP interface LFB

The IP Interface LFB models a container for IP interface-specific attributes. These may include:

- IP protocols supported (IPv4 and/or IPv6)
- IP MTU
- interface MIB counters
- table metadata for associated forwarding tables (LPM, multicast)
- table metadata for associated classification tables.

The IP Interface LFB also performs basic protocol-specific packet heade validation functions (e.g., IP version, IPv4 header length, IPv4 header checksum, MTU, TTL=0, etc.). The IP Interface LFB class supports three different L3 protocols: IPv4, IPv6, and MPLS, although individual LFB instances might support a subset of these protocols, configurable on each interface attribute.

As with the L2 Interface LFB, the IP Interface LFB supports two modes of operation: one needed on the receive side of an FE, and one on the transmit side, using separate sets of LFB inputs and outputs. In the first mode of operation (for FE receive processing), the IP Interface LFB accepts IP packets along with frame length, L3 protocol type, and interface metadata (possibly including additional metadata items such as L2-derived class metadata). The interface metadata is used to select an interface attribute, and the protocol type is checked against the protocols supported for this interface. Error checks are applied, including whether the particular protocol type is supported on this interface, and if no errors occur, the appropriate counters are incremented and the protocol type is used to select the outgoing LFB output from a set dedicated to the first mode of operation. The IP header protocol type/next header field may also be used to select an LFB output; for example, IPv4 packets with AH header may be directed to a particular next LFB, or IPv6 packets with Hop-by-Hop Options. If errors do occur, the appropriate error counters

are incremented, and the error type is used to select a specific exception LFB output.

In the second mode of operation (for FE transmit processing), the IP Interface LFB accepts an IP packet along with frame length, protocol type, and interface metadata. Again, the interface metadata is used to select an interface attribute. The interface attribute stores the outgoing L2 or IP interface (e.g., tunnel) interface metadata. The IP MTU of the outgoing interface is checked, along with the protocol type of the packet. If no errors occur, the appropriate counters are incremented, and the next level interface metadata may be used to select an IP Interface LFB output dedicated to the second mode of operation. Otherwise, the appropriate error counters are incremented, and the error type is used to select an exception output.

Because the IP Interface LFB is the repository for the interface MIB counters, two special pairs of inputs are provided for packets which have been selected to be discarded further downstream (one each for the receive and transmit counters). Packets arriving on these LFB inputs must be accompanied by frame length and L3 interface metadata. An exception output on the LFB should be connected to a dropper LFB.

6.4. Classifier LFB

The function of classification is to logically partition packets into one of N different classes, based on some sequence of one or more mathematical operations applied to the packet and its associated metadata. Various LFBs perform an intrinsic classification function. Where this function is a well-defined protocol operation, a separate LFB may be defined (e.g., IP Interface LFB, which performs header verification).

Several common applications need to classify packets using a particular mathematical operation (e.g., longest prefix match (LPM) or ternary match) against a fixed set of fields in a packet's header plus metadata, or an easily recognized part of the packet payload. Two example applications are classification for Differentiated Services or for security processing. Typically the packet is evaluated against a potentially large set of rules (called "filters") which are processed in a particular order to ensure a deterministic result. This sort of classification functionalit is modeled by the Classifier LFB.

The Classifier LFB accepts an input packet and metadata, and produces the unmodified packet along with a class metadata, which may be used to map the packet to a particular LFB output.

The Classifier LFB supports multiple classifier attributes. Each classifier is parameterized by one or more filters. Classification is performed by selecting the classifier to use on a particular packet (e.g., by metadata lookup on a configurable metadata item), and by evaluating the selected contents of the accepted packet against that classifier's filters. A filter decides if the input packet satisfies particular criteria. According to [DiffServ], "a filter consists of a set of conditions on the component values of a packet's classification key (the header values, contents, and attributes relevant for classification)".

Note that other LFBs may perform simple classification on the packet or its metadata. The purpose of the Classifier LFB is to model an LFB that "digests" large amounts of input data (packet, metadata), to produce a "summary" of the classification results, in the form of additional (or modified) metadata. Other LFBs can then use this summary information to quickly and simply perform trivial classification operations.

The Classifier LFB can be sub-classed into several function-specific LFB classes which perform common classification functions. These may include:

- Longest Prefix Match (LPM)
- IP Multicast lookup (S,G)
- Multifield Exact Match
- Multifield Ternary Match.

6.5. Next Hop LFB

The Next Hop LFB is used to resolve next hop information following a forwarding lookup. Next Hop information normally includes the outgoing interface (or interfaces, in the case of multicast), as well as the outgoing IP address(es). This next hop information associated with a forwarding prefix or classification rule is often separated into a separate data structure in implementations to allow the two pieces of information to be decoupled, because there is frequently a fan-in relationship between forwarding prefix/rule entries and next hop information, and decoupling them can permit more efficient data structure management.

The Next Hop LFB maintains next hop attributes organized into multiple next hop tables. The relevant table for a packet is

selected based on next hop table metadata. A set of one or more next hop attributes is selected based on next hop index metadata. Each next hop attribute stores the following information:

- . a list of one or more outgoing interfaces
- . next hop IP addresses, or, an index to a table of this information
- . that is maintained at a downstream LFB
- . a list of outgoing MTUs
- . TTL decrement value

The Next Hop LFB has two primary operations. The first is to map the incoming next hop table and next hop index metadata into a configurable next hop attribute. This mapping may be direct (one metadata pair to one next hop attribute). If the next hop index metadata selects a set of next hop attributes, final attribute resolution depends on a selection algorithm that uses some additional metadata, or an internal classification operation, to select among a set of possible next hop attributes. One example is weighted next hop selection, where individual packets are mapped to particular next hop attributes in the set according to weights and to some flow order-preserving function (e.g., such as an address pair hash). Another alternative is class-based next hop selection, based on some class metadata.

The second operation is a derivative of the first. The next hop table and next hop index metadata are used to select a set of one or more next hop attributes. Then the outgoing interface values stored in those attributes are compared against the incoming interface metadata provided to the Next Hop LFB, to determine whether the incoming interface is in the set. This operation, in combination with a IP source address forwarding lookup (which provides the next hop table/index metadata), can be used to perform a reverse path forwarding (RPF) check.

The Next Hop LFB has two inputs: one for normal next hop resolution, and one for the incoming interface metadata test (e.g., RPF). The LFB requires incoming interface, frame length, next hop table, and next hop index metadata. There are two normal output groups, one for the normal next hop resolution, and another for the RPF check. No additional metadata is produced for the latter, but for the former, the following metadata may be produced:

- . outgoing interface(s)
- . next hop IP address(es)
- . TTL decrement value (if TTL decrement is not performed by the Next Hop LFB)

An alternative mode of operation produces index metadata instead of outgoing interface and next hop IP address metadata. This index

metadata is used to access a cache of the outgoing interface and next hop IP address that may be stored on the egress FE (this permits more efficient communication across the Fi interface). This index metadata can also be used as input metadata to a MPLS Encapsulation LFB.

The Next Hop LFB supports an exception output port group. Exception conditions include:

- RPF test failed
- No route to host
- No route to network
- Packet too big
- TTL expired

The mapping between exception conditions and exception outputs is configurable, and an exception code metadata is produced on these outputs.

6.6. Rate Meter LFB

The Rate Meter LFB is used to meter the packet flow through the LFB according to a rate- and time-dependent function. Packets are provided to the Rate Meter LFB along with packet length metadata (and optional color metadata) and are associated with a meter attribute either statically (based on LFB input) or via some other configurable metadata item. The metering algorithm of the associated meter attribute is applied to the packet, using the packet length and the current time as inputs, along with previous state maintained by the attribute. A color metadata is associated with the packet in accordance with the metering algorithm used. The color metadata is optionally emitted with the packet, or used to map the packet to a particular LFB output. Color-aware metering algorithms use color metadata if provided with the packet (e.g., by a Classifier LFB), or assume a default color value.

The Rate Meter LFB supports a number of static attributes, including:

- supported metering algorithms
- maximum number of meter attributes.

The Rate Meter LFB supports a number of configurable attributes, including:

- number of LFB inputs
- number of LFB outputs
- mapping of LFB input to meter attribute (when mapped statically)
- metadata item to select for mapping to meter attribute
- mapping of metadata value to meter attribute

. default meter attribute (when not mapped statically or via correct
. metadata)
. per-attribute metering algorithm
. per-attribute metering paramters, including:
. minimum rate
. maximum rate
. burst size
. color metadata enable
. mapping of packet color to LFB output.

A Rate Meter LFB can be used to implement a policing function, by connecting a LFB output directly to a Dropper LFB, and mapping non-conforming (e.g., "red") traffic to that output.

6.7. Redirector (de-MUX) LFB

The Redirector LFB is used to select between alternative datapaths based on the value of some metadata item. The Redirector LFB accepts an input packet P, and uses associated metadata item M to demultiplex that packet onto one of N outputs; e.g., unicast forwarding, multicast, or broadcast. Configurable attributes include:

. number of LFB output ports (N)
. metadata item to demultiplex on (M)
. mapping of metadata value to output port
. default output port (for un-matched input metadata values).

Note that other LFBs may include demultiplexing functionality (i.e., if they have multiple outputs in an output group). The Redirector LFB is especially useful for demultiplexing based on metadata items that are not generated or modified by an immediate upstream LFB.

6.8. Packet Header Rewriter LFB

The Packet Header Rewriter LFB is used to re-write fields in a packet's header. Function-specific sub-classes of the Packet Header Rewriter LFB may be specified as sub-classes of the Modifier LFB. These may include:

. IPv4 TTL/IPv6 Hop Count
. IPv4 header checksum
. DSCP
. IPv4 NAT

The precise means by which the packet header rewriting functions will be specified is TBD.

6.9. Counter LFB

The Counter LFB is used to maintain packet and/or byte statistics on the packet flow through the LFB. Packets are provided to the Counter LFB on an LFB input along with packet length metadata and are associated with a count attribute either statically (based on the LFB input) or via some other configurable metadata item. The Counter LFB modifies neither the packet nor any associated metadata.

The Counter LFB supports a number of static attributes, including:

- supported counting modes (e.g., byte, packet, both)
- supported logging modes (e.g., last recorded packet time)
- maximum number of count attributes

The Counter LFB supports a number of configurable attributes, including:

- number of LFB inputs
- mapping of LFB input to count attribute (when mapped statically)
- metadata item to select for mapping to count attribute
- mapping of metadata value to count attribute
- default count attribute (when not mapped statically or via correct
- metadata)
- counting mode per-attribute
- logging mode per-attribute.

The Counter LFB does not perform any time-dependent counting. The time at which a count is made may, however, be logged as part of the count attribute.

Other LFBs may maintain internal statistics (e.g., interface LFBs). The Counter LFB is especially useful for maintain counts associated with QoS policy.

6.10. Dropper LFB

A Dropper LFB has one input, and no outputs. It discards all packets that it accepts without any modification or examination of those packets.

The purpose of a Dropper LFB is to allow the description of "sinks" within the model, where those sinks do not result in the packet being sent into any object external to the model.

The Dropper LFB has no configurable attributes.

6.11. IPv4 Fragmenter LFB

The IPv4 Fragmenter LFB fragments IPv4 packets according to the MTU of the outgoing interface. The IPv4 Fragmenter LFB accepts packets with frame length and MTU metadata, and produces a sequence of one or more valid IPv4 packets properly fragmented, each along with corrected frame length metadata.

The source of the outgoing interface MTU is TBD. The IPv4 fragmentation function is not incorporated into the IP Interface LFB because forwarding implementations may include additional forwarding functions between fragmentation and final output interface processing.

6.12. L2 Address Resolution LFB

The L2 Address Resolution LFB is used to map an next hop IP address into an L2 address. The LFB accepts packets with output L2 interface and next hop IP address metadata, and produces the packet along with the correct L2 destination address. The L2 Address Resolution LFB maintains multiple address resolution table attributes accessed by the output L2 interface metadata. Each table attribute maintains a set of configurable L2 address attributes, accessed by the next hop IP address.

The L2 Address Resolution LFB has a normal output group which produces the L2 destination address metadata, as well as an exception output. This exception output can be used to divert the packet to another LFB (e.g., an ARP/ND Protocol LFB, or a Port LFB used to reach the CE) for address resolution.

6.13. Queue LFB

The Queue LFB is used to represent queueing points in the packet datapath. It is always used in combination with one or more Scheduler LFBs. The Queue LFB manages one or more FIFO packet queues as configurable attributes. The Queue LFB provides one or more LFB inputs, and packets are mapped from LFB inputs to queues, either statically, or via queue metadata. Each queue attribute is mapped one-to-one with a scheduling input on a downstream Scheduler LFB. The Queue LFB provides one or more LFB outputs, along with optional scheduling input metadata.

Additional per-queue configurable attributes include the following:

- . maximum depth discard behavior (tail drop/head drop/Active Queue Management (AQM))
- . AQM parameters (specific to the AQM algorithm; e.g., RED)
- . Explicit Congestion Notification (ECN) enable.

Packets are provided to the Queue LFB along with a packet length metadata and an optional queue metadata. Because the Queue LFB can model sophisticated AQM mechanisms such as per-color marking thresholds (e.g., Weighted RED), packets may also be accompanied with color metadata.

If ECN is enabled on a queue serving IP packets, then the IP packet header is modified if congestion is marked. A protocol type metadata must accompany the packet to indicate the packet protocol (e.g., IPv4, IPv6, Ethernet), so that the implementation can determine the location of the ECN bits in the header [RFC3168]. In the case of IPv4, if congestion is signaled, the header checksum must be modified. The Queue LFB supports a capability to indicate whether it corrects the IPv4 header checksum after marking congestion experienced. Support for the checksum fixup is not mandatory since the checksum may be recalculated in another LFB further downstream.

6.14. Scheduler LFB

The Scheduler LFB is used to perform packet scheduling at queueing points in the packet datapath, and hence is always used in combination with one or more upstream Queue or Scheduler LFBs. The Scheduler LFB supports one or more logical scheduling inputs. A scheduling input can be mapped one-to-one to a Scheduler LFB input, or the scheduling input can be selected via metadata (and both mechanisms may be used in combination).

The Scheduler LFB multiplexes its scheduling inputs onto a single LFB output, based on its scheduling algorithm along with the per-input scheduling configuration. The packet is not modified during the scheduling process.

Packets are provided to the Scheduler LFB along with a packet length metadata and an optional scheduling input metadata.

Configurable attributes include:

- . number of logical scheduler inputs
- . number of LFB inputs
- . mapping of LFB input to scheduler input
- . scheduling algorithm
- . per-input scheduling parameters, including:

- . priority
- . minimum service rate
- . maximum service rate
- . burst duration (at maximum service rate).

Hierarchical scheduling configurations can be created by cascading two or more Scheduler LFBs.

6.15. MPLS ILM/Decapsulation LFB

The MPLS Incoming Label Map (ILM)/Decapsulation LFB accepts MPLS-encapsulated packets, examines (and possibly removes) the top-most label, and emits the packet on one output within an output group, along with configurable index and class metadata. The configurable metadata can be used as input for an IP Interface LFB, a Next Hop LFB, or the same (or another) MPLS ILM/Decapsulation LFB. This allows the FE to terminate, forward, or "pop and lookup" on the value of the top-most label. The LFB maintains a set of ILM table attributes indexed by incoming IP interface metadata. Each ILM table entry is an attribute specifying whether to remove the label, and which output port to emit the packet on. An exception output is provided for packets with expired TTL.

6.16. MPLS Encapsulation LFB

The MPLS Encapsulation LFB accepts IP or MPLS-encapsulated packets and appends an MPLS label stack, which is selected by output interface and configurable index metadata. The TTL of the accepted packet is copied from the outermost header into the labels in the label stack, and the S bit is set on the bottom label if the accepted packet is IP. The MPLS EXP bits are copied (or mapped) according to per-stack attributes.

The MPLS Encapsulation LFB maintains multiple stack table attributes indexed by output interface metadata. Entry attributes within a table are indexed by configurable index metadata. Each entry attribute maintains a label stack, along with a configurable attribute for EXP bit handling, and possibly class and/or queue metadata to emit with the packet.

MPLS ILM/decapsulation and encapsulation functions are modeled in separate LFBs because some implementations split these operations across FEs.

6.17. Tunnel Encapsulation/Decapsulation LFB

The Tunnel Encapsulation/Decapsulation LFB models tunnel header encapsulation and decapsulation/demultiplexing. The LFB maintains separate encapsulation and decapsulation input and output groups. The encapsulation input group accepts packets with tunnel metadata, appends a tunnel header that is stored in a configurable attribute indexed by the tunnel metadata, and emits the packet on an encapsulation output. The decapsulation input group accepts packets encapsulated with a tunnel header along with tunnel metadata, removes the tunnel header (performing any tunnel-protocol-specific classification) according to attributes configured on a per-tunnel basis and accessed via the tunnel metadata, and emits the packet along with configurable metadata. For example, the configurable metadata that is output may be used as input interface metadata by a downstream IP or L2 Interface LFB. A decapsulation exception output is available and is used in the event that decapsulation fails.

The Tunnel Encapsulation/Decapsulation LFB may be sub-classed into tunnel-protocol-specific LFBs, including:

- IP-IP
- GRE
- L2TP
- Generic IPv6 Tunnels

6.18. Replicator LFB

The Replicator LFB is used to replicate accepted packets and emit them on one or more outputs in an output group. Packets are accepted along with replicator index metadata. The LFB maintains an attribute table indexed by this metadata. Each table entry attribute specifies the number of times the packet must be replicated, the outputs (within the output group) that each replicated packet should be emitted on, and configurable metadata to be associated with each replicated packet.

The Replicator LFB can be used for multicast replication, or for transparent packet interception.

7. Satisfying the Requirements on FE Model

This section describes how the proposed FE model meets the requirements outlined in Section 5 of RFC 3654 [1]. The requirements can be separated into general requirements (Sections 5, 5.1 - 5.4) and the specification of the minimal set of logical functions that the FE model must support (Section 5.5).

The general requirement on the FE model is that it be able to express the logical packet processing capability of the FE, through both a capability and a state model. In addition, the FE model is expected to allow flexible implementations and be extensible to allow defining new logical functions.

A major component of the proposed FE model is the Logical Function Block (LFB) model. Each distinct logical function in an FE is modeled as an LFB. Operational parameters of the LFB that must be visible to the CE are conceptualized as LFB attributes. These attributes support flexible implementations by allowing an FE to specify supported optional features and to indicate which attributes are configurable by the CE for an LFB class (e.g., express the capability of the FE). Configurable attributes also provide the CE some flexibility in specifying the behavior of a LFB. When multiple LFBs belonging to the same LFB class are instantiated on an FE, each of those LFBs could be configured with different attribute settings. By querying the settings of the attributes for an instantiated LFB, one can determine the state of that LFB.

Instantiated LFBs are interconnected in a directed graph that describes the ordering of the functions within an FE. This directed graph is described by the topology model. The combination of the attributes of the instantiated LFBs and the topology describe the packet processing functions available on the FE (current state).

Another key component of the FE model is the FE attributes. The FE attributes are used mainly to describe the capabilities of the FE, but they also convey information about the FE state.

The FE model also includes a definition of the minimal set of LFBs that is required by Section 5.5 of [1]. The sections that follow provide more detail on the specifics of each of those LFBs.

7.1. Port Functions

The FE model can be used to define a Port LFB class and its technology-specific subclasses (see Section 6.1) to map the physical port of the device to the LFB model with both static and configurable attributes. The static attributes model the type of port, link speed etc. The configurable attributes model the addressing, administrative status etc.

7.2. Forwarding Functions

Because forwarding function is one of the most common and important functions in the forwarding plane, it requires special attention in modeling to allow design flexibility, implementation efficiency, modeling accuracy and configuration simplicity. Toward that end, it is recommended that the core forwarding function being modeled by the combination of two LFBs -- Longest Prefix Match (LPM) classifier LFB (see Section 6.4) and Next Hop LFB (see Section 6.5). Special header writer LFB (see Section 6.8) is also needed to take care of TTL decrement and checksum etc.

7.3. QoS Functions

The LFB class library already includes descriptions of the Meter (Section 6.6.), Queue (Section 6.13), Scheduler (Section 6.14), Counter (Section 6.9) and Dropper (Section 6.10) LFBs to support the QoS functions in the forwarding path. FE model can also be used to define other useful QoS functions as needed. These LFBs allow the CE to manipulate the attributes to model IntServ or DiffServ functions.

7.4. Generic Filtering Functions

Various combinations of Classifier (Section 6.4), Redirector (Section 6.7), Meter (Section 6.6.) and Dropper (Section 6.10) LFBs can model a complex set of filtering functions.

7.5. Vendor Specific Functions

New LFB classes can be defined according to the LFB model as described in Section 4 to support vendor specific functions. A new LFB class can also be derived from an existing LFB class through inheritance.

7.6.High-Touch Functions

High-touch functions are those that take action on the contents or headers of a packet based on content other than what is found in the IP header. Examples of such functions include NAT, ALG, firewall, tunneling and L7 content recognition. It is not practical to include all possible high touch functions in the initial LFB library in Section 6 due to the number and complexity. However, the flexibility of the LFB model and the power of interconnection in LFB topology should make it possible to model any high-touch functions.

7.7. Security Functions

Security functions are not included in the initial LFB class library. However, the FE model is flexible and powerful enough to model the types of encryption and/or decryption functions that an FE supports and the associated attributes for such functions.

The IP Security Policy (IPSP) Working Group in the IETF has started work in defining the IPSec Policy Information Base [8]. We should try to reuse the work as much as we can.

7.8. Off-loaded Functions

In addition to the packet processing functions that are typical to find on the FEs, some logical functions may also be executed asynchronously by some FEs, according to a certain finite-state machine, triggered not only by packet events, but by timer events as well. Examples of such functions include finite-state machine execution required by TCP termination or OSPF Hello processing off-loaded from the CE. By defining LFBs for such functions, the FE model is capable of expressing these asynchronous functions, so that the CE may take advantage of such off-loaded functions on the FEs.

7.9. IPFLOW/PSAMP Functions

[9] defines architecture for IP traffic flow monitoring, measuring and exporting. The LFB model supports statistics collection on the LFB by including statistical attributes (Section 4.7.4) in the LFB class definitions; in addition, special statistics collection LFBs such as meter LFB (Section 7.2.2) and counter LFB (Section 7.2.1) can also be used to support accounting functions in the FE.

[10] describes a framework to define a standard set of capabilities for network elements to sample subsets of packets by statistical and other methods. Time event generation, filter LFB, and counter/meter LFB are the elements needed to support packet filtering and sampling functions -- these elements can all be supported in the FE model.

8. Using the FE model in the ForCES Protocol

The actual model of the forwarding plane in a given NE is something the CE must learn and control by communicating with the FEs (or by other means). Most of this communication will happen in the post-association phase using the ForCES protocol. The following types of information must be exchanged between CEs and FEs via the ForCES protocol:

1) FE topology query;

2) FE capability declaration;
3) LFB topology (per FE) and configuration capabilities query;
4) LFB capability declaration;
5) State query of LFB attributes;
6) Manipulation of LFB attributes;
7) LFB topology reconfiguration.

Items 1) through 5) are query exchanges, where the main flow of information is from the FEs to the CEs. Items 1) through 4) are typically queried by the CE(s) in the beginning of the post-association (PA) phase, though they may be repeatedly queried at any time in the PA phase. Item 5) (state query) will be used at the beginning of the PA phase, and often frequently during the PA phase (especially for the query of statistical counters).

Items 6) and 7) are "command" types of exchanges, where the main flow of information is from the CEs to the FEs. Messages in Item 6) (the LFB re-configuration commands) are expected to be used frequently. Item 7) (LFB topology re-configuration) is needed only if dynamic LFB topologies are supported by the FEs and it is expected to be used infrequently.

Among the seven types of payload information the ForCES protocol carries between CEs and FEs, the FE model covers all of them except item 1), which concerns the inter-FE topology. The FE model focuses on the LFB and LFB topology within a single FE. Since the information related to item 1) requires global knowledge about all of the FEs and their inter-connection with each other, this exchange is part of the ForCES base protocol instead of the FE model.

The relationship between the FE model and the seven post-association messages are visualized in Figure 9:

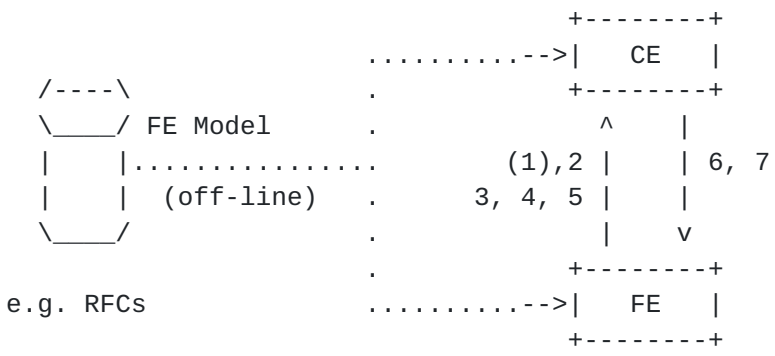


Figure 9. Relationship between the FE model and the ForCES protocol messages, where (1) is part of the ForCES base protocol, and the rest are defined by the FE model.

The actual encoding of these messages is defined by the ForCES protocol and beyond the scope of the FE model. Their discussion is nevertheless important here for the following reasons:

. These PA model components have considerable impact on the FE model. For example, some of the above information can be represented as attributes of the LFBs, in which case such attributes must be defined in the LFB classes.

. The understanding of the type of information that must be exchanged between the FEs and CEs can help to select the appropriate protocol format and the actual encoding method (such as XML, TLVs).

. Understanding the frequency of these types of messages should influence the selection of the protocol format (efficiency considerations).

An important part of the FE model is the port the FE uses for its message exchanges to and from the CE. In the case that a dedicated port is used for CE-FE communication, we propose to use a special port LFB, called the CE-FE Port LFB (a subclass of the general Port LFB in Section 6.1), to model this dedicated CE-FE port. The CE-FE Port LFB acts as both a source and sink for the traffic from and to the CE. Sometimes the CE-FE traffic does not have its own dedicated port, instead the data fabric is shared for the data plane traffic and the CE-FE traffic. A special processing LFB can be used to model the ForCES packet encapsulation and decapsulation in such cases.

The remaining sub-sections of this section address each of the seven message types.

8.1. FE Topology Query

An FE may contain zero, one or more external ingress ports. Similarly, an FE may contain zero, one or more external egress ports. In other words, not every FE has to contain any external ingress or egress interfaces. For example, Figure 10 shows two cascading FEs. FE #1 contains one external ingress interface but no external egress interface, while FE #2 contains one external egress interface but no ingress interface. It is possible to connect these two FEs together via their internal interfaces to achieve the complete ingress-to-egress packet processing function. This provides the flexibility to spread the functions across multiple FEs and interconnect them together later for certain applications.

While the inter-FE communication protocol is out of scope for ForCES, it is up to the CE to query and understand how multiple FEs are inter-connected to perform a complete ingress-egress packet processing function, such as the one described in Figure 10. The inter-FE topology information may be provided by FEs, may be hard-coded into CE, or may be provided by some other entity (e.g., a bus manager) independent of the FEs. So while the ForCES protocol supports FE topology query from FEs, it is optional for the CE to use it, assuming the CE has other means to gather such topology information.

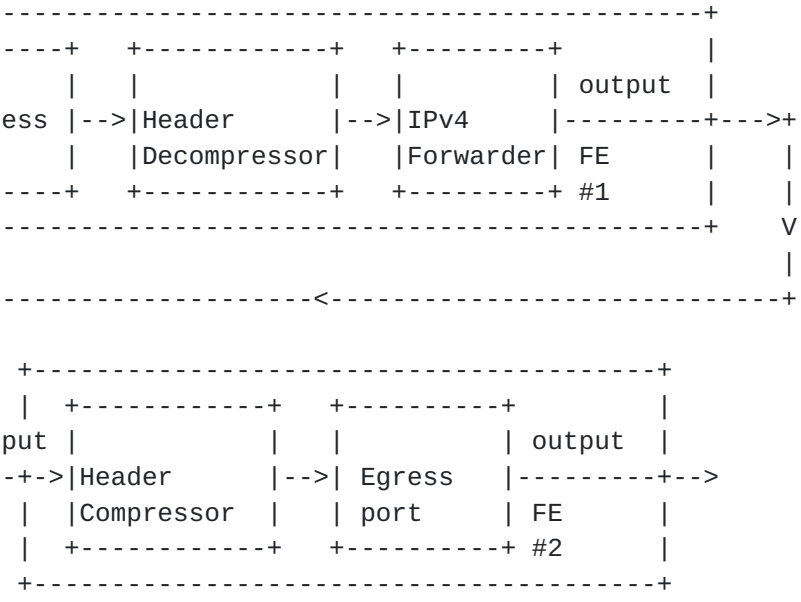


Figure 10. An example of two FEs connected together.

Once the inter-FE topology is discovered by the CE after this query, it is assumed that the inter-FE topology remains static. However, it is possible that an FE may go down during the NE operation, or a board may be inserted and a new FE activated, so the inter-FE topology will be affected. It is up to the ForCES protocol to provide a mechanism for the CE to detect such events and deal with the change in FE topology. FE topology is outside the scope of the FE model.

8.2. FE Capability Declarations

FEs will have many types of limitations. Some of the limitations must be expressed to the CEs as part of the capability model. The CEs must be able to query these capabilities on a per-FE basis. Examples:

- Metadata passing capabilities of the FE. Understanding these capabilities will help the CE to evaluate the feasibility of LFB topologies, and hence to determine the availability of certain services.
- Global resource query limitations (applicable to all LFBs of the FE).
- LFB supported by the FE.
- LFB class instantiation limit.
- LFB topological limitations (linkage constraint, ordering etc.)

8.3. LFB Topology and Topology Configurability Query

The ForCES protocol must provide the means for the CEs to discover the current set of LFB instances in an FE and the interconnections between the LFBs within the FE. In addition, sufficient information should be available to determine whether the FE supports any CE-initiated (dynamic) changes to the LFB topology, and if so, determine the allowed topologies. Topology configurability can also be considered as part of the FE capability query as described in Section 9.3.

8.4. LFB Capability Declarations

LFB class specifications define a generic set of capabilities. When an LFB instance is implemented (instantiated) on a vendor's FE, some additional limitations may be introduced. Note that we discuss only those limitations that are within the flexibility of the LFB class specification. That is, the LFB instance will remain compliant with the LFB class specification despite these limitations. For example, certain features of an LFB class may be optional, in which case it must be possible for the CE to determine

if an optional feature is supported by a given LFB instance or not. Also, the LFB class definitions will probably contain very few quantitative limits (e.g., size of tables), since these limits are typically imposed by the implementation. Therefore, quantitative limitations should always be expressed by capability arguments.

LFB instances in the model of a particular FE implementation will possess limitations on the capabilities defined in the corresponding LFB class. The LFB class specifications must define a set of capability arguments, and the CE must be able to query the actual capabilities of the LFB instance via querying the value of such arguments. The capability query will typically happen when the LFB is first detected by the CE. Capabilities need not be re-queried in case of static limitations. In some cases, however, some capabilities may change in time (e.g., as a result of adding/removing other LFBs, or configuring certain attributes of some other LFB when the LFBs share physical resources), in which case additional mechanisms must be implemented to inform the CE about the changes.

The following two broad types of limitations will exist:

- Qualitative restrictions. For example, a standardized multi-field classifier LFB class may define a large number of classification fields, but a given FE may support only a subset of those fields.
- Quantitative restrictions, such as the maximum size of tables, etc.

The capability parameters that can be queried on a given LFB class will be part of the LFB class specification. The capability parameters should be regarded as special attributes of the LFB. The actual values of these arguments may be, therefore, obtained using the same attribute query mechanisms as used for other LFB attributes.

Capability attributes will typically be read-only arguments, but in certain cases they may be configurable. For example, the size of a lookup table may be limited by the hardware (read-only), in other cases it may be configurable (read-write, within some hard limits).

Assuming that capabilities will not change frequently, the efficiency of the protocol/schema/encoding is of secondary concern.

8.5. State Query of LFB Attributes

This feature must be provided by all FEs. The ForCES protocol and the data schema/encoding conveyed by the protocol must together

satisfy the following requirements to facilitate state query of the LFB attributes:

- Must permit FE selection. This is primarily to refer to a single FE, but referring to a group of (or all) FEs may optional be supported.
- Must permit LFB instance selection. This is primarily to refer to a single LFB instance of an FE, but optionally addressing of a group of LFBs (or all) may be supported.
- Must support addressing of individual attribute of an LFB.
- Must provide efficient encoding and decoding of the addressing info and the configured data.
- Must provide efficient data transmission of the attribute state over the wire (to minimize communication load on the CE-FE link).

8.6. LFB Attribute Manipulation

This is a place-holder for all operations that the CE will use to populate, manipulate, and delete attributes of the LFB instances on the FEs. This is how the CE configures an individual LFB instance.

The same set of requirements as described in Section 9.5 for attribute query applies here for attribute manipulation as well.

Support for various levels of feedback from the FE to the CE (e.g., request received, configuration completed), as well as multi-attribute configuration transactions with atomic commit and rollback, may be necessary in some circumstances.

(Editor's note: It remains an open issue as to whether or not other methods are needed in addition to "get attribute" and "set attribute" (such as multi-attribute transactions). If the answer to that question is yes, it is not clear whether such methods should be supported by the FE model itself or the ForCES protocol.)

8.7. LFB Topology Re-configuration

Operations that will be needed to reconfigure LFB topology:

- Create a new instance of a given LFB class on a given FE.
- Connect a given output of LFB x to the given input of LFB y.
- Disconnect: remove a link between a given output of an LFB and a given input of another LFB.
- Delete a given LFB (automatically removing all interconnects to/from the LFB).

9. Acknowledgments

Many of the colleagues in our companies and participants in the ForCES mailing list have provided invaluable input into this work.

10. Security Considerations

The FE model describes the representation and organization of data sets and attributes in the FEs. ForCES framework document [2] provides a comprehensive security analysis for the overall ForCES architecture. For example, the ForCES protocol entities must be authenticated per the ForCES requirements before they can access the information elements described in this document via ForCES. The access to the information contained in the FE model is accomplished via the ForCES protocol which will be defined in separate documents and so the security issues will be addressed there.

11. Normative References

[1] Khosravi, H. et al., "Requirements for Separation of IP Control and Forwarding", RFC 3654, November 2003.

[2] Yang, L. et al., "Forwarding and Control Element Separation (ForCES) Framework", work in progress, November 2003, <draft-ietf-forces-framework-13.txt>.

12. Informative References

[3] Bernet, Y. et al., "An Informal Management Model for Diffserv Routers", RFC 3290, May 2002.

[4] Chan, K. et al., "Differentiated Services Quality of Service Policy Information Base", RFC 3317, March 2003.

[5] Sahita, R. et al., "Framework Policy Information Base", RFC 3318, March 2003.

[6] Moore, B. et al., "Information Model for Describing Network Device QoS Datapath Mechanisms", RFC 3670, January 2004.

[7] Snir, Y. et al., "Policy Framework QoS Information Model", RFC 3644, Nov 2003.

[8] Li, M. et al., "IPsec Policy Information Base", work in progress, January 2003, <draft-ietf-ipsp-ipsecpib-07.txt>.

[9] Quittek, J. et Al., "Requirements for IP Flow Information Export", work in progress, January 2004, <draft-ietf-ipfix-reqs-15.txt>.

[10] Duffield, N., "A Framework for Passive Packet Measurement ", work in progress, December 2003, <draft-ietf-psamp-framework-05.txt>.

[11] Pras, A. and Schoenwaelder, J., FRC 3444 "On the Difference between Information Models and Data Models", January 2003.

13. Authors' Addresses

L. Lily Yang
Intel Corp.
Mail Stop: JF3-206
2111 NE 25th Avenue
Hillsboro, OR 97124, USA
Phone: +1 503 264 8813
Email: lily.l.yang@intel.com

Joel M. Halpern
Megisto Systems, Inc.
20251 Century Blvd.
Germantown, MD 20874-1162, USA
Phone: +1 301 444-1783
Email: jhalpern@megisto.com

Ram Gopal
Nokia Research Center
5, Wayside Road,
Burlington, MA 01803, USA
Phone: +1 781 993 3685
Email: ram.gopal@nokia.com

Alan DeKok
IDT Inc.
1575 Carling Ave.
Ottawa, ON K1G 0T3, Canada
Phone: +1 613 724 6004 ext. 231
Email: alan.dekok@idt.com

Zsolt Haraszti
Ericsson
920 Main Campus Dr, St. 500
Raleigh, NC 27606, USA
Phone: +1 919 472 9949

Email: zsolt.haraszti@ericsson.com

Steven Blake
Ericsson
920 Main Campus Dr, St. 500
Raleigh, NC 27606, USA
Phone: +1 919 472 9913
Email: steven.blake@ericsson.com

Ellen Deleganes
Intel Corp.
Mail Stop: JF3-206
2111 NE 25th Avenue
Hillsboro, OR 97124, USA
Phone: +1 503 712 4173
Email: ellen.m.deleganes@intel.com

14. Intellectual Property Right

The authors are not aware of any intellectual property right issues pertaining to this document.

15. IANA consideration

A namespace is needed to uniquely identify the LFB type in the LFB class library.

Frame type supported on input and output of LFB must also be uniquely identified.

A set of metadata supported by the LFB model must also be uniquely identified with names or IDs.