

Internet Engineering Task Force
Internet-Draft
Intended status: Informational
Expires: January 09, 2014

E. Haleplidis
University of Patras
July 08, 2013

ForCES Model Extension
draft-haleplidis-forces-model-extension-04

Abstract

Forwarding and Control Element Separation (ForCES) defines an architectural framework and associated protocols to standardize information exchange between the control plane and the forwarding plane in a ForCES Network Element (ForCES NE). RFC5812 has defined the ForCES Model provides a formal way to represent 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.

RFC5812 has been around for two years and experience in its use has shown room for small extensions without a need to alter the protocol while retaining backward compatibility with older xml libraries. This document extends the model to allow complex datatypes for metadata, optional default values for datatypes and optional access types for structures. The document also introduces three new features, bitmap as a new datatype, a new event condition BecomesEqualTo and LFB properties.

Status of This Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at <http://datatracker.ietf.org/drafts/current/>.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on January 09, 2014.

Copyright Notice

Copyright (c) 2013 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust's Legal Provisions Relating to IETF Documents (<http://trustee.ietf.org/license-info>) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document. Code Components extracted from this document must include Simplified BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the Simplified BSD License.

Table of Contents

1. Terminology and Conventions	2
1.1. Requirements Language	2
1.2. Definitions	2
2. Introduction	4
3. ForCES Model Extension proposal	4
3.1. Complex datatypes for Metadata	4
3.2. Optional Default Value for Datatypes	6
3.3. Optional Access Type for Structs	7
3.4. New datatype: Bitmap	8
3.5. New Event Condition: BecomesEqualTo	10
3.6. LFB Properties	10
3.7. Enhancing XML Validation	11
4. XML Extension Schema for LFB Class Library Documents	12
5. Acknowledgements	25
6. IANA Considerations	25
7. Security Considerations	25
8. References	25
8.1. Normative References	25
8.2. Informative References	26
Author's Address	26

1. Terminology and Conventions

1.1. Requirements Language

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 [RFC2119].

1.2. Definitions

This document follows the terminology defined by the ForCES Model in [RFC5812]. The required definitions are repeated below for clarity.

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 LFB modeling of individual Logical Functional Block (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 [RFC5810].

LFB (Logical Functional Block) Class (or type) - A template that represents a fine-grained, logically separable aspect of FE processing. Most LFBs relate to packet processing in the data path. LFB classes are the basic building blocks of the FE model.

LFB Instance - As a packet flows through an FE along a data path, it flows through one or multiple LFB instances, where each LFB is an instance of a specific LFB class. Multiple instances of the same LFB class can be present in an FE's data path. Note that we often refer to LFBs without distinguishing between an 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. XML is used to provide a formal definition of the necessary structures for the modeling. Four types of information are defined in the LFB model. The core part of the LFB model is the LFB class definitions; the other three types of information define constructs associated with and used by the class definition. These are reusable data types, supported frame (packet) formats, and metadata.

Element - Element is generally used in this document in accordance with the XML usage of the term. It refers to an XML tagged part of an XML document. For a precise definition, please see the full set of XML specifications from the W3C. This term is included in this list for completeness because the ForCES formal model uses XML.

Attribute - Attribute is used in the ForCES formal modeling in accordance with standard XML usage of the term, i.e., to provide attribute information included in an XML tag.

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 the per-packet state is implemented within actual hardware. Metadata is sent between the FE and the CE on redirect packets.

ForCES Component - A ForCES Component is a well-defined, uniquely identifiable and addressable ForCES model building block. A component has a 32-bit ID, name, type, and an optional synopsis description. These are often referred to simply as components.

LFB Component - An LFB component is a ForCES component that defines the Operational parameters of the LFBs that must be visible to the CEs.

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

2. Introduction

The ForCES Model [RFC5812] presents a formal way to define FEs Logical Function Blocks (LFBs) using XML. [RFC5812] has been published a more than two years and current experience in its use has demonstrated need for adding new and changing existing modeling concepts.

Specifically this document extends the ForCES Model to allow complex datatypes for metadata, optional default values for datatypes and optional access types for structures. Additionally the document introduces three new features, bitmap as a new datatype, a new event condition BecomesEqualTo and LFB properties.

These extensions are an addendum to the ForCES model [RFC5812] and do not require any changes on the ForCES protocol [RFC5810] as they are simply changes of the schema definition. Additionally backward compatibility is ensured as xml libraries produced with the earlier schema are still valid with the new one.

XXX: Discussion is needed to specify whether bitmap required protocol definition of how bitmap is sent through the wire.

3. ForCES Model Extension proposal

3.1. Complex datatypes for Metadata

Section 4.6. (Element for Metadata Definitions) in the ForCES Model [RFC5812] limits the datatype use in metadata to only atomic types. Figure 1 shows the xml schema excerpt where only typeRef and atomic are allowed for a metadata definition.

However there are cases where complex metadata are used in the datapath, for example two simple use cases can be seen in the OpenFlow switch 1.1 [OpenFlowSpec1.1] and beyond:

1. The Action Set metadata follows a packet inside the Flow Tables. The Action Set metadata is an array of actions to be performed at the end of the pipeline.
2. When a packet is received from a controller it may be accompanied by a list of actions to be performed on it prior to be sent on the flow table pipeline which is also an array.

With this extension (Figure 2), complex data types are also allowed, specifically structs and arrays as metadata. The key declarations are required to check for validity of content keys in arrays and componentIDs in structs.

```
<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 name="metadataID" type="xsd:integer"/>
          <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>
```

Figure 1: Initial MetadataDefType Defintion in the schema

```
<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 name="metadataID" type="xsd:integer"/>
          <xsd:element ref="description" minOccurs="0"/>
          <xsd:choice>
```

```

<xsd:element name="typeRef" type="typeRefNMTOKEN"/>
<xsd:element name="atomic" type="atomicType"/>
<xsd:element name="array" type="arrayType">
  <xsd:key name="contentKeyID1">
    <xsd:selector xpath="lfb:contentKey"/>
    <xsd:field xpath="@contentKeyID"/>
  </xsd:key>
</xsd:element>
<xsd:element name="struct" type="structType">
  <xsd:key name="structComponentID1">
    <xsd:selector xpath="lfb:component"/>
    <xsd:field xpath="@componentID"/>
  </xsd:key>
</xsd:element>
</xsd:choice>
</xsd:sequence>
</xsd:complexType>
</xsd:element>
</xsd:sequence>
</xsd:complexType>

```

Figure 2: New MetadataDefType Defintion for the schema

3.2. Optional Default Value for Datatypes

In the original schema, default values can only be defined for datatypes defined inside LFB components and not inside structures or arrays. Therefore default values of datatypes that are constantly being reused, e.g. counters with default value of 0, have to be constantly respecified. Additionally, datatypes inside complex datatypes cannot be defined with a default value, e.g. a counter inside a struct that has a default value of 0.

This extension allows optionally to add default values to atomic and typeRef types, whether they are as simple or complex datatypes. A simple use case would be to have a struct component where one of the components is a counter which the default value would be zero.

This extension alters the definition of the typeDeclarationGroup in the xml schema from Figure 3 to Figure 4 to allow default values to TypeRef.

```

<xsd:element name="typeRef" type="typeRefNMTOKEN"/>

```

Figure 3: Initial Excerpt of typeDeclarationGroup Defintion in the schema

```

    <xsd:sequence>
    <xsd:element name="typeRef" type="typeRefNMTOKEN"/>
    <xsd:element name="DefaultValue" type="xsd:token"
      minOccurs="0"/>
    </xsd:sequence>

```

Figure 4: New Excerpt of typeDeclarationGroup Definition in the schema

Additionally it appends to the declaration of the AtomicType this xml (Figure 5) to allow default values to Atomic datatypes.

```

<xsd:element name="defaultValue" type="xsd:token" minOccurs="0"/>

```

Figure 5: Appending xml in of AtomicType Definition in the schema

3.3. Optional Access Type for Structs

In the original schema, the access type can be only be defined on components of LFB and not on components in structs or arrays. However when it's a struct datatype it is not possible to fine-tune access type per component in the struct. A simple use case would be to have a read-write struct component where one of the components is a counter where the access-type could be read-reset or read-only, e.g. a read-reset or a read-only counter inside a struct.

With this extension is it allowed to define the access type for a struct component either in the datatype definitions or in the LFB component definitions.

When the optional access type for a struct component is defined it MUST override the access type of the struct. If by accident an access type for a component in a capability is defined, the access type MUST NOT be taken into account and MUST always be considered as read-only.

This extension alters the definition of the struct in the xml schema from Figure 6 to Figure 7.

```

<xsd:complexType name="structType">
  <xsd:sequence>
    <xsd:element name="derivedFrom" type="typeRefNMTOKEN"
      minOccurs="0"/>
    <xsd:element name="component" maxOccurs="unbounded">
      <xsd:complexType>
        <xsd:sequence>
          <xsd:element name="name" type="xsd:NMTOKEN"/>
          <xsd:element ref="synopsis"/>
          <xsd:element ref="description" minOccurs="0"/>

```

```

        <xsd:element name="optional" minOccurs="0"/>
        <xsd:group ref="typeDeclarationGroup"/>
    </xsd:sequence>
    <xsd:attribute name="componentID" type="xsd:unsignedInt"
        use="required"/>
</xsd:complexType>
</xsd:element>
</xsd:sequence>
</xsd:complexType>

```

Figure 6: Initial xml for the struct definition in the schema

```

<xsd:complexType name="structType">
  <xsd:sequence>
    <xsd:element name="derivedFrom" type="typeRefNMTOKEN"
        minOccurs="0"/>
    <xsd:element name="component" maxOccurs="unbounded">
      <xsd:complexType>
        <xsd:sequence>
          <xsd:element name="name" type="xsd:NMTOKEN"/>
          <xsd:element ref="synopsis"/>
          <xsd:element ref="description" minOccurs="0"/>
          <xsd:element name="optional" minOccurs="0"/>
          <xsd:group ref="typeDeclarationGroup"/>
        </xsd:sequence>
        <xsd:attribute name="access" use="optional"
            default="read-write">
          <xsd:simpleType>
            <xsd:list itemType="accessModeType"/>
          </xsd:simpleType>
        </xsd:attribute>
        <xsd:attribute name="componentID" type="xsd:unsignedInt"
            use="required"/>
      </xsd:complexType>
    </xsd:element>
  </xsd:sequence>
</xsd:complexType>

```

Figure 7: New xml for the struct definition in the schema

3.4. New datatype: Bitmap

With the current schema it is valid to create a struct of booleans in order to simulate a bitmap value. However each boolean is sent as 4bytes. This extension adds the bitmap, a set of sequential named bits.

Bitmaps may be useful in describing capabilities, e.g. Link speed capabilities as multiple boolean values.

XXX Discussion may be required as to whether there is a need for protocol description of how the bitmap is sent through the wire.

In the new schema, bits are named followed an optional bit value. An example:

```
<dataTypeDef>
  <name>Bitmap example</name>
  <synopsis>A bitmap field example</synopsis>
  <bitmap>
    <bit name="Bit0" defaultValue="0"/>
    <bit name="Bit1"/>
  </bitmap>
</dataTypeDef>
```

Figure 8: Example of bitmap Defintion

The ordering of the bits MUST be implemented in the order that are defined in the xml library.

The bitmap is defined in the model extension schema is as follows:

```
<xsd:complexType name="bitmapType">
  <xsd:sequence>
    <xsd:element name="bit" maxOccurs="unbounded">
      <xsd:complexType>
        <xsd:attribute name="name" type="xsd:NMTOKEN" use="required"/>
        <xsd:attribute name="defaultValue" type="booleanValues"
          use="optional"></xsd:attribute>
      </xsd:complexType>
    </xsd:element>
  </xsd:sequence>
</xsd:complexType>
<xsd:simpleType name="booleanValues">
  <xsd:restriction base="xsd:integer">
    <xsd:minInclusive value="0"></xsd:minInclusive>
    <xsd:maxInclusive value="1"></xsd:maxInclusive>
  </xsd:restriction>
</xsd:simpleType>
```

Figure 9: New Excerpt of bitmap Defintion in the schema

Along with the needed addition to the typeDeclarationGroup Definition:

```
<xsd:element name="bitmap" type="bitmapType"/>
```

Figure 10: New Excerpt of typeDeclarationGroup Definition in the schema

3.5. New Event Condition: BecomesEqualTo

This extension adds one more event condition in the model schema, that of BecomesEqualTo. The difference between Greater Than and Less Than, is that when the value is exactly that of the BecomesEqualTo, the event is triggered. This event condition is particularly useful when there is a need to monitor one or more states of an LFB or the FE. For example in the CEHA [I-D.ietf-forces-ceha] document it may be useful for the master CE to know which backup CEs have just become associated in order to connect to them and begin synchronizing the state of the FE. The master CE could always poll for such information but getting such an event will speed up the process and the event may be useful in other cases as well for monitoring state.

The event MUST be triggered only when the value of the targeted component becomes equal to the event condition value and MUST NOT generate events while the targeted component's value remains equal to the event condition's value.

The BecomesEqualTo is appended to the schema as follows:

```
<xsd:element name="eventBecomesEqualTo"
substitutionGroup="eventCondition"/>
```

Figure 11: New Excerpt of BecomesEqualTo event condition definition in the schema

3.6. LFB Properties

The current model definition specifies properties for components of LFBs. Experience however has proven valuable at least for debug reasons, to have statistics per LFB instance to monitor sent/received messages and errors for communication between CE and FE. These properties are read-only.

XXX: Discussion for addressing LFB properties. Possibly in the protocol extension?

The following datatype definitions are to be used as properties for LFB instances.

```
<datatypeDef>
  <name>LFBProperties</name>
```

```
<synopsis>LFB Properties definition</synopsis>
<struct>
  <component componentID="1">
    <name>SentToCE</name>
    <synopsis>Messages sent to CE</synopsis>
    <typeRef>uint32</typeRef>
  </component>
  <component componentID="2">
    <name>SentErrorsToCE</name>
    <synopsis>Error messages sent to CE</synopsis>
    <typeRef>uint32</typeRef>
  </component>
  <component componentID="3">
    <name>ReceivedFromCE</name>
    <synopsis>Messages received from CE</synopsis>
    <typeRef>uint32</typeRef>
  </component>
  <component componentID="4">
    <name>ReceivedErrorsFromCE</name>
    <synopsis>Error messages received from CE</synopsis>
    <typeRef>uint32</typeRef>
  </component>
</struct>
</dataTypeDef>
```

Properties for LFB instances

3.7. Enhancing XML Validation

As specified earlier this is not an extension but an enhancement of the schema to provide additional validation rules. This includes adding new key declarations to provide uniqueness as defined by the ForCES Model [RFC5812]. Such validations work only on within the same xml file.

The following validation rules have been appended in the original schema in [RFC5812]:

1. Each metadata ID must be unique.
2. LFB Class IDs must be unique.
3. Component ID, Capability ID and Event Base ID must be unique per LFB.
4. Event IDs must be unique per LFB.

5. Special Values in Atomic datatypes must be unique per atomic datatype.

4. XML Extension Schema for LFB Class Library Documents

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns="urn:ietf:params:xml:ns:forces:lfbmodel:1.0"
  xmlns:lfb="urn:ietf:params:xml:ns:forces:lfbmodel:1.0"
  targetNamespace="urn:ietf:params:xml:ns:forces:lfbmodel:1.0"
  elementFormDefault="qualified" attributeFormDefault="unqualified">
  <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="metadataDefID">
        <xsd:selector xpath="lfb:metadataDefs/lfb:metadataDef" />
        <xsd:field xpath="lfb:metadataID" />
    </xsd:key>
    <xsd:key name="LFBClassDef">
        <xsd:selector xpath="lfb:LFBClassDefs/lfb:LFBClassDef" />
        <xsd:field xpath="lfb:name" />
    </xsd:key>
    <xsd:key name="LFBClassDefID">
        <xsd:selector xpath="lfb:LFBClassDefs/lfb:LFBClassDef" />
        <xsd:field xpath="@LFBClassID" />
    </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 name="derivedFrom" type="xsd:NMTOKEN"
                        minOccurs="0" />
                    <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:sequence>
</xsd:element>
```

```

        </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], string,
      byte[N], boolean, octetstring[N], float32, float64 -->
<xsd:group name="typeDeclarationGroup">
    <xsd:choice>
        <!-- Extension -->
        <xsd:sequence>
            <!-- /Extension -->
            <xsd:element name="typeRef" type="typeRefNMTOKEN" />
            <!-- Extension -->
            <xsd:element name="DefaultValue" type="xsd:token"
                minOccurs="0" />
        </xsd:sequence>
        <xsd:element name="bitmap" type="bitmapType"/>
        <!-- /Extension -->
        <xsd:element name="atomic" type="atomicType" />
        <xsd:element name="array" type="arrayType">
            <!-- Extension -->
            <!--declare keys to have unique IDs -->
            <xsd:key name="contentKeyID">
                <xsd:selector xpath="lfb:contentKey" />
                <xsd:field xpath="@contentKeyID" />
            </xsd:key>
            <!-- /Extension -->
        </xsd:element>
        <xsd:element name="struct" type="structType">
            <!-- Extension -->
            <!-- key for componentIDs uniqueness in a struct -->
            <xsd:key name="structComponentID">
                <xsd:selector xpath="lfb:component" />
                <xsd:field xpath="@componentID" />
            </xsd:key>
            <!-- /Extension -->
        </xsd:element>
        <xsd:element name="union" type="structType" />
        <xsd:element name="alias" type="typeRefNMTOKEN" />
    </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:pattern value="octetstring\[\\d+\\]" />
    </xsd:restriction>
</xsd:simpleType>

```

```
</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">
      <!-- Extension -->
      <xsd:key name="SpecialValue">
        <xsd:selector xpath="specialValue" />
        <xsd:field xpath="@value" />
      </xsd:key>
      <!-- /Extension -->
    </xsd:element>
    <!-- Extension -->
    <xsd:element name="defaultValue" type="xsd:token"
      minOccurs="0" />
    <!-- /Extension -->
  </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>
<!-- Extension -->
```

```
<xsd:complexType name="bitmapType">
  <xsd:sequence>
    <xsd:element name="bit" maxOccurs="unbounded">
      <xsd:complexType>
        <xsd:attribute name="name" type="xsd:NMTOKEN"
          use="required"/>
        <xsd:attribute name="defaultValue" type="booleanValues"
          use="optional"/>
      </xsd:complexType>
    </xsd:element>
  </xsd:sequence>
</xsd:complexType>
<xsd:simpleType name="booleanValues">
  <xsd:restriction base="xsd:integer">
    <xsd:minInclusive value="0"/></xsd:minInclusive>
    <xsd:maxInclusive value="1"/></xsd:maxInclusive>
  </xsd:restriction>
</xsd:simpleType>
<!-- /Extension -->
<xsd:complexType name="arrayType">
  <xsd:sequence>
    <xsd:group ref="typeDeclarationGroup" />
    <xsd:element name="contentKey" minOccurs="0"
      maxOccurs="unbounded">
      <xsd:complexType>
        <xsd:sequence>
          <xsd:element name="contentKeyField"
            type="xsd:string" maxOccurs="unbounded" />
        </xsd:sequence>
        <xsd:attribute name="contentKeyID" type="xsd:integer"
          use="required" />
      </xsd:complexType>
    </xsd:element>
  </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="derivedFrom" type="typeRefNMTOKEN"
      minOccurs="0" />
    <xsd:element name="component" maxOccurs="unbounded">
      <xsd:complexType>
        <xsd:sequence>
          <xsd:element name="name" type="xsd:NMTOKEN" />
          <xsd:element ref="synopsis" />
          <xsd:element ref="description"
            minOccurs="0" />
          <xsd:element name="optional" minOccurs="0" />
          <xsd:group ref="typeDeclarationGroup" />
        </xsd:sequence>
        <!-- Extension -->
        <xsd:attribute name="access" use="optional"
          default="read-write">
          <xsd:simpleType>
            <xsd:list itemType="accessModeType" />
          </xsd:simpleType>
        </xsd:attribute>
        <!-- /Extension -->
        <xsd:attribute name="componentID"
          type="xsd:unsignedInt" use="required" />
      </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 name="metadataID" type="xsd:integer"/>
          <xsd:element ref="description"
            minOccurs="0" />
          <xsd:choice>
            <xsd:element name="typeRef"
              type="typeRefNMTOKEN" />
            <xsd:element name="atomic" type="atomicType" />
          <!-- Extension -->
          <xsd:element name="array" type="arrayType">
            <!--declare keys to have unique IDs -->
            <xsd:key name="contentKeyID1">
              <xsd:selector xpath="lfb:contentKey" />
              <xsd:field xpath="@contentKeyID" />
            </xsd:key>
          </xsd:element>
        </xsd:sequence>
      </xsd:complexType>
    </xsd:element>
  </xsd:sequence>
</xsd:complexType>
```

```

        </xsd:key>
        <!-- /Extension -->
    </xsd:element>
    <xsd:element name="struct" type="structType">
        <!-- Extension -->
        <!-- key declaration to make componentIDs
            unique in a struct -->
        <xsd:key name="structComponentID1">
            <xsd:selector xpath="lfb:component" />
            <xsd:field xpath="@componentID" />
        </xsd:key>
        <!-- /Extension -->
    </xsd:element>
</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="components"
                        type="LFBComponentsType"
                        minOccurs="0" />
                    <xsd:element name="capabilities"
                        type="LFBCapabilitiesType"
                        minOccurs="0" />
                    <xsd:element name="events" type="eventsType"
                        minOccurs="0" />
                    <xsd:element ref="description"
                        minOccurs="0" />
                </xsd:sequence>
                <xsd:attribute name="LFBClassID"
                    type="xsd:unsignedInt" use="required" />
            </xsd:complexType>
        </xsd:element>
    </xsd:sequence>
</xsd:complexType>

```

```
</xsd:complexType>
<!-- Key constraint to ensure unique attribute names
      within a class: -->
<xsd:key name="components">
  <xsd:selector xpath="lfb:components/lfb:component" />
  <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:key name="events">
  <xsd:selector xpath="lfb:events/lfb:event" />
  <xsd:field xpath="lfb:name" />
</xsd:key>
<xsd:key name="eventsIDs">
  <xsd:selector xpath="lfb:events/lfb:event" />
  <xsd:field xpath="@eventID" />
</xsd:key>
<xsd:key name="componentIDs">
  <xsd:selector xpath="lfb:components/lfb:component" />
  <xsd:field xpath="@componentID" />
</xsd:key>
<xsd:key name="capabilityIDs">
  <xsd:selector xpath="lfb:capabilities/lfb:capability"/>
  <xsd:field xpath="@componentID" />
</xsd:key>
<xsd:key name="ComponentCapabilityComponentIDUniqueness">
  <xsd:selector
    xpath="lfb:components/lfb:component|
           lfb:capabilities/lfb:capability|lfb:events" />
  <xsd:field xpath="@componentID|@baseID" />
</xsd:key>
</xsd:element>
</xsd:sequence>
</xsd:complexType>
<xsd:simpleType name="versionType">
  <xsd:restriction base="xsd:NMTOKEN">
    <xsd:pattern value="[1-9][0-9]*\.[0-9]*" />
  </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="xsd:boolean"
      use="optional" default="0" />
  </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="xsd:boolean"
        use="optional" default="0" />
    </xsd:complexType>
    <xsd:complexType name="portProductType">
      <xsd:sequence>
        <xsd:element name="frameProduced" minOccurs="0">
          <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: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="LFBComponentsType">
  <xsd:sequence>

```

```
<xsd:element name="component" maxOccurs="unbounded">
  <xsd:complexType>
    <xsd:sequence>
      <xsd:element name="name" type="xsd:NMTOKEN" />
      <xsd:element ref="synopsis" />
      <xsd:element ref="description"
        minOccurs="0" />
      <xsd:element name="optional" 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:attribute name="componentID"
      type="xsd:unsignedInt" use="required" />
  </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:element name="optional" minOccurs="0" />
          <xsd:group ref="typeDeclarationGroup" />
        </xsd:sequence>
        <xsd:attribute name="componentID" type="xsd:integer"
          use="required" />
      </xsd:complexType>
```

```
        </xsd:element>
      </xsd:sequence>
    </xsd:complexType>
    <xsd:complexType name="eventsType">
      <xsd:sequence>
        <xsd:element name="event" maxOccurs="unbounded">
          <xsd:complexType>
            <xsd:sequence>
              <xsd:element name="name" type="xsd:NMTOKEN" />
              <xsd:element ref="synopsis" />
              <xsd:element name="eventTarget"
                type="eventPathType" />
              <xsd:element ref="eventCondition" />
              <xsd:element name="eventReports"
                type="eventReportsType" minOccurs="0" />
              <xsd:element ref="description"
                minOccurs="0" />
            </xsd:sequence>
            <xsd:attribute name="eventID" type="xsd:integer"
              use="required" />
          </xsd:complexType>
        </xsd:element>
      </xsd:sequence>
      <xsd:attribute name="baseID" type="xsd:integer"
        use="optional" />
    </xsd:complexType>
    <!-- the substitution group for the event conditions -->
    <xsd:element name="eventCondition" abstract="true" />
    <xsd:element name="eventCreated"
      substitutionGroup="eventCondition" />
    <xsd:element name="eventDeleted"
      substitutionGroup="eventCondition" />
    <xsd:element name="eventChanged"
      substitutionGroup="eventCondition" />
    <xsd:element name="eventGreaterThan"
      substitutionGroup="eventCondition" />
    <xsd:element name="eventLessThan"
      substitutionGroup="eventCondition" />
    <!-- Extension -->
    <xsd:element name="eventBecomesEqualTo"
      substitutionGroup="eventCondition"/>
    <!-- /Extension -->
    <xsd:complexType name="eventPathType">
      <xsd:sequence>
        <xsd:element ref="eventPathPart" maxOccurs="unbounded" />
      </xsd:sequence>
    </xsd:complexType>
    <!-- the substitution group for the event path parts -->
```



```
<xsd:element name="eventPathPart" type="xsd:string"
  abstract="true" />
<xsd:element name="eventField" type="xsd:string"
  substitutionGroup="eventPathPart" />
<xsd:element name="eventSubscript" type="xsd:string"
  substitutionGroup="eventPathPart" />
<xsd:complexType name="eventReportsType">
  <xsd:sequence>
    <xsd:element name="eventReport" type="eventPathType"
      maxOccurs="unbounded" />
  </xsd:sequence>
</xsd:complexType>
<xsd:simpleType name="booleanType">
  <xsd:restriction base="xsd:string">
    <xsd:enumeration value="0" />
    <xsd:enumeration value="1" />
  </xsd:restriction>
</xsd:simpleType>
</xsd:schema>
```

OpenFlow XML Library

5. Acknowledgements

The author would like to acknowledge Joel Halpern, Jamal Hadi and Dave Hood for their comments and discussion that helped shape this document in a better way.

6. IANA Considerations

This memo includes no request to IANA.

7. Security Considerations

The security considerations that have been described in the ForCES Model RFC [RFC5812] apply to this document as well.

8. References

8.1. Normative References

[I-D.ietf-forces-ceha]
Ogawa, K., Wang, W., Haleplidis, E., and J. Salim, "ForCES Intra-NE High Availability", draft-ietf-forces-ceha-07 (work in progress), May 2013.

[OpenFlowSpec1.1]

<http://www.OpenFlow.org/>, "The OpenFlow 1.1 Specification.", , <<http://www.OpenFlow.org/documents/OpenFlow-spec-v1.1.0.pdf>>.

- [RFC5810] Doria, A., Hadi Salim, J., Haas, R., Khosravi, H., Wang, W., Dong, L., Gopal, R., and J. Halpern, "Forwarding and Control Element Separation (ForCES) Protocol Specification", RFC 5810, March 2010.
- [RFC5812] Halpern, J. and J. Hadi Salim, "Forwarding and Control Element Separation (ForCES) Forwarding Element Model", RFC 5812, March 2010.

8.2. Informative References

- [RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, March 1997.

Author's Address

Evangelos Haleplidis
University of Patras
Department of Electrical and Computer Engineering
Patras 26500
Greece

Email: ehalep@ece.upatras.gr

Internet Engineering Task Force
Internet-Draft
Intended status: Standards Track
Expires: May 20, 2014

E. Haleplidis
University of Patras
J. Halpern
Ericsson
November 16, 2013

ForCES Packet Parallelization
draft-haleplidis-forces-packet-parallelization-04

Abstract

Forwarding and Control Element Separation (ForCES) defines an architectural framework and associated protocols to standardize information exchange between the control plane and the forwarding plane in a ForCES Network Element (ForCES NE). RFC5812 has defined the ForCES Model provides a formal way to represent 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.

Many network devices support parallel packet processing. This document describes how ForCES can model a network device's parallelization datapath.

Status of This Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at <http://datatracker.ietf.org/drafts/current/>.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on May 20, 2014.

Copyright Notice

Copyright (c) 2013 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust's Legal Provisions Relating to IETF Documents (<http://trustee.ietf.org/license-info>) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document. Code Components extracted from this document must include Simplified BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the Simplified BSD License.

Table of Contents

1. Terminology and Conventions	2
1.1. Requirements Language	3
1.2. Definitions	3
2. Introduction	4
3. Packet Parallelization	4
4. Parallel Base Types	9
4.1. Frame Types	9
4.2. Data Types	9
4.3. MetaData Types	10
5. Parallel LFBs	10
5.1. Splitter	10
5.1.1. Data Handling	10
5.1.2. Components	11
5.1.3. Capabilities	11
5.1.4. Events	11
5.2. Merger	11
5.2.1. Data Handling	11
5.2.2. Components	12
5.2.3. Capabilities	13
5.2.4. Events	13
6. XML for Parallel LFB library	13
7. Acknowledgements	19
8. IANA Considerations	20
8.1. LFB Class Names and LFB Class Identifiers	20
8.2. Metadata ID	20
9. Security Considerations	21
10. References	21
10.1. Normative References	21
10.2. Informative References	21
Authors' Addresses	21

1. Terminology and Conventions

1.1. Requirements Language

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 [RFC2119].

1.2. Definitions

This document follows the terminology defined by the ForCES Model in [RFC5812]. The required definitions are repeated below for clarity.

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 LFB modeling of individual Logical Functional Block (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 [RFC5810].

LFB (Logical Functional Block) Class (or type) - A template that represents a fine-grained, logically separable aspect of FE processing. Most LFBs relate to packet processing in the data path. LFB classes are the basic building blocks of the FE model.

LFB Instance - As a packet flows through an FE along a data path, it flows through one or multiple LFB instances, where each LFB is an instance of a specific LFB class. Multiple instances of the same LFB class can be present in an FE's data path. Note that we often refer to LFBs without distinguishing between an 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. XML is used to provide a formal definition of the necessary structures for the modeling. Four types of information are defined in the LFB model. The core part of the LFB model is the LFB class definitions; the other three types of information define constructs associated with and used by the class definition. These are reusable data types, supported frame (packet) formats, and metadata.

Element - Element is generally used in this document in accordance with the XML usage of the term. It refers to an XML tagged part of an XML document. For a precise definition, please see the full set of XML specifications from the W3C. This term is included in this list for completeness because the ForCES formal model uses XML.

Attribute - Attribute is used in the ForCES formal modeling in accordance with standard XML usage of the term, i.e., to provide attribute information included in an XML tag.

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 the per-packet state is implemented within actual hardware. Metadata is sent between the FE and the CE on redirect packets.

ForCES Component - A ForCES Component is a well-defined, uniquely identifiable and addressable ForCES model building block. A component has a 32-bit ID, name, type, and an optional synopsis description. These are often referred to simply as components.

LFB Component - An LFB component is a ForCES component that defines the Operational parameters of the LFBs that must be visible to the CEs.

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

2. Introduction

A lot of network devices can process packets in a parallel manner. The ForCES Model [RFC5812] presents a formal way to describe the Forwarding Plane's datapath with Logical Function Blocks (LFBs) using XML. This document describes how packet parallelization can be described with the ForCES model.

The modelling concept has been influenced by Cilc [Cilc]. Cilc is a programming language that has been developed since 1994 at the MIT Laboratory to allow programmers to identify elements that can be executed in parallel. The two Cilc concepts used in this document is spawn and sync. Spawn being the place where parallel work can start and sync being the place where the parallel work finishes and must collect all parallel output.

3. Packet Parallelization

This document addresses the following two types of packet parallelization:

1. Flood - where a copy of a packet is sent to multiple LFBs to be processed in parallel.

2. Split - where the packet will be split in equal size chunks specified by the CE and sent to multiple LFB instances probably of the same LFB class to be processed in parallel.

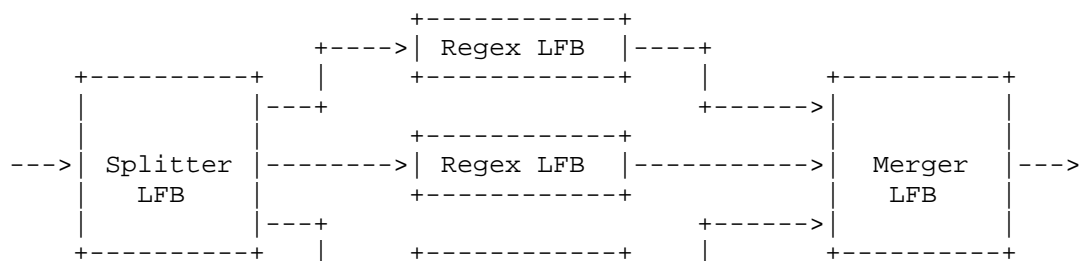
It must be noted that the process of copying the packet in the Flood parallel type is implementation depended and is loosely defined here. An implementor may either decide to physical copy the packet and send all packets on the parallel paths, or may decide to logically copy the packet by simply sending for example pointers of the same packet provided that the necessary interlocks are taken into account. The implementor has to take into account the device's characteristics to decide which approach fits best to the hardware.

Additionally in the split parallel type, while harder, the implementor may also decide to logically split the packet and send for example pointers to parts of the packet, provided that the necessary interlocks are managed.

This document introduces two LFBs that are used in before and after the parallelization occurs:

1. Splitter - similar to Cilc's spawn. An LFB that will split the path of a packet and be sent to multiple LFBs to be processed in parallel.
2. Merger - similar to Cilc's sync. An LFB that will receive packets or chunks of the same initial packet and merge them into one.

Both parallel packet distribution types can currently be achieved with the ForCES model. The splitter LFB has one group output that produces either chunks or packets to be sent to LFBs for processing and the merger LFB has one group input that expects either packets or chunks to aggregate all the parallel packets or chunks and produce a single packet. Figure 1 shows an simple example of a split parallel datapath along with the splitter and merger LFB. Figure 2 shows an example of a flood parallel datapath along with the splitter and merger LFB.



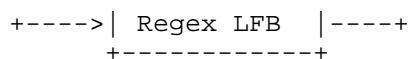


Figure 1: Simple split parallel processing

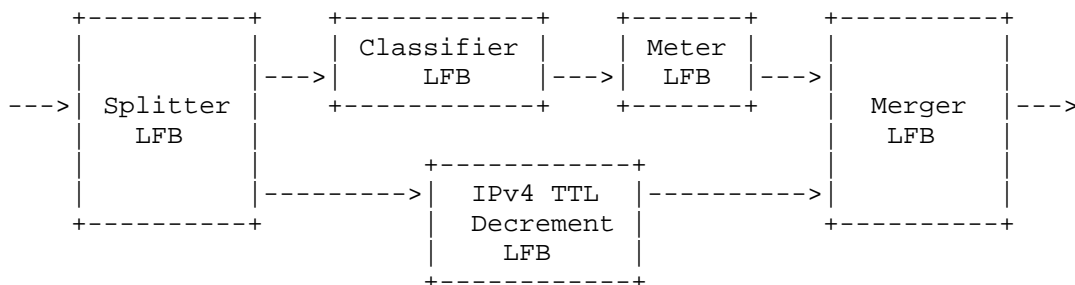


Figure 2: Simple flood parallel processing

This version of the modelling framework does not allow for nested parallel datapath topologies. This decision was reached by the authors and the ForCES working group as there was no strong use case or need at the time. This led to a more simple metadata definition needed to be transported between the splitter and the corresponding merger. If there is a need for nested parallel datapaths a new version of a splitter and merger will be needed to be defined as well as an augmentation to the defined metadata.

One important element to a developer is the ability to define which LFBs can be used in a parallel mode, with which other LFBs can they be parallelized with and the order of the LFBs can be assembled. This information must be accessible in the core LFBs and therefore this document needs to append one more capability in the FEObject LFB. The topology of the parallel datapath can be deferred and manipulated from the FEObject LFB's LFBTopology.

The FEObject LFB currently specifies the LFBTopology and supported LFBs in an FE. In order to support parallelization the following component is needed in order to specify each LFB that can be used in a parallel mode :

- o The Name of the LFB.
- o The Class ID of the LFB.
- o The Version of the LFB.
- o The number of instances that class can support in parallel.

- o A list of LFB classes that can follow this LFB class in a pipeline for a parallel path.
- o A list of LFB classes that can exist before this LFB class in a pipeline for a parallel path.
- o A list of LFB classes that can process packets or chunks in parallel with this LFB class.

```
<!-- Datatype -->
<dataTypeDef>
  <name>ParallelLFBType</name>
  <synopsis>Table entry for parallel LFBs</synopsis>
  <struct>
    <component componentID="1">
      <name>LFBName</name>
      <synopsis>The name of an LFB Class</synopsis>
      <typeRef>string</typeRef>
    </component>
    <component componentID="2">
      <name>LFBClassID</name>
      <synopsis>The id of the LFB Class</synopsis>
      <typeRef>uint32</typeRef>
    </component>
    <component componentID="3">
      <name>LFBVersion</name>
      <synopsis>The version of the LFB Class used by this FE
</synopsis>
      <typeRef>string</typeRef>
    </component>
    <component componentID="4">
      <name>LFBParallelOccurenceLimit</name>
      <synopsis>The upper limit of instances of the same
parallel LFBs of this class</synopsis>
      <optional />
      <typeRef>uint32</typeRef>
    </component>
    <component componentID="5">
      <name>AllowedParallelAfters</name>
      <synopsis>List of LFB Classes that can follow this LFB
in a parallel pipeline</synopsis>
      <optional />
      <array>
        <typeRef>uint32</typeRef>
      </array>
    </component>
    <component componentID="6">
      <name>AllowedParallelBeforees</name>
```

```

    <synopsis>List of LFB Classes that this LFB class can
        follow in a parallel pipeline</synopsis>
    <optional />
    <array>
        <typeRef>uint32</typeRef>
    </array>
</component>
<component componentID="7">
    <name>AllowedParallel</name>
    <synopsis>List of LFB Classes that this LFB class be run
        in parallel with</synopsis>
    <array>
        <typeRef>uint32</typeRef>
    </array>
</component>
</struct>
</dataTypeDef>

<!-- Capability -->
    <capability componentID="32">
        <name>ParallelLFBs</name>
        <synopsis>List of all supported parallel LFBs</synopsis>
        <array type="Variable-size">
            <typeRef>ParallelLFBType</typeRef>
        </array>
    </capability>

```

Figure 3: XML Definition for FEObjectLFB extension

While the ForCES model cannot describe how the splitting or the merging is actually done as that is an implementation issue of the actual LFB, however this document defines operational parameters to control the splitting and merging, namely the size of the chunks, what happens if a packet or chunk has been marked as invalid and whether the merge LFB should wait for all packets or chunks to arrive. Additionally this document defines metadata, which contain necessary information to assist the merging procedure. The following metadata set as a struct is defined:

1. ParallelType - Flood or split
2. Correlator - Identify packets or chunks that belonged to the initial packet that entered the Splitter LFB
3. ParallelNum - Number of packet or chunk for specific Correlator.
4. ParallelPartsCount - Total number of packets or chunks for specific Correlator.

This metadata is produced from the Splitter LFB and is opaque to LFBs in parallel paths and is passed along to the merger LFB without being consumed.

In case of a packet/chunk being branded invalid by an LFB in a parallel path, it MUST be sent by an output port of said LFB

An LFB inside a parallel path decides that a packet or a chunk has to be dropped it MAY drop it but the metadata MUST be sent to the Merger LFB's InvalidIn input port for merging purposes.

Additional metadata produced by LFBs inside a datapath MAY be aggregated within the Merger LFB and sent on after the merging process. In case of receiving the same metadata definition with multiple values the merger LFB MUST keep the first received from a valid packet or chunk.

4. Parallel Base Types

4.1. Frame Types

One frame type has been defined in this library.

Frame Type Name	Synopsis
Chunk	A chunk is a frame that is part of an original larger frame

Parallel Frame Types

4.2. Data Types

One data type has been defined in this library.

DataType Name	Type	Synopsis
ParallelTypes	Atomic uchar. Special Values Flood (0), Split (1).	The type of parallelization this packet will go through

Parallel Data Types

4.3. MetaData Types

The following metadata structure with ID 16, using the ForCES model extension [I-D.ietf-forces-model-extension], is defined for the parallelization library:

Metadata Name	Type	ID	Synopsis
ParallelType	uchar	1	The type of parallelization this packet will go through. 0 for flood, 1 for split.
Correlator	uint32	2	An identification number to specify that packets or chunks belong to the same parallel work.
ParallelNum	uint32	3	Defines the number of the specific packet or chunk of the specific parallel ID.
ParallelPartsCount	uint32	4	Defines the total number of packets or chunks for the specific parallel ID.

Metadata Structure for Merging

5. Parallel LFBs

5.1. Splitter

A splitter LFB takes part in parallelizing the processing datapath by sending either the same packet or chunks of the same packet to multiple LFBs.

5.1.1. Data Handling

The splitter LFB receives any kind of packet via the singleton input, Input. Depending upon the CE's configuration of the ParallelType component, if the parallel type is of type flood (0), the same packet MUST be sent through all of the group output ParallelOut's instances. If the parallel type is of type split (1), the packet will be split into same size chunks except the last which MAY be smaller, with the max size being defined by the ChunkSize component. All chunks will be sent out in a round-robin fashion through the group output

ParallelOut's instances. Each packet or chunk will be accompanied by the following metadata set as a struct :

- o ParallelType - The paralleltype split or flood.
- o Parallel ID - generated by the splitter LFB to identify that chunks or packets belong to the same parallel work.
- o Parallel Num - each chunk or packet of a parallel id will be assigned a number in order for the merger LFB to know when it has gathered them all along with the ParallelPartsCount metadata.
- o ParallelPartsCount - the number of chunks or packets for the specific parallel id.

5.1.2. Components

This LFB has only two components specified. The first is the ParallelType, an uint32 that defines how the packet will be processed by the Splitter LFB. The second is the ChunkSize, an uint32 that specifies the maximum size of a chunk when a packet is split into multiple same size chunks.

5.1.3. Capabilities

This LFB has only one capability specified, the MinMaxChunkSize a struct of a uint32 to specify the minimum chunk size and a uint32 to specify the maximum chunk size.

5.1.4. Events

This LFB has no events specified.

5.2. Merger

A merger LFB receives multiple packets or multiple chunks of the same packet and merge them into one merged packet.

5.2.1. Data Handling

The Merger LFB receives either a packet or a chunk via the group input ParallelIn, along with the ParallelType metadata to identify whether what was received was a packet or a chunk, the Correlator, the ParallelNum and the ParallelPartsCount.

In case that an LFB has dropped a packet or a chunk within a parallel path the merger LFB MAY receive only the metadata or both metadata and packet or chunk through the InvalidIn group input port. It

SHOULD receive a metadata specifying the error code. Current defined metadata's in the Base LFB Library [RFC6956] are the ExceptionID and the ValidateErrorID. The Merger LFB MAY store the parallel metadata along with the exception metadata as a string in the optional InvalidateMetadataSets as a means for the CE to debug errors in the parallel path.

If the MergeWaitType is set to false the Merger LFB will initiate the merge process upon receiving the first packet. If false it will wait for all packet in the Correlator to arrive.

If one packet or chunk has been received through the InvalidIn port then the merging procedure will be operate as configured by the InvalidAction component. If the InvalidAction component has been set to 0 then if one packet or chunk is not valid all will dropped, else the process will initiate. Once the merging process has been finished the resulting packet will be sent via the singleton output port PacketOutput.

If the Merger LFB receives different values for the same metadata from different packets or chunks that has the same correlator then the Merger LFB will use the first metadata from a packet or chunk that entered the LFB through the ParallelIn input port.

5.2.2. Components

This LFB has the following components specified:

1. InvalidAction - a uchar defining what the Merge LFB will do if an invalid chunk or packet is received. If set to 0 (DropAll) the merge will be considered invalid and all chunks or packets will be dropped. If set to 1 (Continue) the merge will continue.
2. MergeWaitType - a boolean. If true the Merger LFB will wait for all packets or chunks to be received prior to sending out a response. If false, when one packet or a chunk with a response is received by the merge LFB it will start with the merge process.
3. InvalidMergesCounter - a uint32 that counts the number of merges where there is at least one packet or chunk that entered the merger LFB through the InvalidIn input port.
4. InvalidAllCounter - a uint 32 that counts the number of merges where all packets/chunks entered the merger LFB through the InvalidIn input port.

5. InvalidIDCounters - a struct of two arrays. Each array has a uint32 per row. Each array counts number of invalid merges where at least one packet or chunk entered through InvalidID per error ID. The first array is the InvalidExceptionID and the second is the InvalidValidateErrorID.
6. InvalidMetadataSets - an array of strings. An optional component that stores metadata sets along with the error id as a string. This could provide a debug information to the CE regarding errors in the parallel paths.

5.2.3. Capabilities

This LFB has no capabilities specified.

5.2.4. Events

This LFB specifies only two event. The first detects whether the InvalidMergesCounter has exceeded a specific value and the second detects whether the InvalidAllCounter has exceeded a specific value. Both error reports will send the respective counter value.

6. XML for Parallel LFB library

```
<?xml version="1.0" encoding="UTF-8"?>
<LFBLibrary xmlns="urn:ietf:params:xml:ns:forces:lfbmodel:1.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="urn:ietf:params:xml:ns:forces:lfbmodel:1.0"
  provides="Parallel">
  <load library="BaseTypeLibrary" location="BaseTypeLibrary.LFB" />
  <frameDefs>
    <frameDef>
      <name>Chunk</name>
      <synopsis>A chunk is a frame that is part of an original
        larger frame</synopsis>
    </frameDef>
  </frameDefs>
  <dataTypeDefs>
    <dataTypeDef>
      <name>ParallelTypes</name>
      <synopsis>The type of parallelization this packet will go
        through</synopsis>
      <atomic>
        <baseType>uchar</baseType>
        <specialValues>
          <specialValue value="0">
            <name>Flood</name>
            <synopsis>The packet/chunk has been sent as a whole
```

```

        to multiple recipients</synopsis>
    </specialValue>
    <specialValue value="1">
        <name>Split</name>
        <synopsis>The packet/chunk has been split into
            multiple chunks and sent to recipients</synopsis>
    </specialValue>
    </specialValues>
</atomic>
</dataTypeDef>
</dataTypeDefs>
<metadataDefs>
    <metadataDef>
        <name>ParallelMetadataSet</name>
        <synopsis>A metadata Set for parallelization related LFBs
        </synopsis>
        <metadataID>32</metadataID>
        <struct>
            <component componentID="1">
                <name>ParallelType</name>
                <synopsis>The type of parallelization this packet/chunk
                    has gone through</synopsis>
                <typeRef>ParallelTypes</typeRef>
            </component>
            <component componentID="2">
                <name>Correlator</name>
                <synopsis>An identification number to specify that
                    packets or chunks originate from the same packet.
                </synopsis>
                <typeRef>uint32</typeRef>
            </component>
            <component componentID="3">
                <name>ParallelNum</name>
                <synopsis>Defines the number of the specific packet or
                    chunk of the specific parallel ID.</synopsis>
                <typeRef>uint32</typeRef>
            </component>
            <component componentID="4">
                <name>ParallelPartsCount</name>
                <synopsis>Defines the total number of packets or chunks
                    for the specific parallel ID.</synopsis>
                <typeRef>uint32</typeRef>
            </component>
        </struct>
    </metadataDef>
</metadataDefs>
<LFBClassDefs>
    <LFBClassDef LFBClassID="18">

```



```
<name>Splitter</name>
<synopsis>A splitter LFB takes part in parallelizing the
  processing datapath. It will either send the same packet
  or chunks of one packet to multiple LFBs</synopsis>
<version>1.0</version>
<inputPorts>
  <inputPort>
    <name>PacketIn</name>
    <synopsis>An input port expecting any kind of frame
    </synopsis>
    <expectation>
      <frameExpected>
        <ref>Arbitrary</ref>
      </frameExpected>
    </expectation>
  </inputPort>
</inputPorts>
<outputPorts>
  <outputPort group="true">
    <name>ParallelOut</name>
    <synopsis>An parallel output port that sends the same
      packet to all output instances or chunks of the same
      packet different chunk on each instance.</synopsis>
    <product>
      <frameProduced>
        <ref>Arbitrary</ref>
        <ref>Chunk</ref>
      </frameProduced>
      <metadataProduced>
        <ref>ParallelMetadataSet</ref>
      </metadataProduced>
    </product>
  </outputPort>
</outputPorts>
<components>
  <component componentID="1" access="read-write">
    <name>ParallelType</name>
    <synopsis>The type of parallelization this packet will
      go through</synopsis>
    <typeRef>ParallelTypes</typeRef>
  </component>
  <component componentID="2" access="read-write">
    <name>ChunkSize</name>
    <synopsis>The size of a chunk when a packet is split
      into multiple same size chunks</synopsis>
    <typeRef>uint32</typeRef>
  </component>
</components>
```

```
<capabilities>
  <capability componentID="31">
    <name>MinMaxChunkSize</name>
    <synopsis>The minimum and maximum size of a chunk
      capable of splitted by this LFB</synopsis>
    <struct>
      <component componentID="1">
        <name>MinChunkSize</name>
        <synopsis>Minimum chunk size</synopsis>
        <optional/>
        <typeRef>uint32</typeRef>
      </component>
      <component componentID="2">
        <name>MaxChunkSize</name>
        <synopsis>Maximum chunk size</synopsis>
        <typeRef>uint32</typeRef>
      </component>
    </struct>
  </capability>
</capabilities>
</LFBClassDef>
<LFBClassDef LFBClassID="19">
  <name>Merger</name>
  <synopsis>A merger LFB receives multiple packets or multiple
    chunks of the same packet and merge them into one merged
    packet</synopsis>
  <version>1.0</version>
  <inputPorts>
    <inputPort group="true">
      <name>ParallelIn</name>
      <synopsis>An parallel input port that accepts packets
        or chunks from all output instances</synopsis>
      <expectation>
        <frameExpected>
          <ref>Arbitrary</ref>
          <ref>Chunk</ref>
        </frameExpected>
        <metadataExpected>
          <ref>ParallelMetadataSet</ref>
        </metadataExpected>
      </expectation>
    </inputPort>
    <inputPort group="true">
      <name>InvalidIn</name>
      <synopsis>When a packet is sent out of an error port of
        an LFB in a parallel path will be sent to this
        output port in the Merger LFB</synopsis>
      <expectation>
```

```

    <frameExpected>
      <ref>Arbitrary</ref>
      <ref>Chunk</ref>
    </frameExpected>
    <metadataExpected>
      <one-of>
        <ref>ExceptionID</ref>
        <ref>ValidateErrorID</ref>
      </one-of>
    </metadataExpected>
  </expectation>
</inputPort>
</inputPorts>
<outputPorts>
  <outputPort>
    <name>PacketOutput</name>
    <synopsis>An output port expecting any kind of frame
    </synopsis>
    <product>
      <frameProduced>
        <ref>Arbitrary</ref>
      </frameProduced>
    </product>
  </outputPort>
</outputPorts>
<components>
  <component componentID="1" access="read-write">
    <name>InvalidAction</name>
    <synopsis>What the Merge LFB will do if an invalid
      chunk or packet is received</synopsis>
    <atomic>
      <baseType>uchar</baseType>
      <specialValues>
        <specialValue value="0">
          <name>DropAll</name>
          <synopsis>Drop all packets or chunks
          </synopsis>
        </specialValue>
        <specialValue value="1">
          <name>Continue</name>
          <synopsis>Continue with the merge</synopsis>
        </specialValue>
      </specialValues>
    </atomic>
  </component>
  <component componentID="2" access="read-write">
    <name>MergeWaitType</name>
    <synopsis>Whether the Merge LFB will wait for all

```

```
        packets or chunks to be received prior to sending
        out a response</synopsis>
        <typeRef>boolean</typeRef>
    </component>
    <component componentID="3" access="read-reset">
        <name>InvalidMergesCounter</name>
        <synopsis>Counts the number of merges where there is at
        least one packet/chunk that entered the merger LFB
        through the InvalidIn input port</synopsis>
        <typeRef>uint32</typeRef>
    </component>
    <component componentID="4" access="read-reset">
        <name>InvalidAllCounter</name>
        <synopsis>Counts the number of merges where all
        packets/chunks entered the merger LFB through the
        InvalidIn input port</synopsis>
        <typeRef>uint32</typeRef>
    </component>
    <component componentID="5" access="read-reset">
        <name>InvalidIDCounters</name>
        <synopsis>Counts number of invalid merges where at
        least one packet/chunk entered through InvalidID per
        error ID</synopsis>
        <struct>
            <component componentID="1">
                <name>InvalidExceptionID</name>
                <synopsis>Per Exception ID</synopsis>
                <array>
                    <typeRef>uint32</typeRef>
                </array>
            </component>
            <component componentID="2">
                <name>InvalidValidateErrorID</name>
                <synopsis>Per Validate Error ID</synopsis>
                <array>
                    <typeRef>uint32</typeRef>
                </array>
            </component>
        </struct>
    </component>
    <component componentID="6" access="read-reset">
        <name>InvalidMetadataSets</name>
        <synopsis>Buffers metadata sets along with the error id
        as a string.</synopsis>
        <optional/>
        <array>
            <typeRef>string</typeRef>
        </array>
```

```

        </component>
    </components>
    <events baseID="30">
        <event eventID="1">
            <name>ManyInvalids</name>
            <synopsis>An event that specifies if there are too many
                invalids</synopsis>
            <eventTarget>
                <eventField>InvalidCounter</eventField>
            </eventTarget>
            <eventGreaterThan></eventGreaterThan>
            <eventReports>
                <eventReport>
                    <eventField>InvalidMergesCounter</eventField>
                </eventReport>
            </eventReports>
        </event>
        <event eventID="2">
            <name>ManyAllInvalids</name>
            <synopsis>An event that specifies if there are too many
                invalids</synopsis>
            <eventTarget>
                <eventField>InvalidCounter</eventField>
            </eventTarget>
            <eventGreaterThan></eventGreaterThan>
            <eventReports>
                <eventReport>
                    <eventField>InvalidAllCounter</eventField>
                </eventReport>
            </eventReports>
        </event>
    </events>
</LFBClassDef>
</LFBClassDefs>
</LFBLibrary>

```

Figure 4: Parallel LFB library

7. Acknowledgements

The authors would like to thank Jamal Hadi Salim and Dave Hood for comments and discussions that made this document better.

8. IANA Considerations

8.1. LFB Class Names and LFB Class Identifiers

LFB classes defined by this document belong to LFBs defined by Standards Track RFCs. According to IANA, the registration procedure is Standards Action for the range 0 to 65535 and First Come First Served with any publicly available specification for over 65535. This specification includes the following LFB class names and LFB class identifiers:

LFB Class Identifier	LFB Class Name	LFB Version	Description	Reference
18	Splitter	1.0	A splitter LFB will either send the same packet or chunks of one packet to multiple LFBs.	This document
19	Merger	1.0	A merger LFB receives multiple packets or multiple chunks of the same packet and merge them into one.	This document

Logical Functional Block (LFB) Class Names and Class Identifiers

8.2. Metadata ID

The Metadata ID namespace is 32 bits long. Values assigned by this specification:

Value	Name	Definition
0x00000010	ParallelMetadataSet	This document

Metadata ID assigned by this specification

9. Security Considerations

10. References

10.1. Normative References

- [I-D.ietf-forces-model-extension]
Haleplidis, E., "ForCES Model Extension", draft-ietf-forces-model-extension-00 (work in progress), September 2013.
- [RFC5810] Doria, A., Hadi Salim, J., Haas, R., Khosravi, H., Wang, W., Dong, L., Gopal, R., and J. Halpern, "Forwarding and Control Element Separation (ForCES) Protocol Specification", RFC 5810, March 2010.
- [RFC5812] Halpern, J. and J. Hadi Salim, "Forwarding and Control Element Separation (ForCES) Forwarding Element Model", RFC 5812, March 2010.
- [RFC6956] Wang, W., Haleplidis, E., Ogawa, K., Li, C., and J. Halpern, "Forwarding and Control Element Separation (ForCES) Logical Function Block (LFB) Library", RFC 6956, June 2013.

10.2. Informative References

- [Cilk] MIT, "Cilk language", ,
<<http://supertech.csail.mit.edu/cilk/>>.
- [RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, March 1997.

Authors' Addresses

Evangelos Haleplidis
University of Patras
Department of Electrical and Computer Engineering
Patras 26500
Greece

Email: ehalep@ece.upatras.gr

Joel Halpern
Ericsson
P.O. Box 6049
Leesburg 20178
VA

Phone: +1 703 371 3043
Email: joel.halpern@ericsson.com

Internet Engineering Task Force
Internet-Draft
Intended status: Informational
Expires: July 9, 2014

J. Hadi Salim
Mojatatu Networks
January 5, 2014

ForCES Protocol Extensions
draft-jhs-forces-protoextenstion-02

Abstract

Experience in implementing and deploying ForCES architecture has demonstrated need for a few small extensions both to ease programmability and to improve wire efficiency of some transactions. This document describes extensions to the ForCES Protocol Specification[RFC 5810] semantics to achieve that end goal.

Status of This Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at <http://datatracker.ietf.org/drafts/current/>.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on July 9, 2014.

Copyright Notice

Copyright (c) 2014 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust's Legal Provisions Relating to IETF Documents (<http://trustee.ietf.org/license-info>) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document. Code Components extracted from this document must include Simplified BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the Simplified BSD License.

Table of Contents

1. Terminology and Conventions	2
1.1. Requirements Language	2
1.2. Definitions	2
2. Introduction	4
3. Problem Overview	4
3.1. Table Ranges	4
3.2. Error codes	5
4. Protocol Update Proposal	5
4.1. Table Ranges	5
4.2. Error Codes	6
4.2.1. New Codes	7
4.2.2. Vendor Codes	7
4.2.3. Extended Result TLV	7
5. IANA Considerations	8
6. Security Considerations	9
7. References	9
7.1. Normative References	9
7.2. Informative References	9
Author's Address	9

1. Terminology and Conventions

1.1. Requirements Language

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 [RFC2119].

1.2. Definitions

This document reiterates the terminology defined by the ForCES architecture in various documents for the sake of clarity.

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 LFB modeling of individual Logical Functional Block (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 [RFC5810].

LFB (Logical Functional Block) Class (or type) - A template that represents a fine-grained, logically separable aspect of FE processing. Most LFBs relate to packet processing in the data path. LFB classes are the basic building blocks of the FE model.

LFB Instance - As a packet flows through an FE along a data path, it flows through one or multiple LFB instances, where each LFB is an instance of a specific LFB class. Multiple instances of the same LFB class can be present in an FE's data path. Note that we often refer to LFBs without distinguishing between an 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. XML is used to provide a formal definition of the necessary structures for the modeling. Four types of information are defined in the LFB model. The core part of the LFB model is the LFB class definitions; the other three types of information define constructs associated with and used by the class definition. These are reusable data types, supported frame (packet) 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 the per-packet state is implemented within actual hardware. Metadata is sent between the FE and the CE on redirect packets.

ForCES Component - A ForCES Component is a well-defined, uniquely identifiable and addressable ForCES model building block. A component has a 32-bit ID, name, type, and an optional synopsis description. These are often referred to simply as components.

LFB Component - An LFB component is a ForCES component that defines the Operational parameters of the LFBs that must be visible to the CEs.

ForCES Protocol - Protocol that runs in the Fp reference points in the ForCES Framework [RFC3746].

ForCES Protocol Layer (ForCES PL) - A layer in the ForCES protocol architecture that defines the ForCES protocol messages, the protocol state transfer scheme, and the ForCES protocol architecture itself as defined in the ForCES Protocol Specification [RFC5810].

ForCES Protocol Transport Mapping Layer (ForCES TML) - A layer in ForCES protocol architecture that uses the capabilities of existing transport protocols to specifically address protocol message transportation issues, such as how the protocol messages are mapped to different transport media (like TCP, IP, ATM, Ethernet, etc.), and how to achieve and implement reliability,

ordering, etc. the ForCES SCTP TML [RFC5811] describes a TML that is mandated for ForCES.

2. Introduction

Experience in implementing and deploying ForCES architecture has demonstrated need for a few small extensions both to ease programmability and to improve wire efficiency of some transactions. This document describes a few extensions to the ForCES Protocol Specification [RFC5810] semantics to achieve that end goal.

This document describes and justifies the need for 2 small extensions which are backward compatible.

1. A table range operation to allow a controller or control application to request an arbitrary range of table rows.
2. Improved Error codes returned to the controller (or control application) to improve granularity of existing defined error codes.

3. Problem Overview

In this section we present sample use cases to illustrate the challenge being addressed.

3.1. Table Ranges

Consider, for the sake of illustration, an FE table with 1 million reasonably sized table rows which are sparsely populated. Assume, again for the sake of illustration, that there are 2000 table rows sparsely populated between the row indices 23-10023.

ForCES GET and DEL requests sent from a controller (or control app) are prepended with a path to a component and sent to the FE. In the case of indexed tables, the component path can either be to a table or a table row index. The approaches for retrieving or deleting a sizeable number of table rows is at the programmatically (from an application point of view unfriendly, tedious, and abusive of both compute and bandwidth resources.

As an example, a control application attempting to retrieve the first 2000 table rows appearing between row indices 23 and 10023 can achieve its goal in one of:

- o Dump the whole table and filter for the needed 2000 table rows.

- o Send upto 10000 ForCES PL requests with monotonically incrementing indices and stop when the needed 2000 entries are retrieved.
- o If the application had knowledge of which table rows existed (not unreasonable given the controller is supposed to be aware of state within an NE), then the application could take advantage of ForCES batching to send fewer large messages (each with different path entries for a total of two thousand).

As argued, while the above options exist - all are tedious.

3.2. Error codes

[RFC5810] has defined a generic set of error codes that are to be returned to the CE from an FE. Deployment experience has shown that it would be useful to have more fine grained error codes. As an example, the error code E_NOT_SUPPORTED could be mapped to many FE error source possibilities that need to be then interpreted by the caller based on some understanding of the nature of the sent request. This makes debugging more time consuming.

4. Protocol Update Proposal

This section describes proposals to update the protocol for issues discussed in Section 3

4.1. Table Ranges

We propose to add a Table-range TLV (type ID 0x117) that will be associated with the PATH-DATA TLV in the same manner the KEYINFO-TLV is.

```
OPER = GET
PATH-DATA:
  flags = F_SELTABRANGE, IDCount = 2, IDs = {1,6}
  TABLERANGE-TLV content = {11,23}
```

Figure 1: ForCES table range request

Figure 1 illustrates a GET request for a range of rows 11 to 23 of a table with component path of "1/6".

Path flag of F_SELTABRANGE (0x2 i.e bit 1, where bit 0 is F_SELKEY as defined in RFC 5810) is set to indicate the presence of the Table-range TLV. The pathflag bit F_SELTABRANGE can only be used in a GET or DEL and is mutually exclusive with F_SELKEY. The FE MUST enforce those constraints and reject a request with an error code of

E_INVALID_TFLAGS with a description of what the problem is (refer to Section 4.2).

The Table-range TLV contents constitute:

- o A 32 bit start index. An index of 0 implies the beginning of the table row.
- o A 32 bit end index. A value of 0xFFFFFFFFFFFFFFFF implies the last entry. XXX: Do we need to define the "end wildcard"?

The response for a table range query will either be:

- o The requested table data returned (when at least one referenced row is available); in such a case, a response with a path pointing to the table and whose data content contain the row(s) will be sent to the CE. The data content MUST be encapsulated in sparsedata TLV. The sparse data TLV content will have the "I" (in ILV) for each table row indicating the table indices.
- o An Extended result TLV when:
 - * Response is to a range delete request. The Result will either be:
 - + A success if any of the requested for rows is deleted
 - + A proper error code if none of the requested for rows cannot be deleted
 - * data is absent where the result code of E_EMPTY with an optional content string describing the nature of the error (refer to Section 4.2).
 - * When both a path key and path table range are reflected on the the pathflags, an error code of E_INVALID_TFLAGS with an optional content string describing the nature of the error (refer to Section 4.2).
 - * other standard ForCES errors (such as ACL constraints trying to retrieve contents of an unreadable table), accessing unknown components etc.

4.2. Error Codes

We propose several things:

1. A new set of error codes.

2. Allocating currently reserved codes for vendor use.
3. A new TLV, EXTENDED-RESULT-TLV (0x118) that will carry a code (which will be a superset of what is currently specified in RFC 5812) but also an optional cause content. This is illustrated in Figure 2.

4.2.1. New Codes

Extended-Result TLV Result Value is 32 bits and is a superset of RFC 5810 Result TLV Result Value. The new version code space is 32 bits as opposed to the RFC 5810 code size of 8 bits.

Code	Mnemonic	Details
0x100	E_EMPTY	Table is empty
0x101	E_INVALID_TFLAGS	Invalid table flags
0x102	E_INVALID_OP	Requested operation is invalid
0x103	E_CONGEST_NT	Node Congestion notification

Table 1: New codes

4.2.2. Vendor Codes

Codes 0x18-0xFE are reserved for use as vendor codes. Since these are freely available it is expected that the FE and CE side will both understand the semantics of any used codes.

4.2.3. Extended Result TLV

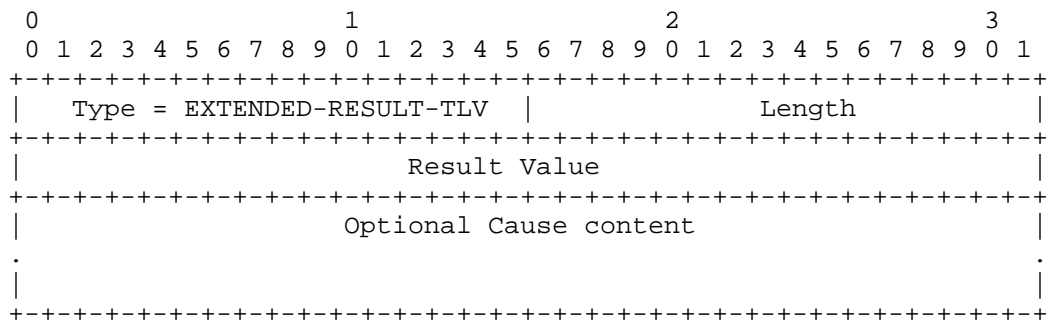


Figure 2: Extended Result TLV

- o Like all other ForCES TLVs, the Extended Result TLV is expected to be 32 bit aligned.

- o The Result Value derives and extends from the same current namespace as specified in RFC 5810, section 7.1.7. The main difference is that we now have 32 bit result value (as opposed to the old 8 bit).
- o The optional result content is defined to further disambiguate the result value. It is expected Utf-8 values to be used. However, vendor specific error codes may choose to specify different contents. Additionally, future codes may specify cause contents to be of types other than string..
- o It is recommended that the maximum size of the cause string should not exceed 32 bytes. We do not propose the cause string be standardized.

XXX: Backward compatibility may require that we add a FEPO capability to advertise ability to do extended results so that the CE is able to interpret the results and a FEPO compatibility flag to define what TLV setting would be used. Alternatively, the backward compatibility can be made a configuration option (which helps reduce clutter on FEPO LFB given that it is expected that in the future it makes sense for implementations to support only extended Result TLVs).

5. IANA Considerations

This document registers two new top Level TLVs and two new path flags.

The following new TLVs are defined:

- o Table-range TLV (type ID 0x117)
- o EXTENDED-RESULT-TLV (type ID 0x118)

The following new path flags are defined:

- o F_SELTABRANGE (value 0x2 i.e bit 1)

The Defined Result Values are changed:

- o codes 0x18-0xFE are reserved for vendor use.
- o codes 0x100-102 are defined by this document.

6. Security Considerations

TBD

7. References

7.1. Normative References

- [RFC3746] Yang, L., Dantu, R., Anderson, T., and R. Gopal, "Forwarding and Control Element Separation (ForCES) Framework", RFC 3746, April 2004.
- [RFC5810] Doria, A., Hadi Salim, J., Haas, R., Khosravi, H., Wang, W., Dong, L., Gopal, R., and J. Halpern, "Forwarding and Control Element Separation (ForCES) Protocol Specification", RFC 5810, March 2010.
- [RFC5811] Hadi Salim, J. and K. Ogawa, "SCTP-Based Transport Mapping Layer (TML) for the Forwarding and Control Element Separation (ForCES) Protocol", RFC 5811, March 2010.
- [RFC5812] Halpern, J. and J. Hadi Salim, "Forwarding and Control Element Separation (ForCES) Forwarding Element Model", RFC 5812, March 2010.

7.2. Informative References

- [RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, March 1997.

Author's Address

Jamal Hadi Salim
Mojatatu Networks
Suite 400, 303 Moodie Dr.
Ottawa, Ontario K2H 9R4
Canada

Email: hadi@mojatatu.com