Internet Engineering Task Force	E.H. Haleplidis
Internet-Draft	O.K. Koufopavlou
Intended status: Informational	S.D. Denazis
Expires: December 25, 2011	University of Patras
	June 23, 2011

ForCES Implementation Experience Draft
draft-haleplidis-forces-implementation-experience-03

Abstract

The Forwarding and Control Element Separation (ForCES) protocol defines a standard communication and control mechanism through which a Control Element (CE) can control the behavior of a Forwarding Element (FE). This document captures the experience of implementing the ForCES protocol and model. Its aim is to help others by providing examples and possible strategies for implementing the ForCES protocol.

Status of this Memo

Copyright Notice

Table of Contents

1. **Terminology and Conventions**

The terminology used in this document is the same as in the Forwarding and Control Element Separation Protocol *[RFC5810]* and part of it is copied in this document.
Control Element (CE): A logical entity that implements the ForCES protocol and uses it to instruct one or more FEs on how to process packets. CEs handle functionality such as the execution of control and signaling protocols.
Forwarding Element (FE): A logical entity that implements the ForCES protocol. FEs use the underlying hardware to provide per-packet

processing and handling as directed/controlled by one or more CEs via the ForCES protocol.

LFB (Logical Function Block): The basic building block that is operated on by the ForCES protocol. The LFB is a well-defined, logically separable functional block that resides in an FE and is controlled by the CE via the ForCES protocol. The LFB may reside at the FE's data path and process packets or may be purely an FE control or configuration entity that is operated on by the CE. Note that the LFB is a functionally accurate abstraction of the FE's processing capabilities, but not a hardware-accurate representation of the FE implementation.

LFB Class and LFB Instance: LFBs are categorized by LFB classes. An LFB instance represents an LFB class (or type) existence. There may be multiple instances of the same LFB class (or type) in an FE. An LFB class is represented by an LFB class ID, and an LFB instance is represented by an LFB instance ID. As a result, an LFB class ID associated with an LFB instance ID uniquely specifies an LFB existence.

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

ForCES Protocol: While there may be multiple protocols used within the overall ForCES architecture, the terms "ForCES protocol" and "protocol" refer to the Fp reference points in the ForCES framework *[RFC3746]*. This protocol does not apply to CE-to-CE communication, FE-to-FE communication, or communication between FE and CE managers. Basically, the ForCES protocol works in a master-slave mode in which FEs are slaves and CEs are masters. This document defines the specifications for this ForCES protocol.

2. Introduction

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). [RFC3654] has defined the ForCES requirements, and [RFC3746] has defined the ForCES framework. The ForCES protocol works in a master-slave mode in which FEs are slaves and CEs are masters. The protocol includes commands for transport of Logical Functional Block (LFB) configuration information, association setup, status, and event notifications, etc. The reader is encouraged to read the Forwarding and Control Element Separation Protocol *[RFC5810]* for further information.

[RFC5812] presents a formal way to define FE Logical Functional Blocks (LFBs) using XML. LFB configuration components, capabilities, and associated events are defined when LFBs are formally created. The LFBs within the Forwarding Element (FE) are accordingly controlled in a standardized way by the ForCES protocol.

The Transport Mapping Layer (TML) transports the protocol messages. The TML is where the issues of how to achieve transport level reliability, congestion control, multicast, ordering, etc., are handled. It is expected that more than one TML will be standardized. The various possible TMLs could vary their implementations based on the capabilities of underlying media and transport. However, since each TML is standardized, interoperability is guaranteed as long as both endpoints support the same TML. All ForCES Protocol Layer implementations must be portable across all TMLs. Although more than one TML may be standardized for the ForCES Protocol, all ForCES implementations must implement the SCTP TML *[RFC5811]*.
The Forwarding and Control Element Separation Applicability Statement *[RFC6041]* captures the applicable areas in which ForCES can be used.

2.1. Document Goal

This document captures the experience of implementing the ForCES protocol and model, and its main goal is not to tell others how to implement, but to provide alternatives, ideas and proposals as how it can be implemented.
Also, this document mentions possible problems and potential choices that can be made, in an attempt to help implementors develop their own products.
Additionally this document takes into account that the reader has become familiar with the three main ForCES RFCs, the Forwarding and Control Element Separation Protocol *[RFC5810]*, the Forwarding and Control Element Separation Forwarding Element Model *[RFC5812]* and the SCTP-Based Transport Mapping Layer (TML) for the Forwarding and Control Element Separation Protocol *[RFC5811]*.

3. ForCES Architecture

In general ForCES has undergone two successfull interoperability tests, where very few issues were caught and resolved.
This section discusses the ForCES architecture, implementation challenges and how to overcome them.

3.1. Pre-association setup - Initial Configuration

The initial configuration of the FE and the Control Element (CE) is done respectively by the FE Manager and the CE Manager. These entities have not as yet been standardized.
The simplest solution, are static configuration files, which play the role of the Managers and are read by FEs and CEs.
For more dynamic solutions however, it is expected that the Managers will be entities that will talk to each other and exchange details regarding the associations. Any developer can create any Manager, but they should at least be able to exchange the following details:
From the FE Manager side:

1. FE Identifiers (FEIDs)

2. FE IP addresses, if the FEs and CEs will be communicating via network.

3. TML. The TML that will be used. If this is omitted, then SCTP must be chosen as default.

4. TML Priority ports. If this is omitted as well, then the CE must use the default values from the respective TML RFC.

From the CE Manager side:

1. CE Identifiers (CEIDs)

2. CE IP addresses, if the FEs and CEs will be communicating via network.

3. TML. The TML that will be used. If this is omitted, then SCTP must be chosen as default.

4. TML Priority ports. If this is omitted as well, then the FE must use the default values from the respective TML RFC.

3.2. TML

All ForCES implementations must support the SCTP as TML. Even if another TML will be chosen by the developer, SCTP is mandatory and must be supported.
There are several issues that should concern a developer for the TML.

1. Security. TML must be secure according to the respective RFC. For SCTP you have to use IPsec.

2. Remote connection. While ForCES is meant to be used locally, both interoperability tests have proven that ForCES can be deployed everywhere where SCTP/IP is available. In both interoperability tests there were connections between Greece and China and the performance was very satisfactory. However in order for the FE and CE to work in a not local environment if they are behind NATs an implementor must ensure that the SCTP-TML ports are forwarded to the CE and/or FE and if there is a firewall it will allow the SCTP ports through. These were identified during the first ForCES interoperability test and documented in the Implementation Report for Forwarding and Control Element Separation *[RFC6053]*.

3.3. Model

The ForCES model inherently is very dynamic. Using basic atomic data types that are specified in the model, new datatypes can be built using atomic (single valued) and/or compound (structures and arrays). Thus developers are free to create their own LFBs. One other advantage that the ForCES model provides is inheritance. New versions of existing LFBs can be created to suit any extra developer requirements.
The difficulty for a developer is to create an architecture that is completely scalable so there is no need to write the same code for new LFBs, or for new components, etc. Just create code for the defined atomic values and then new components can be built based on already written code thus re-using it.
The model itself provides the key which is inheritance.

3.3.1. Components

First, a basic component needs to be created as the mother of all the components with the basic parameters of all the components:

*The ID of the component.

*The access rights of that component.

*If it is an optional component.

*If it is of variable size.

*Minimum data size.

*Maximum data size.

If the data size of the component is not variable, then the size is either the minimum or the maximum size, as both should have the same value.
Next, some basic functions are in order:

*A common constructor.

*A common destructor.

*Retrieve Component ID.

*Retrieve access right property.

*Query if it is an optional component.

*Get Full Data.

*Set Full Data.

*Get Sparse Data.

*Set Sparse Data.

*Del Full Data.

*Del Sparse Data.

*Get Property

*Set Property

*Get Value.

*Set Value.

*Del Value.

*Get Data.

*Clone component.

The Get/Set/Del Full/Sparse Data and Get/Set Property functions handle the respective ForCES commands and return the respective TLV, for example the Set Full Data should return a Result TLV. The Get/Set/Del Value are called from the Get/Set/Del Full/Sparse Data respectively and provide the interface to the actual values in the hardware, separating the forces handling logic from the interface to the actual values.
The Get Data function should return the value of the data only, not in TLV format.
The last function seems out of place. That function must return a new component that has the exact same values and attributes. This function is useful in array components as described further.
The only requirement is to implement the base atomic data types. Any new atomic datatype can be built as a child of a base data type which will inherit all the functions and if necessary override them.
The struct component can then be built. A struct component is a component by itself, but consists of a number of atomic components. These atomic components create a static array within the struct. The ID of each atomic component is the array's index. The Clone function, for a struct component, must create and return an exact copy of the struct component with the same static array.
The most difficult component to be built is the array. The difficulty lies in the actual benefit of the model. You have absolute freedom over what you build. An array is an array of components. In all rows you have the exact same type of component either a single component or a struct. The struct can have multiple single components, or a combination of single components, structs and arrays and so on. So, the difficulty lies in how to create a new row, a new component by itself. This is where the Clone function is very useful. For the array a mother

component that can spawn new components exactly like itself is needed. Once a Set command is received, the mother component can spawn a new component, if the targeted row does not exists, and add it into the array, and with the Set Full Data the value is set in the recently spawned component, as the spawned component knows how the data is created. In order to distinguish these spawned components from each other and their functionality, some kind of index is required that will also reflect on how the actual data of the specific component is stored on the hardware.
Once the basic constructors of all possible components are created, then a developer only has to create his LFB components or datatypes as a child of one of the already created components and the only thing the developer really needs to add, is the three functions of Get/Set/Del value of each component which is platform dependent. The rest stays the same.

3.3.2. LFBs

The same architecture in the components can be used for the LFBs, allowing a developer to write LFB handling code only once. The parent LFB has some basic attributes:

*The LFB Class ID.

*The LFB Instance ID.

*An Array of Components.

*An Array of Capabilities.

*An Array of Events.

Then some common functions:

*Handle Configuration Command.

*Handle Query Command.

*Get Class ID.

*Get Instance ID.

Once these are created each LFB can inherit all these from the parent and the only thing it has to do is to add the components that have already been created.
An example can be seen in Figure 1. The following code creates a part of FEProtocolLFB:

```
//FEID
cui = new Component_uInt(FEPO_FEID, ACCESS_READ_ONLY, FE_id);
Components[cui->get_ComponentId()]=cui; //Add component to array list

//Current FEHB Policy Value
cub = new Component_uByte(FEPO_FEHBPolicy, ACCESS_READ_WRITE, 0);
Components[cub->get_ComponentId()]=cub; //Add component to array list

//FEIDs for BackupCEs Array
cui = new Component_uInt(0, ACCESS_READ_WRITE, 0);
ca = new Component_Array(FEPO_BackupCEs, ACCESS_READ_WRITE);
ca->AddRow(cui, 1);
ca->AddMotherComponent(cui);
Components[ca->get_ComponentId()]=ca; //Add component to array list
```

The same concept can be applied to handling LFBs as one FE. An FE is a collection of LFBs. Thus all LFBs can be stored in an array based on the LFB's class id, version and instance. Then what is required is an LFBHandler that will handle the array of the LFBs. A specific LFB, for example, can be addressed using the following scheme:
LFBs[ClassID][Version][InstanceID].
Note: While an array can be used in components, capabilities and events, a hash table or a similar concept is better suited for storing LFBs using the component ID as the hash key with linked lists for collision handling, as the created array can have large gaps if the values of LFB Class ID vary greatly.

3.4. Protocol

3.4.1. TLVs

The goal, for protocol handling, is to create a general and scalable architecture that handles all protocol messages instead of something implementation specific. There are certain difficulties that have to be overcome first.
Since the model allows a developer to define any LFB required, the protocol has been thus created to give the user the freedom to configure and query any component whatever the underlying model. While this being a strong point for the protocol itself, one difficulty lies with the unkwown underlying model and the unlimited number of types of messages that can be created, making creating generic code a daunting task.
Additionally the protocol also allows two different path approaches to LFB components and the CE or FE must handle both or even a mix of them making a generic decoding of the protocol message difficult.
Another difficulty also arises from the batching capabilities of the protocol. You can have multiple Operations within a message, you can select more than one LFB to command, and more than one component to manipulate.

Possible solution is again provided by inheritance. There are two basic components in a protocol message.

1. The common header.

2. The rest of the message.

The rest of the message is divided in Type-Length-Value (TLV) units, and in one case Index-Length-Value (ILV) units.
The TLV hierarchy can be seen in the Figure 2:

```
         Common Header
               |
   +-----------+----------+-----------+
   |           |          |           |
Redirect   LFBSelect   ASResult   ASTreason
  TLV         TLV        TLV         TLV
               |
               |
           Operation
              TLV
               |
               |            Optional
            PathData  ---> KeyInfo TLV
              TLV
               |
    +----------+---------+----------+
    |          |         |          |
SparseData   Result   FullData   PathData
   TLV        TLV       TLV        TLV
```

The above figure shows only the basic hierarchy level of TLVs and does not show batching. Also this figure does not show the recursion that can occur at the last level of the hierarchy. The figure shows one kind of recursion with PathData within PathData. FullData can be within FullData and SparseData. The possible combination of TLVs are described in detail in the Forwarding and Control Element Separation Protocol *[RFC5810]* as well as the data packing rules.
A TLV's main attributes are:

*Type

*Length

*Data

*An array of TLVs.

The array of TLVs is the next hierarchy level of TLVs nested in this TLV.

A TLVs common function could be:

*A basic constructor.

*A constructor using data from the wire.

*Add a new TLV for next level.

*Get the next TLV of next level.

*Get a specific TLV of next level.

*Replace a TLV of next level.

*Get the Data.

*Get the Length.

*Set the Data.

*Set the Length.

*Set the Type.

*Serialize the header.

*Serialize the TLV to be written on the wire.

All TLVs inherit these functions and attributes and either override them or create new where it is required.

3.4.2. Message Deserialization

What follows is a the algorithm for deserializing any protocol message:

1. Get the message header.

2. Read the length.

3. Check the message type to understand what kind of message this is.

4. If the length is larger than the message header then there is data for this message.

5. A check can be made here regarding the message type and the length of the message

If the message is a Query or Config type then for this level there are LFBSelector TLVs:

1. Read the next 2 shorts(type-length). If the type is an LFBSelector then the message is valid.

2. Read the necessary length for this LFBSelector and create the LFBSelector from the data of the wire.

3. Add this LFBSelector to the main header array of LFBSelectors

4. Repeat all above steps until the rest of the message has finished.

The next level of TLVs are Operation TLVs

1. Read the next 2 shorts(type-length). If the type is an OperationTLV then the message is valid.

2. Read the necessary length for this OperationTLV and create the OperationTLV from the data of the wire.

3. Add this OperationTLV to the LFBSelector array of TLVs.

4. Do this until the rest of the LFBSelector TLV has finished.

The next level of TLVs are PathData TLVs

1. Read the next 2 shorts(type-length). If the type is a PathData then the message is valid.

2. Read the necessary length for this PathDataTLV and create the PathDataTLV from the data of the wire.

3. Add this PathData TLV to the Operation TLV's array of TLVs.

4. Do this until the rest of the Operation TLV is finished.

Here it gets interesting, as the next level of PathDataTLVs can be either:

*PathData TLVs.

*FullData TLV.

*SparseData TLV.

*Result TLV.

The solution to this difficulty is recursion. If the next TLV is PathDataTLV then the PathDataTLV that is created uses the same kind of deserialisation until it reaches a FullDataTLV or SparseDataTLV. There can be only one FullDataTLV or SparseData within a PathData.

1. Read the next 2 shorts(type-length).

2. If the Type is a PathDataTLV then do again the previous algorithm but add the PathDataTLV to this PathDataTLV's array of TLVs.

3. Do this until the rest of the PathData TVL is finished.

4. If the Type is a FullDataTLV then create the FullData TLV from the message and add this to the PathData's array of TLVs.

5. If the Type is a SparseDataTLV then create the SparseData TLV from the message and add this to the PathData's array of TLVs.

6. If the Type is a ResultTLV then create the Result TLV from the message and add this to the PathData's array of TLVs.

If the message is a Query it must not have any kind of data inside the PathData.
If the message is a Query Response then it must either have a ResultTLV or a FullData TLV.
If the message is a Config it must contain either a FullDataTLV or a SparseData TLV.
If the message is a Config Reponse, it must contain a ResultTLV.
More details regarding message validation can be be read in Section 7 of the Forwarding and Control Element Separation Protocol *[RFC5810]*.
Note: When deserializing, implementors must take care to ignore padding of TLVs as all must be 32-bit aligned. The length value in TLVs includes the Type and Length (4 bytes) but does not include padding.

3.4.3. **Message Serialization**

The same concept can be applied in the message creation process. Having the TLVs ready, a developer can go bottom up. All that is required is the serialization function that will transform the TLV into bytes ready to be transfered on the network.
For example for the creation of a simple query from the CE to the FE, all the PathData are created. Then they will be serialized and inserted into an Operation TLV, which in turn will be serialized and inserted into an LFB Selector and in turn serialized and entered into the Common Header which will be passed to the TML to be transported to the FE.
Having an array of TLVs inside a TLV that are next in the TLV hierarchy, allows the developer to insert any number of next level TLVs thus creating any kind of message.
Note: When the TLV is serialized to be written on the wire, implementors must take care to include padding to TLVs as all must be 32-bit aligned.

4. Development Platforms

Any development platform that can support the SCTP TML and the TML of the developer's choosing is available for use.
Figure 3 provides an initial survey of SCTP support for C/C++ and Java at present time.

Platform / Language	Windows	Linux	Solaris
C/C++	Supported	Supported	Supported
Java	Limited Third Party Not from SUN	Supported	Supported

A developer should be aware of some limitations regarding Java implementations.
Java inherently does not support unsigned types. A workaround this can be found in the creation of classes that do the translation of unsigned to java types. The problem is that the unsigned long cannot be used as is in the Java platform. The proposed set of classes can be found in *[Java Unsigned Types]*.

5. Acknowledgements

The authors would like to thank Adrian Farrel for sponsoring this document and Jamal Hadi Salim for discussions to make this document better.

6. IANA Considerations

This memo includes no request to IANA.

7. Security Considerations

Developers of ForCES FEs and CEs must take the security considerations of the Forwarding and Control Element Separation Framework *[RFC3746]* and the Forwarding and Control Element Separation Protocol *[RFC5810]* into account.
Also, as specified in the security considerations section of the SCTP-Based Transport Mapping Layer (TML) for the Forwarding and Control Element Separation Protocol *[RFC5811]* the transport-level security, has to be ensured by IPsec.

8. References

8.1. Normative References

[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.
[RFC6041]	Crouch, A., Khosravi, H., Doria, A., Wang, X. and K. Ogawa, "Forwarding and Control Element Separation (ForCES) Applicability Statement", RFC 6041, October 2010.
[RFC6053]	Haleplidis, E., Ogawa, K., Wang, W. and J. Hadi Salim, "Implementation Report for Forwarding and Control Element Separation (ForCES)", RFC 6053, November 2010.

8.2. Informative References

, "

[RFC3654]	Khosravi, H. and T. Anderson, "Requirements for Separation of IP Control and Forwarding", RFC 3654, November 2003.
[RFC3746]	Yang, L., Dantu, R., Anderson, T. and R. Gopal, "Forwarding and Control Element Separation (ForCES) Framework", RFC 3746, April 2004.
[Java Unsigned Types]	Classes that support unsigned primitive types for Java. All except the unsigned long", .

Authors' Addresses

Evangelos Haleplidis Haleplidis University of Patras Patras, Greece EMail: ehalep@ece.upatras.gr

Odysseas Koufopavlou Koufopavlou University of Patras Patras, Greece EMail: odysseas@ece.upatras.gr

Spyros Denazis Denazis University of Patras Patras, Greece EMail: sdena@upatras.gr