Internet Engineering Task Force
Internet-Draft
Intended status: Informational
Expires: April 11, 2010

E. Haleplidis
O. Koufopavlou
S. Denazis
University of Patras

October 8, 2009

ForCES Implementation Experience Draft
draft-ietf-forces-implementation-experience-00

Status of this Memo

Copyright Notice

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. It's aim is to help others by providing examples and possible strategies for implementing the ForCES protocol.

Table of Contents

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].

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 Function Block (LFB) configuration information, association setup, status, and event notifications, etc. The reader is encouraged to read FE-protocol [I-D.ietf-forces-protocol] for further information.

The FE-MODEL [I-D.ietf-forces-model] presents a formal way to define FE Logical Function Blocks (LFBs) using XML. LFB configuration components, capabilities, and associated events are defined when the LFB is formally created. The LFBs within the FE are accordingly controlled in a standardized way by the ForCES protocol.

The TML transports the PL 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 [I-D.ietf-forces-sctptml].

2.1. Document Goal

This document captures the experience of implementing the ForCES protocol and model and it's 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 document, the FE-protocol [I-D.ietf-forces-protocol], the FE-MODEL [I-D.ietf-forces-model] and

the SCTP-TML [I-D.ietf-forces-sctptml].

2.2. Definitions

This document follows the terminology defined by the ForCES Requirements in [RFC3654] and by the ForCES framework in [RFC3746].

3. ForCES Architecture

This section discusses the ForCES architecture, difficulties and how to overcome them.

3.1. Pre-association setup - Initial Configuration

The initial configuration of the FE and the CE respectively is done by the FE Manager and the CE Manager. These entities has not as yet been standardized and anyone can build their own. It is expected that somehow they will talk to each other and exchange details regarding the upcoming associations. Any developer can create any Manager, but at least they should be able to exchange the following details:

From the FE side:

1. FEID

2. FE IP, if your FE and CE will be communicating via network.

3. TML. The TML that you will be using. If you skip this, then SCTP MUST be chosen as default from the CE side.

4. Priority ports. If you also skip this, then the CE MUST use the default ones from the respective TML rfc.

From the CE side:

1. CEID

2. CE IP, if your FE and CE will be communicating via network.

3. TML. The TML that you will be using. If you skip this, then SCTP MUST be chosen as default from the FE side.

4. Priority ports. If you also skip this, then the FE MUST use the default ones 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 draft. For SCTP you have to use IPSec.

2. NAT issues. ForCES can be deployed everywhere and can run over SCTP/IP. If you are behind a NAT, you must forward the TML priority ports to your CE who listens at these ports. The issue is what happens to the ports that the FE uses. Unless you bind the same port numbers to the FE, your FE would have a random port

number. This means that port forwarding to the FE must include all sctp ports. The CE can be easily behind a NAT since it will bind the specific sctp ports. In order for the FE to work, it must either have a global IP, or it must bind specific ports that are forwarded to it unless all sctp ports are forwarded to the FE.

3.3. Model

The model is very dynamic and can be used to model anything. Using the basic atomic values that are defined, new datatypes can be built using atomic (single valued) and/or compound (structures and arrays).

The difficulty is to create something that is completely scalable so a develeper doesn't 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.

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 of variable length.
- If it is an optional component.
- The size of its data.

Next, some basic functions are in order:

- A common constructor.

- A common deconstructor.

- 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 Hardware Value.
- Set Hardware Value.
- Del Hardware Value.
- Get Data.
- Clone component.

While almost all functions are logical, 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 especially useful in array components.

Now any atomic datatype can be built as a child of that basic component which will inherit all the functions and if necessary override the mother's functions.

Next struct components can be built. A struct component is a component itself, but contains an array of basic components. The ID of the component is the array index. The Clone function must create and return a similar struct component.

The most difficult component to be built is the array. The difficulty lie 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 specific struct. The struct can have multiple basic components, or a combination of basic components, structs and arrays and so on. So, the difficulty lies in how do to create a new row since the array is very dynamic. This is where the clone function is very useful. For the array you need a mother component. Once a set command is received, the mother component can spawn a new component and adds them into the array, and with the set fulldata the value is set in the recently spawned component, as the spawned component knows how the data is created.

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 hardware value of each component. The rest is the same.

3.3.2. LFBs

The same architecture in the components can be used for the LFBs. The parent LFB has some basic attributes:

- The LFB Class ID.
- The LFB Instance ID.
- An Array of Components.

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 of this is the following. The code next creates a part of FEProtocolLFB:

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

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

//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 BackupCEs Array component
```

Then all it is required is an LFBHandler that will have an array of all the LFBs:

LFBs[ClassID][InstanceID][LFB].

3.4. Protocol

3.4.1. TLVs

Since the model is so free to create anything the developer needs, the protocol itself has been created thus to provide the user the freedom to manipulate any component. This creates some difficulties in developing a scalable architecture for handling the protocol messages.

Another difficulty 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.

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

1. A main header.

2. The rest of the packet.

The rest of the packet is divided in Type-Length-Value (TLV) packets, and in one case Index-Length-Value (ILV) packets.

The possible TLVs and ILVs the are described in detail in the forces protocol rfc.

A TLV's main attributes are:

- Type
- Length
- Data
- An array of TLVs.

The array of TLVs is the next level of TLVs for 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.

Next all TLVs can inherit all these functions and attributes and either override them or create some new functions for each.

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 packet this is.
4. If the length is larger than the message header then there is data for this packet.
5. A check can be made here regarding the message type and the length of the packet

3.4.2.1. Config or Query

If the packet 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==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 mainheader array of LFBSelectors

4. Do this until the rest of the packet has finished.

The next level of TLVs are Operation TLVs

1. Read the next 2 shorts(type-length). If the type==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's Packet has finished.

1. Read the next 2 shorts(type-length). If the type==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 OperationTLV's Packet has finished.

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

- PathData TLVs.
- FullData TLV.
- SparseData TLV.
- Result TLVs

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

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

2. If the Type==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's Packet has finished.

4. If the Type==FullDataTLV create the FullData TLV from the packet and add this to the PathData's array of TLVs

5. If the Type==SparseDataTLV create the SparseData TLV from the packet and add this to the PathData's array of TLVs

6. If the Type==ResultTLV create the Result TLV from the packet 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 have inside either a FullDataTLV or a SparseData TLV.

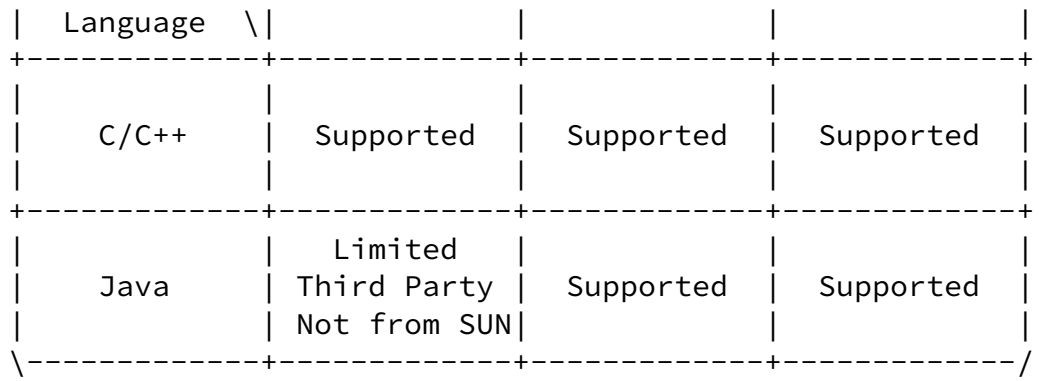
If the message is a Config Reponse, it MUST have inside a ResultTLV

4. Developement Platforms

Any Developmens platfor that can support the SCTP TML and the TML of the developer's choosing is available for use.

The next table provides an initial survey of sctp support for C/C++ and Java.

Platform	Windows	Linux	Solaris



A developer should keep some limitations regarding Java.

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 it is in the Java platform. The proposed set of classes can be found in [Java Unsigned Types].

5. Acknowledgements

TBA

6. IANA Considerations

This memo includes no request to IANA.

7. Security Considerations

The security considerations of the ForCES framework in [RFC3746] and FE-protocol [I-D.ietf-forces-protocol] are applicable in this document. Implementers or users of ForCES FEs and CEs should take these considerations into account.

Also, as specified in the security considerations section of the SCTP-TML draft [I-D.ietf-forces-sctptml] the transport-level security, has to be ensured by TLS (Transport Layer Security).

8. References

8.1. Normative References

[I-D.ietf-forces-model]
Halpern, J. and J. Salim, "ForCES Forwarding Element Model", draft-ietf-forces-model-16 (work in progress), October 2008.

[I-D.ietf-forces-protocol]
Dong, L., Doria, A., Gopal, R., HAAS, R., Salim, J., Khosravi, H., and W. Wang, "ForCES Protocol Specification", draft-ietf-forces-protocol-21 (work in progress), February 2009.

[I-D.ietf-forces-sctptml]
Salim, J. and K. Ogawa, "SCTP based TML (Transport Mapping Layer) for ForCES protocol", draft-ietf-forces-sctptml-05 (work in progress), August 2009.

8.2. Informative References

[Java Unsigned Types]
"Classes that support unsigned primitive types for Java. All except the unsigned long", <http://darksleep.com/player/JavaAndUnsignedTypes.html>.

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

[RFC2629] Rose, M., "Writing I-Ds and RFCs using XML", RFC 2629, June 1999.

[RFC3552] Rescorla, E. and B. Korver, "Guidelines for Writing RFC Text on Security Considerations", BCP 72, RFC 3552, July 2003.

[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.

[RFC5226] Narten, T. and H. Alvestrand, "Guidelines for Writing an IANA Considerations Section in RFCs", BCP 26, RFC 5226, May 2008.

Authors' Addresses

Evangelos Haleplidis
University of Patras
Patras,
Greece

Email: ehalep@ece.upatras.gr

Odysseas Koufopavlou
University of Patras
Patras,
Greece

Email: odysseas@ece.upatras.gr

Spyros Denazis
University of Patras
Patras,
Greece

Email: sdena@upatras.gr