View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by Aaltodoc Publication Archive

Jan Gröndahl

Implementation and Evaluation of a Network Element Control Protocol

Faculty of Electronics, Communications and Automation

Thesis submitted for examination for the degree of Master of Science in Technology.

Espoo 24.5.2010

Thesis supervisor:

Professor Andrei Gurtov

Thesis instructor:

M.Sc. (Tech.) Olli-Pekka Lamminen

Aalto University
School of Science
and Technology

AALTO UNIVERSITY
SCHOOL OF SCIENCE AND TECHNOLOGY

ABSTRACT OF THE
MASTER'S THESIS

Author: Jan Gröndahl

Title: Implementation and Evaluation of a Network Element Control Protocol

Date: 24.5.2010 Language: English Number of pages:11+77

Faculty of Electronics, Communications and Automation

Department of Communications and Networking

Professorship: Data communications software Code: T-109/T-110

Supervisor: Professor Andrei Gurtov

Instructor: M.Sc. (Tech.) Olli-Pekka Lamminen

This thesis consists of the implementation and evaluation of a network element control protocol that is used for the communication between the control element and the forwarding element in a network element in a network operator's core network. The author has investigated three commonly used dynamic switch control protocols and one of them is chosen to be implemented in this study. This protocol is the forwarding and control element separation (ForCES).

In the EU 7th framework program's ETNA project, there has been done some modifications to the IETF specified ForCES protocol: the PATH-DATA-TLV layer was removed among other type-length-value (TLV) data structure modifications. ETNA has also added extra IDs, data types, event and result codes, and other values choosing them so that they do not overlap with the values in the IETF ForCES specification.

The implementation done in this study is a C++ library with 31 classes and 15 436 lines of source code. The code was tested with a test program written by the author and is working without any known bugs.

The evaluation part of the study consists of memory allocation and message construction time performance measurements. The test messages with one kilobyte length and 185 TLVs allocated over 500 % extra memory compared to the message network length. However, this overhead is proportional to the network length of the message and will decrease as the length of the TLVs increase. The processing time for the message construction was linearly increasing, but there was some offset time at the beginning of the message processing especially with short messages with a few TLVs.

In Aalto University, we designed and implemented the control element implementation of the ForCES protocol and the Ben Gurion University was responsible for the design and implementation of the forwarding element. Both implementations were integrated together and there was demonstrated a new co-developed proof-of-concept core network model.

Keywords: network element, control element, ForCES, Ethernet, carrier grade, Internet

AALTO-YLIOPISTO
TEKNILLINEN KORKEAKOULU

DIPLOMITYÖN
TIIVISTELMÄ

Tekijä: Jan Gröndahl

Työn nimi: Erään verkkoelementin ohjausprotokollan toteuttaminen ja sen arviointi

Päivämäärä: 24.5.2010 Kieli: Englanti Sivumäärä:11+77

Elektroniikan, tietoliikenteen ja automaation tiedekunta

Tietoliikenne- ja tietoverkkotekniikan laitos

Professuuri: Tietoliikenneohjelmistot Koodi: T-109/T-110

Valvoja: Professori Andrei Gurtov

Ohjaaja: DI Olli-Pekka Lamminen

Tämä opinnäytetyö koostuu erään ohjaus- ja välityselementtien väliseen tiedonsiirtoon käytettävän verkkoelementin ohjausprotokollan toteutuksesta ja arvioinnista. Tätä ohjausprotokollaa käytetään verkkoelementin sisällä verkko-operaattorin runkoverkossa. Kirjoittaja on tutkinut kolmea yleisesti käytössä olevaa dynaamista verkkokytkimen ohjausprotokollaa ja yksi näistä on valittu toteutettavaksi tässä tutkimuksessa. Tämä kyseinen protokolla on forwarding and control element separation (ForCES).

Euroopan Unionin seitsemännen puiteohjelman ETNA-projektissa on tehty IETF:n määrittelystä poikkeavia muutoksia ForCES-protokollaan: PATH-DATA-TLV-kerros poistettiin sekä muita TLV-tietorakenteita muutettiin projektin tarpeiden mukaisesti. ETNA on myös lisännyt ylimääräisiä ID-arvoja, tietotyyppejä, tapahtuma- ja tuloskoodeja sekä muita arvoja. Arvot on valittu niin, etteivät ne mene päällekäin IETF:n ForCES määrittelyjen kanssa.

Tässä tutkimuksessa tehty ohjelmistototeutus on C++-kirjasto, jossa on 31 luokkaa ja 15 436 riviä lähdekoodia. Ohjelmistokoodi on testattu kirjoittajan tekemällä testiohjelmalla ja se toimii ilman tunnettuja virhetoimintoja.

Tutkimuksen arviointiosuus koostuu muistinvarauksen ja viestin muodostusajan mittaamisesta. Kilotavun mittaiset ja 185 TLV-tietorakennetta sisältävät testiviestit varasivat yli 500 % ylimääräistä muistia verrattuna viesten nettopituuteen. Tämän ylimääräisen muistinkäytön suhteellinen osuus viestien nettopituuteen verrattuna kuitenkin pienenee, samalla kun TLV-tietorakenteiden pituus kasvaa. Viestinmuodostuksen käsittelyaika kasvoi lineaarisesti, mutta mittauksissa havaittiin viestien käsittelyn alussa jonkin verran ylimääräistä käsittelyaikaa etenkin lyhyillä viesteillä, joissa oli vähän TLV-tietorakenteita.

Suunnittelimme ja toteutimme Aalto-yliopistossa ohjauselementin ForCES-protokollan toteutuksen ja Ben Gurionin yliopisto Israelissa oli vastuussa välityselementin suunnittelusta ja toteutuksesta. Molemmat toteutukset yhdistettiin ja demonstroitiin yhdessä kehitettyä uutta runkoverkon mallia.

Avainsanat: verkkoelementti, ohjauselementti, ForCES, Ethernet, operaattoritasoinen, Internet

Preface

This Master's thesis has been done at the Department of Communications and Networking in Aalto University School of Technology, Finland. The work was carried out as a part of the Ethernet Transport Networks, Architectures of Networking (ETNA) project which was co-funded by the European Commission in the seventh EU framework program of research and technological development (FP7).

I would like to thank my supervisor, professor Andrei Gurtov and my instructor, M.Sc. (Tech.) Olli-Pekka Lamminen for their contribution to this Master's thesis. Thanks also to William Martin for proofreading the final version of the manuscript.

I thank the Department of Communications and Networking for provided funding and a work environment and also my colleagues in ETNA for a nice co-working atmosphere. I also thank my family for their support and ideas on this thesis.

Otaniemi, 24.5.2010

Jan Gröndahl

Contents

Abstract ii

Abstract (in Finnish) iii

Preface iv

Contents v

Abbreviations viii

List of Figures x

List of Tables xi

1 Introduction to the Study **1**

1.1 Background 1

1.1.1 Ethernet Transport Networks, Architectures of Networking . . 2

1.1.2 ETNA Network Model 2

1.1.3 Network Element 3

1.2 Scope and Structure of the Thesis 3

1.2.1 Scope 3

1.2.2 Structure 4

1.3 Key Terminology 4

2 Dynamic Switch Control Protocols **6**

2.1 Network Switch Constructions 6

2.2 Protocols Selected for Investigation in the Study 6

2.2.1 Forwarding and Control Element Separation Protocol 7

2.2.2 General Switch Management Protocol 9

2.2.3 OpenFlow Protocol 10

2.3 Comparison of the Protocols 11

2.4 Summary and Conclusions 12

3 ForCES Protocol Details 13
3.1 Message Flows . . . 13
3.1.1 Association Setup State . . . 13
3.1.2 Association Established State . . . 14
3.2 Message Encapsulation and Structure . . . 14
3.2.1 Common Header . . . 15
3.2.2 Type-Length-Value Data Structure . . . 17
3.2.3 Identifier-Length-Value Data Structure . . . 19
3.3 ForCES Messages . . . 19
3.3.1 Association Messages . . . 20
3.3.2 Configuration Messages . . . 22
3.3.3 Query Messages . . . 23
3.3.4 Event Notification Message . . . 24
3.3.5 Packet Redirect Message . . . 24
3.3.6 Heartbeat Message . . . 25
3.4 Summary and Conclusions . . . 25

4 ETNA Requirements 26
4.1 Network Element Architecture Overview . . . 26
4.2 Requirements for the ForCES Messages . . . 27
4.3 Summary and Conclusions . . . 28

5 Software Implementation of the ForCES Protocol 30
5.1 About the Implementation . . . 30
5.2 Code Structure . . . 31
5.3 Class Inheritance . . . 32
5.4 Library Use . . . 32
5.4.1 ForCES Message Construction . . . 33
5.4.2 ForCES Message Destruction . . . 36
5.4.3 Object Member Variable Accessing Functions . . . 37
5.4.4 Object Validation . . . 38
5.4.5 Object Length Calculation . . . 38
5.4.6 ForCES Message Printing . . . 39
5.5 Development Environment . . . 39

5.5.1 Platform . . . 39
5.5.2 Testing and Debugging . . . 40
5.6 Coding process . . . 41
5.7 Summary and Conclusions . . . 41

6 Evaluation of the Implementation 43
6.1 Evaluation Methods . . . 43
6.2 Performance Measurements . . . 43
6.2.1 Planning of the Measurements . . . 44
6.2.2 Tools Used in the Measurements . . . 44
6.2.3 Measurement Environment . . . 45
6.2.4 Measurements with ForCES Test Message 1 . . . 45
6.2.5 Measurements with ForCES Test Message 2 . . . 46
6.2.6 Measurements with ForCES Test Message 3 . . . 47
6.2.7 Measurements with ForCES Test Message 4 . . . 48
6.3 Results of the Measurements . . . 49
6.3.1 Analysis of the Memory Allocations . . . 49
6.3.2 Analysis of the Message Construction Times . . . 50
6.4 Summary and Conclusions . . . 52

7 Summary and Conclusions of the Study 54
7.1 Summary . . . 54
7.2 Conclusions . . . 55
7.3 Future Work . . . 55

References 56

Appendices 59

A Code Snippets 59

B Test Runs 65

C Numerical Values Used in ETNA 67

D Measurements 72

Abbreviations

2PC	Two Phase Commit
ACK	Acknowledged
API	Application Programming Interface
ASIC	Application Specific Integrated Circuit
BGU	Ben Gurion University of the Negev, Israel
CE	Control Element
CP	Control Plane
DP	Data Plane
ETNA	Ethernet Transport Networks, Architectures of Networking
FDB	Forwarding Database
FE	Forwarding Element
ForCES	Forwarding and Control Element Separation
GSMP	General Switch Management Protocol
HB	Heartbeat
IEEE	Institute of Electrical and Electronics Engineers
IETF	Internet Engineering Task Force
IF	Interface
ILV	Identifier-Length-Value
IP	Internet Protocol
IPv4	Internet Protocol version 4
IPv6	Internet Protocol version 6
ITU	International Telecommunication Union
ITU-D	ITU, Telecommunication Development Sector
LFB	Logical Function Block
MP	Management Plane
MPLS	Multiprotocol Label Switching
mRSVP	Management Resource Reservation Protocol
MS	Management System
NE	Network Element
OAM	Operations, Administration and Maintenance
OOP	Object-Oriented Programming
PDU	Protocol Data Unit
PL	Protocol Layer
QoS	Quality of Service
RFC	Request For Comments

RSVP	Resource Reservation Protocol
RSVP-TE	Resource Reservation Protocol – Traffic Engineering
RSVP-TEEth	Resource Reservation Protocol – Traffic Engineering for Ethernet
SOAP	SOAP (former Simple Object Access Protocol)
TCP	Transmission Control Protocol
TLV	Type-Length-Value
TML	Transport Mapping Layer
VoD	Video-on-Demand
VoIP	Voice over Internet Protocol
WP	Work Package
WWW	World Wide Web
XML	Extensible Markup Language

List of Figures

Figure 1.1 Layers of the ETNA network model. 3

Figure 2.1 Classical network switch architecture. 7

Figure 2.2 Architecture with control and forwarding logic separated. . . . 8

Figure 2.3 ForCES architecture. 9

Figure 3.1 An example of establishing an NE association. 14

Figure 3.2 An example of message exchange during steady state. 15

Figure 3.3 Common header layout. 16

Figure 3.4 Type-Length-Value (TLV) data structure layout. 17

Figure 3.5 An example of TLV encapsulation with a set of sub-TLVs. . . . 18

Figure 3.6 LFBselect TLV layout. 18

Figure 3.7 Identifier-Length-Value (ILV) data structure layout. 19

Figure 4.1 Network element structure with layer separation. 27

Figure 5.1 TLV class inheritance. 32

Figure 6.1 ForCES test messages construction time comparison. 50

Figure 6.2 ForCES test messages construction time comparison (zoomed). 51

Figure 6.3 ForCES test message 2 construction times with 1–5 messages. . 52

List of Tables

Table 3.1 ForCES message types. 20

Table 4.1 ForCES message types used in ETNA. 28

Table 6.1 ForCES test message 1 measurements with non-optimized code. 46
Table 6.2 ForCES test message 1 measurements with optimized code. . . 46
Table 6.3 ForCES test message 2 measurements with non-optimized code. 46
Table 6.4 ForCES test message 2 measurements with optimized code. . . 47
Table 6.5 ForCES test message 3 measurements with non-optimized code. 47
Table 6.6 ForCES test message 3 measurements with optimized code. . . 48
Table 6.7 ForCES test message 4 measurements with non-optimized code. 48
Table 6.8 ForCES test message 4 measurements with optimized code. . . 48
Table 6.9 Comparison between message length and memory allocated. . . 49
Table 6.10 Calculated ForCES test message construction time offsets. . . . 51

Chapter 1

Introduction to the Study

In this first chapter the general background needed to understand the subject of this thesis is explained and the project in which this thesis has been done is described. The second section describes the scope and structure of the thesis. The essential key terminology is listed in the third section.

1.1 Background

The capacity of the current Internet is rapidly becoming insufficient to cater for the vast amount of network traffic and the number of Internet users which is increasing all the time [1] [2]. Scalability and robustness of the network are growing in importance. There are many reasons for this growth, for example, new and future services such as video-on-demand (VoD), real-time services such as Internet protocol television (IPTV) and voice over Internet protocol (VoIP) based services. Moreover, the higher resolution of movies and pictures, as technology makes it possible to use higher resolution recordings, further adds to the problems of capacity limitations.

Traditional Internet protocol (IP) [3, p. 572] networks work well when there are not many users using resources at the same time. IP was not designed to meet the quality-of-service requirements of, for example, real-time voice and high-bandwidth video and it does not include extensive monitoring capabilities [4].

While the amount of the network traffic has been rising, Ethernet [5, pp. 292–295] technology has consolidated its position in homes and the prices of network hardware have come down as the amount of manufactured devices has increased. Also the quality and reliability of Ethernet technology is excellent today. This is why Ethernet technology can be a solution to the capacity problems outlined above on the Internet, if Ethernet will be utilized in the Internet service providers' core networks [6].

1.1.1 Ethernet Transport Networks, Architectures of Networking

The Ethernet Transport Networks, Architectures of Networking (ETNA) project is looking for a solution to this existing problem by designing and analysing future metro and core networks based on Ethernet technology. The goal of the project is to design and implement a working prototype of a low cost European Ethernet transport network that can serve millions of subscribers. The designed network is aimed to be a common and secure transport architecture for different network services that are in use now, and also to take into consideration future services that have not yet been implemented.

ETNA was carried out in the years 2008 and 2009, and was co-funded by the European Commission in the Seventh EU Framework Program of Research and Technological Development (FP7). During the project solutions have been made that have been fixed and implemented by partners from universities and notable telecommunications companies and operators [7].

In the Aalto University School of Science and Technology (former Helsinki University of Technology), we have implemented a portion of a working demonstration of a core network based on Ethernet transport. Generally known protocols are used for packet transport and tunnel operations adapted to fit our implementation. In this network model, a network element can be divided into a control element and a forwarding element. Our task has been designing and implementing the control element part and the Ben Gurion University of the Negev (BGU) has been responsible for the design and implementation of the forwarding element part. Both implementations have been integrated together and has been demonstrated [8, p. 7] the new co-developed core network model.

As a part of Aalto University implementation this study contains the implementation and evaluation of the protocol used between the control and forwarding element communication.

1.1.2 ETNA Network Model

The ETNA network model consist of three distinct layers (Figure 1.1). These layers are the transport layer, the transport services layer, and the value added services layer [9, p. 19].

The transport layer operates the packet transmission between the network endpoints. The functionality of this layer includes three distinct planes: a control plane (CP), a forwarding plane (FP), and a management plane (MP). Physically the transport layer consists of network elements (NE) that possess the functionality of all these three planes. The middle layer is the transport services layer that offers a set of services required to support customers of the transport network. These services utilize the basic functionalities of the transport layer such as tunnel creation. The value added services layer utilizes the services offered by the transport services layer

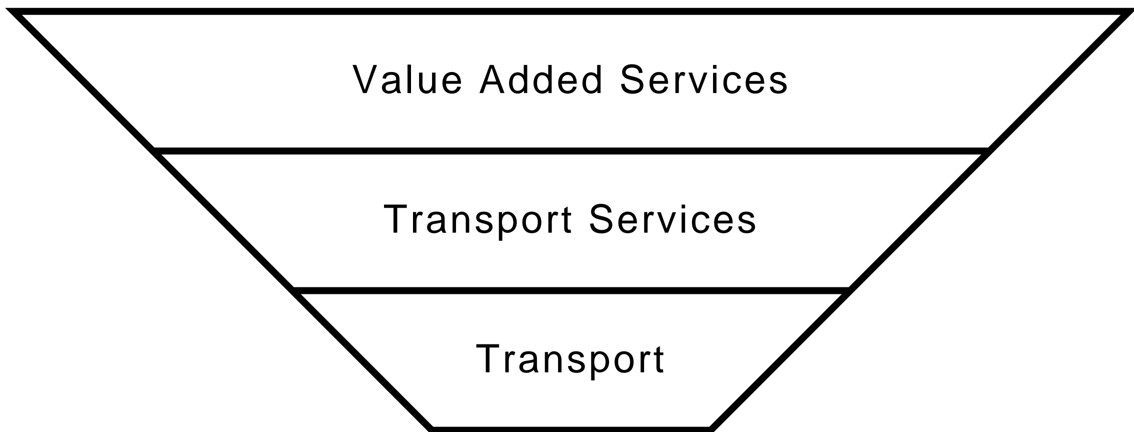

Figure 1.1: Layers of the ETNA network model [9, p. 19].

and implements such services as routing [10] [11] [12].

1.1.3 Network Element

A network element is a manageable logical entity that consists of one or more physical elements. To external entities it looks like one integrated network element. Network elements normally have two types of components: control plane components and forwarding plane components. Control plane components are usually based on general-purpose processors that provide control functionality, for example, processing routing or signaling protocols. In general, forwarding plane components are application specific integrated circuit (ASIC), network processor, or general-purpose processor based devices that handle all the datapath operations. Defining a standard set of ways for connecting these components provides significantly more scalability and allows the control and forwarding planes to evolve independently [13].

The software implemented in this study will be located in the control plane and is used to communicate with the forwarding plane via protocol messages.

1.2 Scope and Structure of the Thesis

In this section the scope and the structure of the thesis are described.

1.2.1 Scope

This thesis focuses on implementing and evaluating a switch control protocol for message transport between a control element and a forwarding element in a network element. One switch control protocol is chosen to be implemented and evaluated.

Two other protocols are also investigated to compare their features with the protocol chosen to be implemented and with each other. The original specification of the chosen switch control protocol is investigated and compared with the protocol specification that has been modified by the project. The actual implementation is done following the ETNA modified specification.

The implementation work includes implementing all the forwarding and control element separation (ForCES) messages and type-length-value (TLV) data structures in the ETNA modified switch control protocol specification. The needed classes and class member variables and functions are to be designed, implemented, and tested. The evaluation of the implementation includes performance measurements in message construction with selected test messages. The performance is evaluated by measuring the processing times of operations and memory usage during program execution. In addition to the measurements, the functioning of the implementation is evaluated.

1.2.2 Structure

In the first chapter, the reader is introduced to the topic and given some background. In the second chapter, the concept of the dynamic switch control protocol is explained and a selection of protocols and their features are discussed. The third chapter takes one of these protocols, the ForCES protocol, for deeper analysis.

The fourth chapter tells about ETNA architecture and requirements for the ForCES messages. The software implementation, that is how the code is designed and implemented, is explained in the fifth chapter. Testing and debugging is also covered in this chapter. The sixth chapter has the evaluation of the ForCES protocol implementation. The seventh and final chapter includes the conclusions and summary of the study.

1.3 Key Terminology

In this section the most important key terminology is explained to understand the principles behind the subject of this thesis.

Control Element

In a network switch model where control logic and forwarding logic are separated, a control element (CE) is the controlling part of a network element and is often implemented with some standard hardware platform and software running on that. Usually a control element is used to instruct one or more forwarding elements on how to process packets. A control element usually implements controlling and signaling protocols.

ForCES Message

A forwarding and control element separation (ForCES) protocol message is used between the CE and FE communication. A ForCES message has a common header which length is 24 bytes. After the header is the top-level TLV which can include one or many sub-TLVs.

Forwarding Element

A forwarding element (FE) is the forwarding part of a network element that can be implemented, for instance, with network processors for faster packet processing performance. The forwarding element provides packet processing and handling as controlled by one or more control elements.

Logical Function Block

A logical function block (LFB) is a well defined, logically separable functional block in a forwarding element and is controlled by the control element via the Forwarding and Control Element Separation (ForCES) protocol. A logical function block can, for instance, belong to the forwarding element datapath and process packets or it can be a separate control or configuration entity.

Network Element

A network element (NE) is a part of a network that is a manageable logical enity and has one or more physical components. In this thesis the network element is usually referred to as a network switch.

Operations, Administration and Maintenance

Operations, administration and maintenance (OAM) is a service or device on a network that is used for operating, administrating, managing and maintaining any system in that network. OAM can offer, for instance, troubleshooting and performance measuring sevices.

Type-Length-Value

A type-length-value (TLV) is a data structure that contains three fields: a 16-bit type field, a 16-bit length field, and a varying length value field. The Length of the TLV is calculated in bytes and includes also the type and the length fields as well as the value field. A TLV can have one or more sub-TLVs in its value field.

Chapter 2

Dynamic Switch Control Protocols

In the first section of this chapter there are described common network switch architectures as background knowledge. The second section analyzes protocols selected for investigation in the study and the third section compares them with each other. The fourth section presents the protocol chosen to be implemented in ETNA and the fifth and final section has the summary and conclusions for this chapter.

2.1 Network Switch Constructions

A classical network switch architecture has the control and forwarding logic parts both integrated as shown in Figure 2.1. The other model is to have the control logic and forwarding logic separated [14] so that only the forwarding logic is located in the switch and the control logic is placed in a separate controller that can be located in the same place as the switch or at a distance from it (Figure 2.2).

The forwarding logic can then be implemented with fast hardware solutions whereas the control logic can be implemented, for example, with some inexpensive general hardware platform and switch control software customized by the network operator involved. This makes it easier to modify the controller and add features. Further, by using modular design it is possible to improve the availability of the network switch [15]. In this thesis the control logic part is called a control element (CE) and, correspondingly, the forwarding logic part is called a forwarding element (FE).

2.2 Protocols Selected for Investigation in the Study

There are many protocols available for dynamic switch controlling. A few of them are explained in this chapter. The protocols explained here are the Forwarding and Control Element Separation (ForCES) protocol, the General Switch Management Protocol (GSMP) and the OpenFlow protocol. These were chosen because the

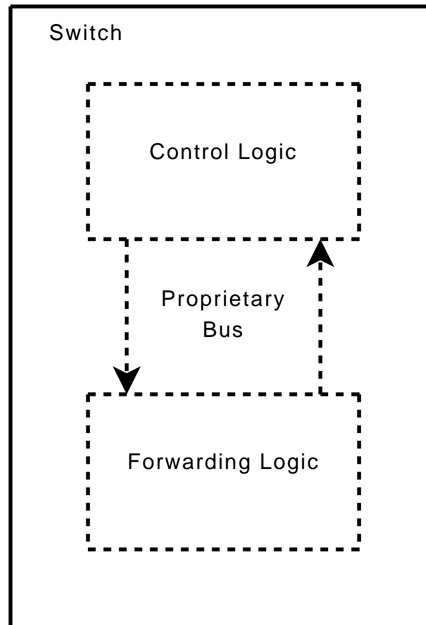

Figure 2.1: Classical network switch architecture.

ForCES protocol was decided to be utilized in ETNA and the other two protocols were chosen to be investigated here also to compare their features with each other. Both ForCES and GSMP have been available for several years while OpenFlow has been released more recently.

2.2.1 Forwarding and Control Element Separation Protocol

The Forwarding and Control Element Separation (ForCES) protocol is defined in the IETF Network Working Group Internet-Draft *ForCES Protocol Specification* [16]. The requirements of the ForCES protocol are defined in Request for Comments (RFC) 3654 [13]. The ForCES architectural framework is defined in RFC 3746 [17]. In addition to the framework, there are defined associated protocols [18] to standardize information exchange between the control plane and the forwarding plane in a ForCES network element (NE). The ForCES protocol works in master-slave mode in which the forwarding elements (FE) are slaves and the control elements (CE) are masters.

A control element controls a forwarding element [19] by sending specific messages defined in the ForCES protocol specification. The FE can respond to the received messages, if the protocol allows responding to that message type. Because the control element is a master, it normally initiates the messaging, but in some situations the forwarding element can also send messages first. These situations are discussed later in this chapter.

There are six different kinds of messages available: association messages, con-

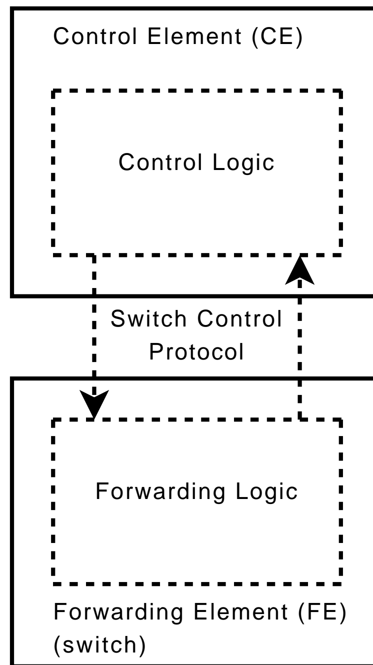

Figure 2.2: A network switch architecture with control and forwarding logic separated.

figuration change messages, messages to query FE part configurations, event notification messages, messages for packet redirecting and heartbeat signaling messages. Association messages are used to make associations between CEs and FEs and to teardown them when associations are not needed anymore. Configuration messages are used by CE to set new configurations to FE part. Query messages are used also by CE to query current configuration settings on FE. Event Notification messages are used when FE notices status changes in the network or FE's operating state. Heartbeat messages are used by one ForCES element to notify other ForCES elements of its existence.

Protocol Framework

The ForCES protocol architecture has two layers: the protocol layer (PL) and the transport mapping layer (TML). The protocol layer defines the ForCES protocol messages, the protocol state transfer scheme, and the ForCES protocol architecture itself. The transport mapping layer uses the capabilities of existing transport protocols to specifically address protocol message transportation issues such as how the protocol messages are transported through the network using different transport media, such as TCP, IPv4, IPv6, ATM [20] or Ethernet. In addition to this, the TML layer implements reliability, multicast, and ordering of protocol message transport. Both the PL and TML are defined in the ForCES protocol specification [16].

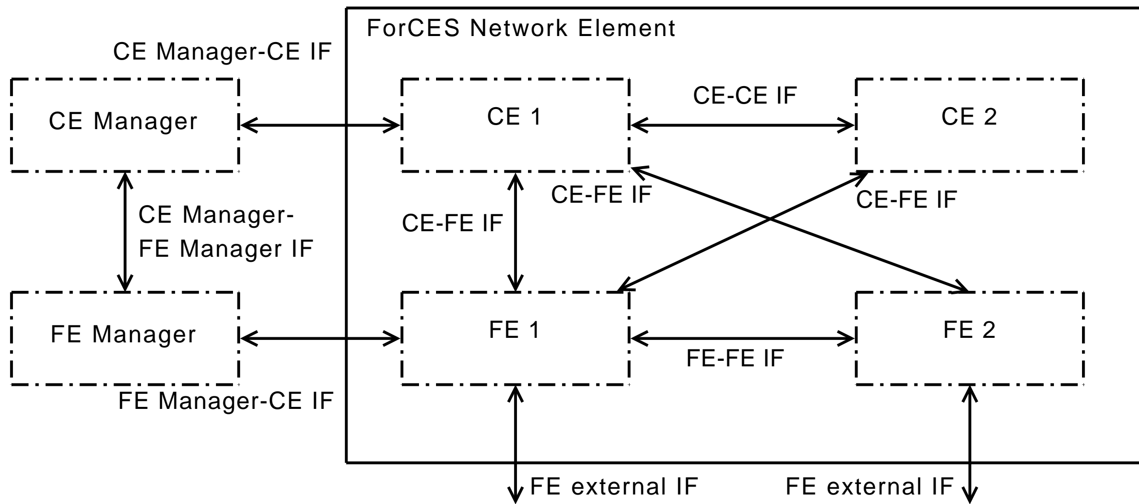

Figure 2.3: ForCES architecture [16, p. 12].

A ForCES network element can have one or more CEs and one or more FEs. An architectural diagram of a ForCES NE with two CEs and two FEs is shown in Figure 2.3. Outside the ForCES network element, there are CE Manager and FE Manager. The CE Manager duties control the CEs of the ForCES NE and respectively the FE Manager controls the FEs in the ForCES NE. The network traffic is transported from the FE 1 external interface to the FE 2 external interface and vice versa.

The protocol layer is common to all implementations of ForCES. The PL associates an FE or CE to an NE and also tears down the associations when they are no longer needed. The CE configures both the FE and associated LFBs' operational parameters using the PL. The CE can send various requests to the FE to activate or deactivate it, reconfigure its availability parameters, subscribe to specific events, among other things.

2.2.2 General Switch Management Protocol

The general switch management protocol (GSMP) is a general purpose protocol that is used to control a label switch. Label switch uses labeled datagrams with existing IP routing protocols, such as the multiprotocol label switching (MPLS) protocol.

A controller can establish and release connections across the switch, add or delete leaves on a multicast connection, manage switch ports, request configuration information, request and delete reservation of switch resources, and request statistics. The GSMP can be used to inform the controller of events, for example, when a link goes down. The protocol is asymmetric, the controller being the master and the switch being the slave. A single controller can control multiple switches using multiple instantiations of the protocol over separate control connections [21].

A connection across a switch is formed by connecting an incoming labeled channel to one or more outgoing labelled channels. GSMP supports point-to-point and point-to-multipoint connections. A multipoint-to-point connection is specified by establishing multiple point-to-point connections, each of them specifying the same output branch. A multipoint-to-multipoint connection is specified by establishing multiple point-to-multipoint trees each of them specifying the same output branches [21].

A connection is established with a certain quality of service (QoS). The default QoS configuration in GSMP version 3 includes three QoS models: a service model, a simple abstract model with strict priorities and a QoS profile model. The differences between these will not, however, be discussed in this document [21].

The GSMP is very adaptable: packets can be encapsulated in ATM, Ethernet and TCP transport, for example. The controller issues request messages to the switch which send a response message, if it is required by the message sent by the controller. The response message contains a value indicating either a successful result or a failure. There are six classes of GSMP request messages that require a response: Connection Management, Reservation Management, Port Management, State and Statistics, Configuration, and Quality of Service. In addition to this, it is allowed for the switch to generate asynchronous event messages. There is also a method to send messages for synchronization across a link and for maintaining a handshake.

Request messages and successful response messages have their own format. Failure response messages have the same format with the request message that caused the failure. The code field then tells the reason for the failure.

GSMP version 1.1 was released in August 1996 and version 2.0 in March 1998. Version 3 is the newest so far and was released in June 2002. The versions 1.1 and 2.0 were designed for ATM switches only but version 3.0 extended the applicability to other types of network switches also [21] [22, p. 1].

2.2.3 OpenFlow Protocol

OpenFlow is an open protocol which originally has been developed at Stanford University[1]. Nowadays, the OpenFlow Switch Consortium[2] is continuing the protocol's development and support work. The consortium's goal is to get the Ethernet switches and routers that are in use in the universities to support the OpenFlow protocol. OpenFlow allows researchers to test new functionalities in their own networks: new routing protocols, management techniques, or packet processing algorithms.

An OpenFlow switch has three main parts: A flow table which contains the actions to be done for each flow entry, a secure connection between the switch and the controller, and the OpenFlow protocol itself. The flow table is located in the

[1] http://www.stanford.edu/
[2] http://www.openflowswitch.org/

OpenFlow switch and the controller can access it using the proper queries using the OpenFlow protocol. An entry in the flow table contains three fields: The packet header that defines the flow, the action which defines how the packets should be processed, and statistics data to help controlling flows [23, p. 3] [24, p. 1].

At the starting point of the ETNA project the OpenFlow protocol did not yet exist, so it has not been along in choosing the protocol for ETNA needs. Today, the latest version available is version 1.0.

The OpenFlow development has also an approach to switch virtualization and there has been developed a network virtualization layer called FlowVisor [25]. The idea of network virtualization is that the same hardware can be used to serve multiple logical networks, each with a distinct forwarding logic.

2.3 Comparison of the Protocols

The GSMP protocol is the oldest of these protocols and has been available since version 1.1 designed for ATM switches in 1996. Today it has reached the version 3. The ForCES protocol is a quite new protocol, with its first specification being released in 2004, and its version number is now 1. OpenFlow is the newest of these and already this year version 1.0 is available.

The message type and length fields are the same length. The GSMP protocol has the transaction identifier which is 24 bits in length. ForCES has the correlator field which acts in the same way but is 64 bits in length. These fields are used to mark different packets belonging to a specific transaction. That means, if the packets belong to the same transaction, they will have the same identifier value. Because ForCES has a longer field for this purpose, it has also a bigger address space. In addition to these ForCES has source and destination ID fields to identify requesting and responding elements. All elements have a unique ID number which is used to identify them.

The result field in GSMP acts like an Ack-flag in ForCES wasting six bits more of header space. This field is used to tell the receiver if it must respond to the message. Pri-flag does exist only in ForCES and is used for prioritizing packets. Higher priority packets are processed first if possible. This could be of use. SubMessage number is correlated in TP-flag in some way, but they are specified slightly differently. The TP-flag is the transaction phase in the ForCES protocol and is used to identify if the packet starts or ends the transaction, ie. its the first or last packet or if it is in the middle of the transaction. They are used when the message is segmented into multiple packets.

The I-flag in GSMP toggles the sub-message number field usage whether it indicates the total number of sub-message segments that compose the entire message. The other usage is for the I-flag to indicate the sequence number of the current sub-message segment within the whole message. The AT-flag in ForCES tells if the message belongs to an atomic transaction and must be set if the transaction

operation has multiple messages.

The ForCES protocol is often implemented inside a closed system, meaning that the source code and the implementation details are not available to the public. That is usually so, because ForCES is used in communication between the CE and FE in a network element (NE) and the internal implementation is not needed or wanted to be open to everyone. In contrast to this, OpenFlow is designed in Stanford University and is meant to be an open framework.

In this Master's thesis, ForCES is the protocol chosen to be utilized in ETNA to implement CE-FE communication. This protocol was chosen because some of the project members had previous experience of it and it seemed to be the most flexible protocol for a new Ethernet environment which is not dependent on MPLS or IP.

2.4 Summary and Conclusions

There are two models of network switch architectures: the classical with the control and forwarding logic integrated and the other model in which the control and forwarding logic are separated. The separated architecture has some benefits such as the architecture can be made modular so it is easier, for example, to add control elements and forwarding elements in case of performance issues or some element gets broken. That also means the availability of the network switch is improved.

Three dynamic network switch control protocols were discussed and compared with each other. These protocols are the general switch management protocol (GSMP), the forwarding and control element separation protocol (ForCES), and the OpenFlow protocol. The ForCES protocol was chosen by ETNA and will be implemented and evaluated further in this thesis. This protocol was chosen because some of the project members had previous experience of that and it seemed to be the most flexible protocol for a new Ethernet environment which is not dependent on MPLS or IP.

It seems that the ForCES protocol chosen to be implemented in this project has all the functionality needed. The original IETF protocol specifications can be extended to meet the project needs.

Chapter 3

ForCES Protocol Details

In this chapter the Forwarding and Control Element Separation (ForCES) protocol will be described more deeply. The first section describes the message flow in different protocol states. The second section explains the ForCES message encapsulation and structure for each type of the ForCES messages. In the third section there is described the structure of all the ForCES messages and for what purpose they are used. The fourth and final section contains the summary and conclusions of the chapter.

3.1 Message Flows

ForCES messages are used, for example, to establish an association, to tear down an association, send configuration queries and changes, and to send heartbeat signals. In this section there are shown two different kinds of message flows: one for association setup state and the other for association established or steady state.

3.1.1 Association Setup State

An example of an association setup state message flow is shown in Figure 3.1. The association setup state starts after the FE has booted up. First the FE sends an association setup request to the CE. The CE responds to that with an association setup response. Now the CE knows that the FE is available and sends a query for LFB capabilities which the FE responds with a query response. After that the CE queries topology information from the FE and the FE responds to that also. The FE can also send event notifications to inform the CE like in this example the *OperEnable* event. The last message from the CE is a configuration change request also in which the FE sends a response.

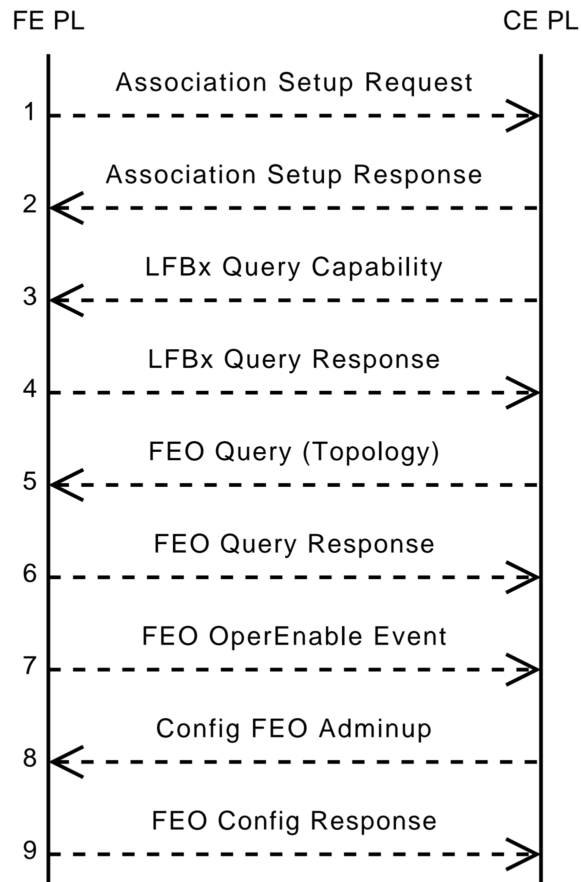

Figure 3.1: An example of message exchange between CE and FE to establish an NE association [16, p. 29].

3.1.2 Association Established State

The association established state is the state after the association between the CE and FE was successful. In this state the heartbeats are sent in distinct time intervals and there can be sent configuration changes and queries as well as event reports and packet redirects.

An example of the association established state is shown in Figure 3.2. The heartbeat signals are sent by both the CE and FE. The CE sends configuration changes and queries to the LFBs in the FE. The FE informs the CE with an event report and sends a packet redirect message also.

3.2 Message Encapsulation and Structure

All Protocol Layer (PL) Protocol Data Units (PDUs) start with a common header followed by Type-Length-Value (TLV) data structures. The common header and

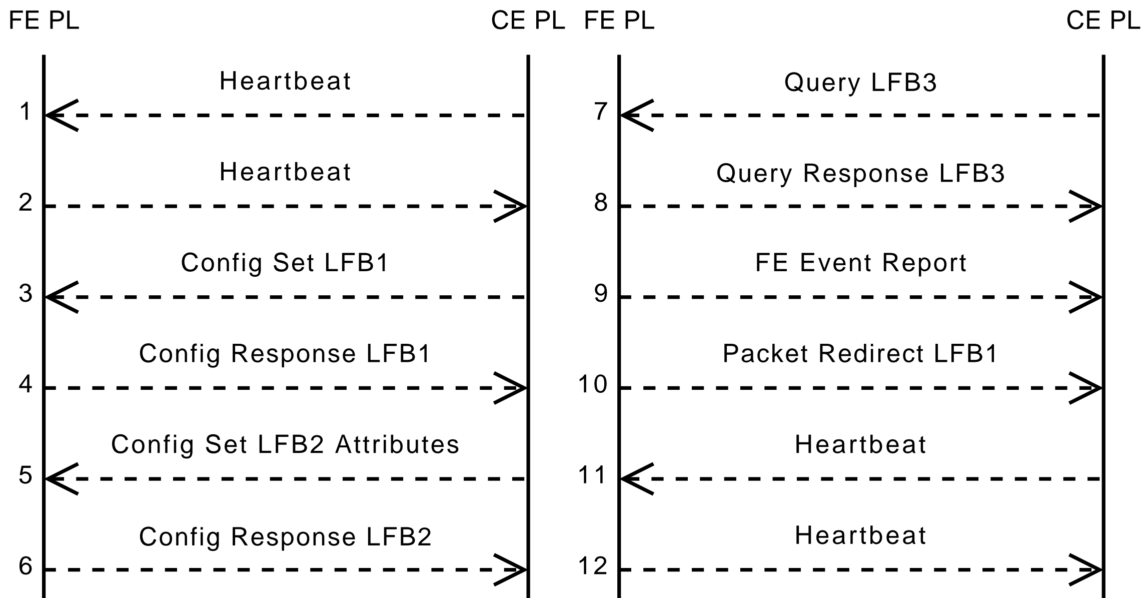

Figure 3.2: An example of message exchange between CE and FE during steady state [16, p. 30].

the TLV data structures are explained in the following sub-sections.

3.2.1 Common Header

The format of the ForCES message common header is presented in Figure 3.3. The x-axis in the figure shows the bits in a 32-bit word starting from the left and the y-axis shows the number of the bytes from the beginning of the header. All messages have this common header. Version field (4 bits) tells the version number of the ForCES protocol used in the current session. The next four bits are reserved for future use. The sender must set them to zero and the receiver should not try to interpret them. After that follows the message type field (8 bits) that has the value of the ForCES message type. The length field (16 bits) contains the value of the length of the message in 32-bit words.

The Source ID consist of a 2-bit source type selector (sTS) and a 30-bit sub-ID (together 32 bits). The destination ID field correspondingly has a 2-bit destination type selector (dTS) and a 30-bit sub-ID (together 32 bits). The correlator field (64 bits) is the message correlator that is used to match the request and response messages with each other. The CE generates the correlator value for each of the ForCES requests. The FE must assign the same correlator value for the response it sends back to the CE. If the correlator is not relevant, for example, when a response is not expected, the field is set to zero [26].

After the correlator field there is the 32-bit flags field containing flags ACK (Acknowledge, 2 bits), Pri (Priority, 3 bits), EM (Execution Mode, 2 bits), AT

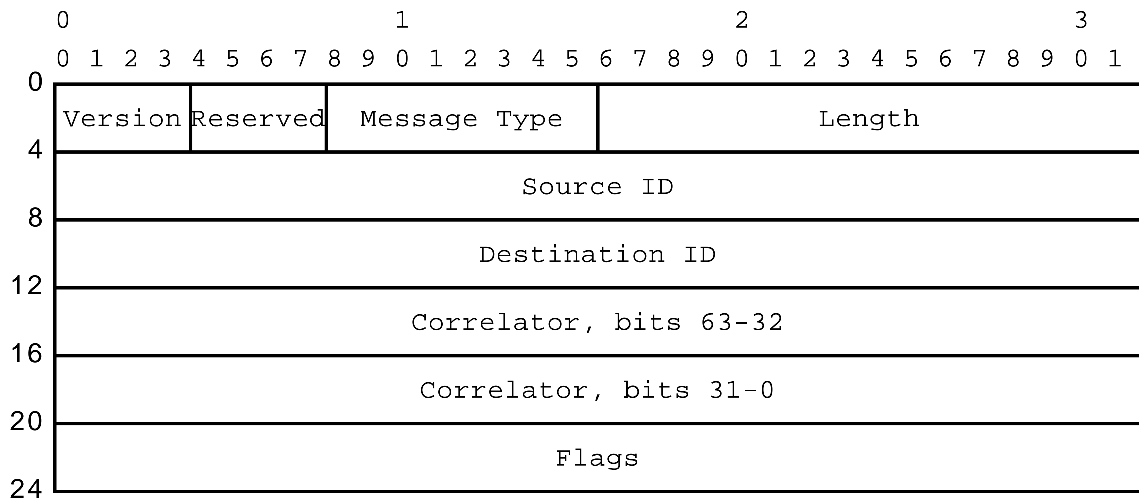

Figure 3.3: Common header layout [16, p. 36].

(Atomic Transaction, 1 bit) and TP (Transaction Phase, 2 bits).

When sending a Config message, or Heartbeat message, the CE uses the ACK-flag to tell the message receiver whether or not a response is required by the sender. For all other messages the ACK-flag is not used. The FE does not set the ACK-flag, it only reads the status of this flag when needed. The flag values [16, p. 39] are the following: *NoACK*, *SuccessACK*, *FailureACK*, and *AlwaysACK*. The *NoACK* value means that the message receiver must not send any response message back to the message sender while the *SuccessACK* means that the message receiver must send a response message back only when the message has been successfully processed by the receiver. The *FailureACK* tells the message receiver that it must send a response message back to the sender only when the message processing has failed, and the *AlwaysACK* tells the message receiver that it must send a response message to all the received messages in any case.

An Association Setup message and Query message always expects a response. An Association Teardown message, Packet Redirect message, and all response messages never expect a response. The ACK-flag is ignored with these messages.

The Pri-flag contains the priority of the message. The ForCES protocol defines eight priority values from 0 to 7, seven being the most important priority value. The normal priority value is 1.

The EM-flag can contain one of three execution modes: *execute-all-or-none*, *execute-until-failure*, and *continue-execute-on-failure*. The *execute-all-or-none* mode executes all operations serially and the FE must not have any execution failure for any of the operations. If a failure happens anyway, all operations done must be undone. The *execute-until-failure* mode executes all operations on the FE serially. If a failure happens, the rest of the operations are not executed, but operations

already completed are not undone. The *continue-execute-on-failure* mode continues execution to the end, even when a failure happens.

The AT-flag is used to indicate if the message is a stand-alone message or belongs to an atomic transaction operation with multiple messages. The TP-flag indicates the transaction phase this message belongs to. There are four possible phases for an atomic transactional operation: SOT (start of transaction), MOT (middle of transaction), EOT (end of transaction), and ABT (abort). The first message in an atomic transaction operation has the SOT TP-flag value and the second and further messages have the MOT value in the TP-flag. If there is no failure the CE sends the last message with the EOT transaction value. Any failure notified by a FE causes the CE to send a Config message with the TP-flag set to the ABT value to abort the transaction on all FEs involved.

3.2.2 Type-Length-Value Data Structure

Type-Length-Value (TLV) data structure is a basic building block in all of the ForCES messages. After the common header the payload data is encapsulated to these data structures. Figure 3.4 shows the layout of the TLV data structure.

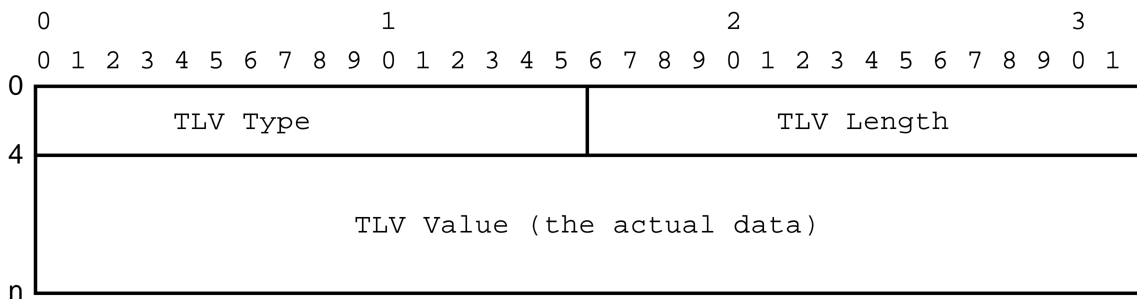

Figure 3.4: Type-Length-Value (TLV) data structure layout [16, p. 41].

The x-axis in the figure shows the bits in a 32-bit word starting from the left and the y-axis shows the number of the bytes from the beginning of the TLV. There is first a 16-bit TLV type field which has the TLV type ID. After that there is the TLV length field that tells the length of the TLV in bytes including both the type and the length fields. After that comes the TLV value, ie. payload.

TLVs can be either only one or many inside a payload and they can be nested inside the value field of the other TLV. An example of this is shown in Figure 3.5. All fields are in network byte order. Depending on the message type, there can be one or several TLVs after the common header.

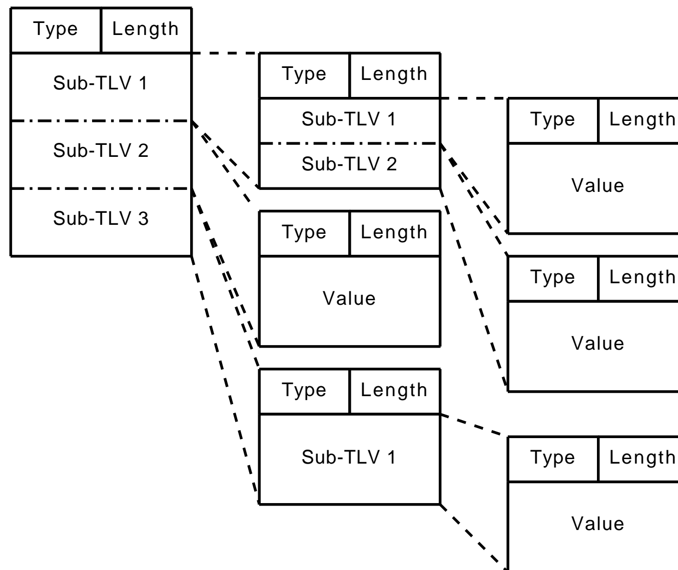

Figure 3.5: An example of TLV encapsulation with a set of sub-TLVs.

LFBselect TLV

LFBselect TLV is used in messages that need a specific operation on a selected logical function block (LFB). These message types are Association Setup messages, Config messages, Config Response messages, Query messages, Query Response messages and Event Notification messages. Association Setup Response messages have ASRresult TLV, and Association Teardown messages have ASTreason TLV as their top-level TLV. Packet Redirect messages have Redirect TLV as a top-level TLV.

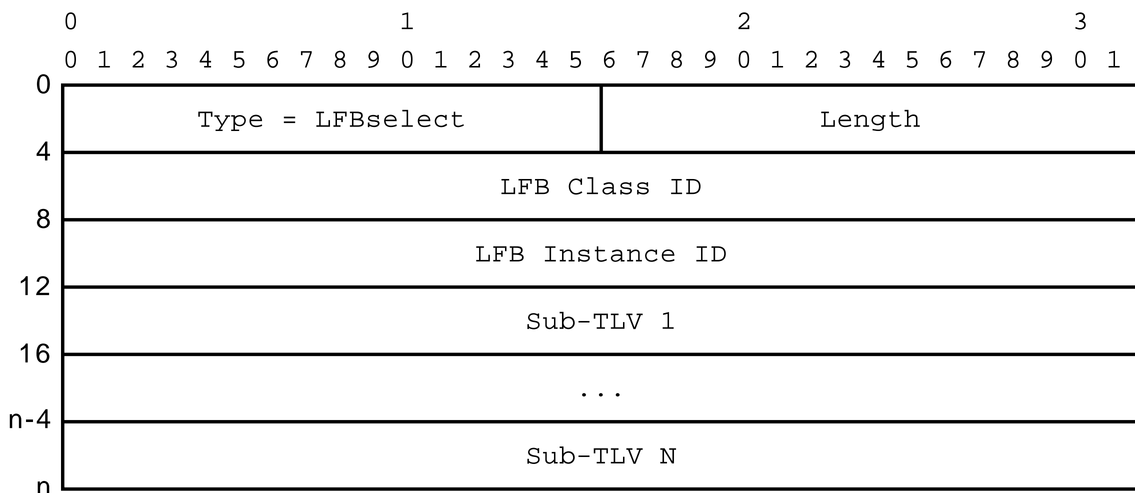

Figure 3.6: LFBselect TLV layout [16, p. 51].

Heartbeat messages have no top-level TLV at all.

The layout of the LFBselect TLV is shown in Figure 3.6. The type of the TLV is *LFBselect*, the LFB Class ID and LFB Instance ID fields define the target LFB. There can be one or more sub-TLVs at the end of the PDU.

The message body consist of type-length-value (TLV) structures that can be one or several sequentially or embedded to the other TLV's value field. This type of structuring is very flexible: data can be added to the end of the message and after the addition only the length of the message must be updated.

3.2.3 Identifier-Length-Value Data Structure

In the ForCES protocol there are two types of TLV kind of data structures used: the TLV data structures with 16-bit type and length fields and the identifier-type-length (ILV) data structures which use 32-bit type and length fields (Figure 3.7).

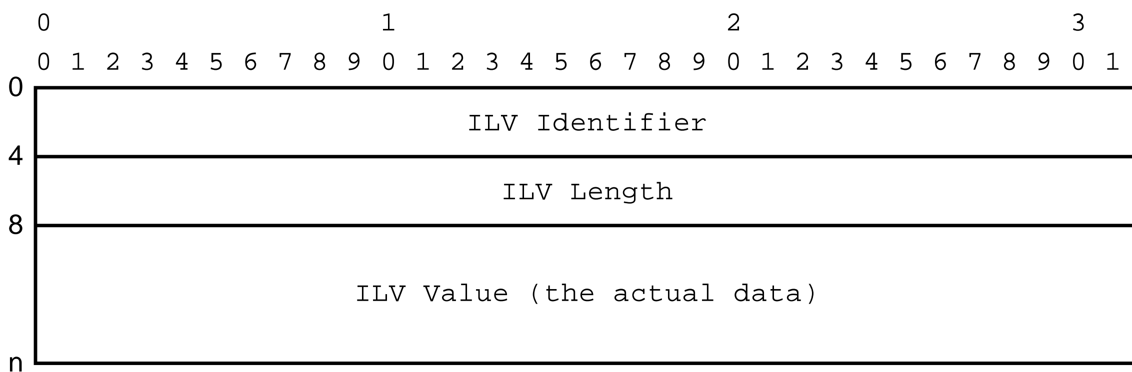

Figure 3.7: Identifier-Length-Value (ILV) data structure layout [16, p. 43].

The operation of the ILV is basically the same as with the TLV structures. The nature of type (or identifier) field is that the type value should be unique in the same context. The length of the structure is expressed in bytes, including type and length fields.

Both the TLV and ILV structures have to be padded to the 32-bit boundary, so that if the length of the structure is not dividable by four, there must be added zeros to the end until the length criteria is met.

3.3 ForCES Messages

The IETF ForCES protocol specification defines six different kinds of messages. These are association messages, configuration messages, query messages, event notification messages, packet redirect messages, and heartbeat (HB) messages. Associ-

Table 3.1: ForCES message types.

Name	Value	Top-level TLV	OPER-TLV(s)
Association Setup	0x01	LFBselect	REPORT
Association Setup Response	0x11	ASRresult-TLV	none
Association Teardown	0x02	ASTreason-TLV	none
Config	0x03	(LFBselect)+	(SET | SET-PROP | DEL | COMMIT | TRCOMP)+
Config Response	0x13	(LFBselect)+	(SET-RESPONSE | SET-PROP-RESPONSE | DEL-RESPONSE | COMMIT-RESPONSE)+
Query	0x04	(LFBselect)+	(GET | GET-PROP)+
Query Response	0x14	(LFBselect)+	(GET-RESPONSE | GET-PROP-RESPONSE)+
Event Notification	0x05	LFBselect	REPORT
Packet Redirect	0x06	REDIRECT-TLV	none
Heartbeat	0x0F	none	none

ation, configuration, and query messages have also response messages that are sent as a response to a request message. There is also an association teardown message to tear down an association. Message types are listed in Table 3.1. The ForCES specification allows adding multiple TLVs in a sequence. The '+' sign indicates this after the TLV name [26].

3.3.1 Association Messages

The ForCES association messages are used to establish and tear down associations between FEs and CEs [16, p. 67]. FEs always start establishing a ForCES association by sending an association setup message to a chosen target CE. The CE then responds with an association response message containing a result, whether the association was successful or not. When an association is no longer needed, it can be torn down by either of the FE or the CE.

Association Setup Message

The association setup message is sent by the FE to the CE to setup a ForCES association between them [16, p. 67].

The message type is *AssociationSetup*. The ACK-flag in the common header is ignored, since the association setup message always expects to get a response from the message receiver (CE), whether the setup was successful or not. The correlator field in the common header is set, so that the FE can correlate the response coming back from the CE correctly [16, p. 67].

The association setup message body can contain zero, one or two LFBselect TLVs. The LFB class ID in the LFBselect TLV can point either to the FE Object LFB or to the FE Protocol LFB. If neither of the LFBs need to be specified, LFBselect TLVs are omitted. If both LFBs are to be specified, then there will be two LFBselect TLVs in an association setup message. The OPER-TLV is optional. The type of the OPER-TLV is *REPORT*. The value field contains a PATH-DATA-TLV for REPORT which can contain FULLDATA-TLV(s), but not any RESULT-TLV [16, p. 67].

Association Setup Response Message

The association setup response message is sent by the CE to the FE in response to the association setup message. It indicates to the FE whether the setup was successful or not [16, p. 69].

The message type is *AssociationSetupResponse*. The ACK-flag in the common header is ignored. The association setup response message never expects a response from the message receiver (FE). The destination ID in the header will be set to the source ID in the corresponding association setup message, if the value is not zero [16, p. 69]. The top-level TLV type is *ASRresult* and it contains a 32-bit association setup result in the value field [16, p. 70].

Association Teardown Message

The Association Teardown message can be sent by the FE or the CE to any ForCES element to end its ForCES association with that element [16, p. 70].

The message type is *AssociationTeardown*. The ACK-flag is ignored. The correlator field is not used in this message and is set to zero. The top-level TLV type is *ASTreason* and it contains a 32-bit teardown reason in the value field [16, p. 71].

3.3.2 Configuration Messages

The ForCES configuration messages are used by the CE to configure the FEs in a ForCES NE and report the results back to the CE. A configuration change starts when the CE sends a Config message to the FE. The FE then responds with a Config Response message indicating the result of the configuration change. The receiver (CE) no longer replies to this message whatever the result is [16, p. 72].

Config Message

The Config message is sent by a CE to a FE in a ForCES NE to configure LFB components in the FE. This message is also used by the CE to subscribe or unsubscribe to LFB events [16, p. 72].

The message type in the common header is *Config*. The ACK-flag can be set to any value. The OPER-TLV type field has the operation type for the config message. The value field has the PATH-DATA-TLV for the Config message. The operation type of the OPER-TLV for the config message has five values: *SET*, *SET-PROP*, *DEL*, *COMMIT* and *TRCOMP* [16, p. 72].

The *SET* operation type is used to set the LFB components in the FE. The *SET-PROP* type is used to set the properties for the LFB component. The *DEL* operation type deletes some LFB components. The *COMMIT* operation type is sent to the FE to commit in a two phase commit (2PC) [27] transaction. 2PC is a classical transactional protocol that is used to achieve the transactional operations utilizing configuration messages. A *COMMIT* TLV is an empty TLV with no value, i.e. its length is four bytes containing the header only. The *TRCOMP* operation is sent to the FE to mark the success from a NE perspective of a 2PC transaction. A *TRCOMP* TLV is also an empty TLV with no value, i.e. its length is four bytes containing the header only [16, p. 72].

The PATH-DATA-TLV for *SET* or *SET-PROP* operations must contain either a FULLDATA-TLV or SPARSEDATA-TLV(s), but must not contain any RESULT-TLV. The *DEL* operation may contain a FULLDATA-TLV or SPARSEDATA-TLV(s), but must not contain any RESULT-TLV [16, p. 73].

Config Response Message

The Config Response message is sent by the FE to the CE in response to the Config message. The response indicates whether the configuration change was successful or not on the FE and also gives a detailed response regarding the configuration result of each component [16, p. 74].

The message type is *Config Response*. The ACK-flag must be always ignored, and the message receiver (CE) must not send any further response to this message. The OPER-TLV has a PATH-DATA-TLV in its value field. The type of the OPER-TLV is one of the following: *SET-RESPONSE*, *SET-PROP-RESPONSE*, *DEL-*

RESPONSE, or *COMMIT-RESPONSE* [16, p. 74–75].

The *SET-RESPONSE* operation is for the response of the *SET* operation of LFB components. The *SET-PROP-RESPONSE* operation is for the response of the *SET-PROP* operation of LFB component properties. The *DEL-RESPONSE* operation is for the response of the *DEL* operation of LFB components. The *COMMIT-RESPONSE* operation is sent to the CE to confirm a commit success in a 2PC transaction. A *COMMIT-RESPONSE* type of TLV must contain a RESULT-TLV indicating success or failure [16, p. 75].

The PATH-DATA-TLV for *SET-RESPONSE* operation must contain one or more RESULT-TLVs so that each of the operations get a result for the *SET* operation. The same restriction applies to the *DEL-RESPONSE* operation [16, p. 75].

3.3.3 Query Messages

The ForCES query messages are used by the CE to query LFBs in the FE for the LFB component, capability, statistics, and that kind of information. A query message is sent by the CE and the FE responds back with a query response message containing the query result [16, p. 76].

Query Message

The ForCES Query messages are sent by the CE to the FE to query LFBs for information such as LFB components, capabilities, and statistics data. The common header is the same as in other ForCES messages and the message body has one or more TLVs. The message type is *Query*. The ACK-flag in the common header is always ignored. A response for all queries in the message is always expected. The correlator field in the common header is used by the CE to locate the response back from the FE correctly [16, p. 77].

The OPER-TLV has a PATH-DATA-TLV in its value field. The type of the OPER-TLV is either *GET* or *GET-PROP*. Neither of the operations must contain any SPARSEDATA-TLV, FULLDATA-TLV, or RESULT-TLV [16, p. 77].

Query Response Message

When the FE receives a ForCES Query message, it processes the message and makes a query result. Then it sends the query result back to the message sender (CE) by use of the Query Response message. If the query was successful, the reply message contains the information being queried otherwise it contains an error code if the query operation fails, indicating the reason for the failure. One Query Response message can contain several answers to queries, as many as there were in the original Query message for which the response is to [16, p. 78].

A Query Response message starts with a common header. After that comes the message body consisting of one or more TLVs describing the query result. The message type in the common header is *QueryResponse* and the ACK-flag is ignored. The Query Response message does not expect a further response [16, p. 78].

The operation type is one of the two: *GET-RESPONSE* or *GET-PROP-RESPONSE*. They are responses for the *GET* and *GET-PROP* operations respectively. These responses can contain SPARSEDATA-TLV, FULLDATA-TLV and/or RESULT-TLV(s) in the PATH-DATA-TLV data encoding [16, p. 79].

3.3.4 Event Notification Message

ForCES Event Notification messages are used by the FE to asynchronously notify the CE of events in the FE. Different events can be generated in the FE, and the CE can subscribe to all of those events by sending a Config message with a *SET-PROP* operation, where the included path specifies the event [16, p. 80].

An Event Notification message starts with a common header. After that comes the message body consisting of one or more TLVs describing the event notifications. The message type in the common header is *EventNotification* and the ACK-flag is ignored. The Event Notification message does not expect a further response [16, p. 80].

Only *REPORT* operation type is defined for the event notification message. The PATH-DATA-TLV can contain SPARSEDATA-TLV or FULLDATA-TLV, but no RESULT-TLV(s) in the PATH-DATA-TLV data encoding [16, p. 81].

3.3.5 Packet Redirect Message

A Packet Redirect message is used to transfer data packets between the CE and FE. These data packets are normally control packets, but they may be also data-path packets that need further processing. It is possible also to use this message for transferring only metadata [16, p. 82].

A ForCES Packet Redirect message starts with a common header. After that comes the message body consisting of one or more TLVs containing or describing the packet being redirected. The message type in the common header is *PacketRedirect*. The TLV is a Redirect TLV with the type *Redirect*. Messages can be sent from the CE to FE or from the FE to CE [16, p. 82].

The Redirect TLV has two sub-TLVs: Meta Data TLV and Redirect Data TLV. The Meta Data TLV specifies meta-data associated with the followed redirected data. The type of the TLV is *METADATA-TLV*. The value field contains a number of Meta Data ILVs. The Identifier-Length-Value (ILV) format is similar to TLV format, except the 16-bit type field and the 16-bit length fields are replaced by the 32-bit identifier and the 32-bit length fields. The Meta Data ILV contains the actual meta data for the packet redirect message [16, p. 83].

The Redirect Data TLV contains the packet that is to be redirected in the network byte order. The packet should be 32-bits aligned as all the data for the TLVs. That is, if the data is not dividible by four bytes, the data is padded with zeros until the criterion is met [16, p. 83].

3.3.6 Heartbeat Message

A ForCES element, either the CE or FE, can use the Heartbeat (HB) message to asynchronously notify one or more other ForCES elements in the same ForCES NE of its existence. The heartbeat message is sent periodically. It has no message body.

The message type in the common header is *Heartbeat*. The ACK-flag in the common header is set to either *NoACK* or *AlwaysACK* when the HB is sent. The *NoACK* value means that response from the receiver is not expected. When the ACK-flag is set to *AlwaysACK* value, the response is always expected. When the response is expected, the correlator field should be set to the proper value so that the receiver can correlate the response correctly.

3.4 Summary and Conclusions

Different types of ForCES messages are used in different protocol states. When the forwarding element (FE) starts up, it establishes an association with the CE. During the steady state both the CE and FE send heartbeat signals to the other element to show that they are alive. After the association has been made, the CE can send configuration queries and changes to the FE. The FE informs CE of event changes with an event notification.

A ForCES message consists of a common header and a number of TLV/ILV data structures. The message payload consist of TLVs which can include one or many sub-TLVs inside them. The message type indicates how many TLVs a ForCES message can have.

The IETF ForCES protocol specification defines six different kinds of messages. These are association messages, configuration messages, query messages, event notification messages, packet redirect messages, and heartbeat (HB) messages.

The message encapsulation and structure are specified quite well in the IETF ForCES specification and the message flows seem to be quite reasonable, so we can proceed to the next chapter how ETNA is going to modify the original ForCES specifications to meet its needs in the project.

Chapter 4

ETNA Requirements

In this chapter the ETNA requirements for the ForCES protocol are discussed. First a general picture of a network element used in ETNA network model is given to the reader. The second section describes the requirements for the ForCES messages and the modifications to the original IETF specification.

4.1 Network Element Architecture Overview

The ETNA project's view of the architecture of a network element (NE) is described in Figure 4.1. There are three logical planes: the management plane (MP), the control plane (CP) and the data (forwarding) plane (DP).

The management plane is where the network operator manages network elements using a user interface (UI) module. The control plane has control elements (CE) and a management system (MS). There can be one or more control elements in parallel, depending on the configuration. The IETF ForCES specification allows more than one CE-FE connection for each of the CEs, but in the ETNA model every control element has only one forwarding element it is associated with. The management system is accessed by the user interface module and is further on connected to the control elements. The data plane contains forwarding elements (FE) which handle the transport of the actual network traffic [28].

The communication between the management plane and control plane goes through the MP-CP interface which is realized with SOAP (former simple object access protocol) [29, p. 11]. This interface has been implemented and is described further in [30].

The CE to MS communication utilizes an extended resource reservation protocol (mRSVP) which is based on the resource reservation protocol (RSVP). mRSVP utilizes similar message structures that are used in RSVP. It has been designed to handle communication between several distributed control elements and one management system. Communication between multiple distributed control elements within the control plane is managed by the RSVP-TE with Ethernet extensions

(RSVP-TEEth) which is based on the RSVP-TE [31]. The mRSVP and RSVP-TEEth are implemented and described further in [32].

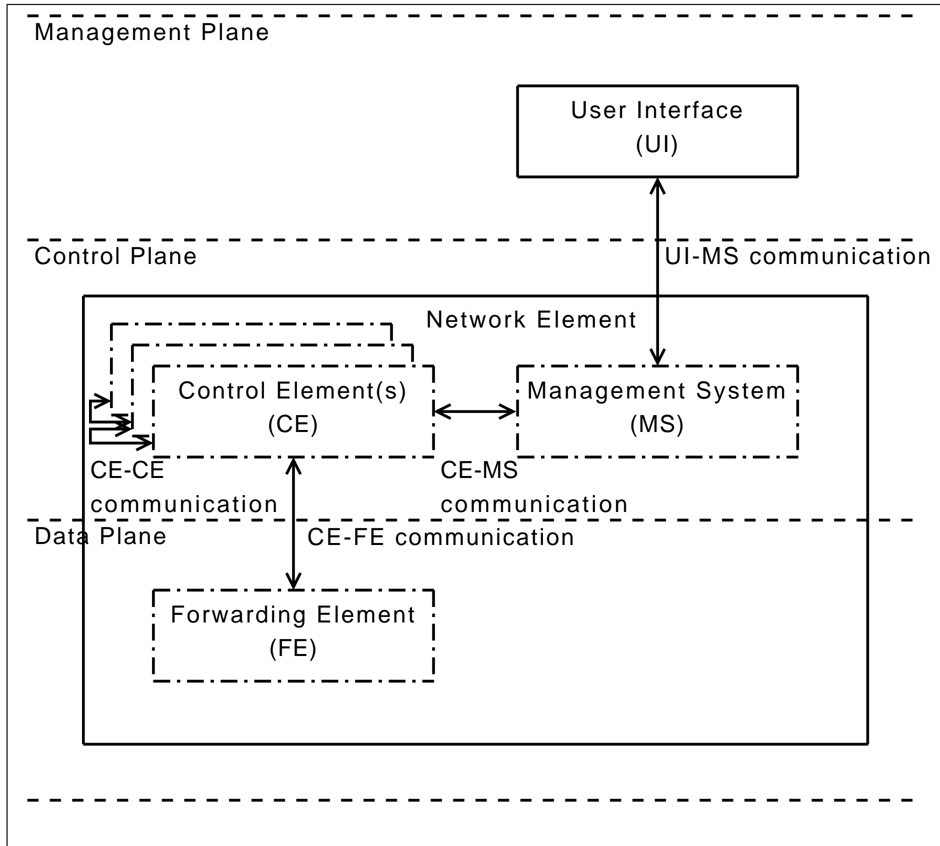

Figure 4.1: Network element structure with layer separation.

A control element and a forwarding element communicate with each other through the control plane and data plane application protocol interface (API). The protocol used for that communication is the forwarding and control element separation (ForCES) protocol, of which the author has created an implementation for this project. The implementation of the protocol is evaluated in this thesis.

4.2 Requirements for the ForCES Messages

The format of the common header remains the same in the ETNA specification as in the original IETF ForCES specification. The version number used in ETNA is 1. Valid message type values are the same as in the IETF specification and can be found in Table 4.1. Source and destination IDs consist of a type selector (TS) and a sub-ID. The usage of the type selector is the same as in the IETF specification but the sub-ID used in ETNA is 1 because there is only one forwarding element associated with a corresponding control element. The correlator and the flags functionalities are not changed [26].

Table 4.1: ForCES message types used in ETNA.

Name	Value	Top-level TLV	Sub-TLV type
Association Setup	0x01	LFBselect	Report
Association Setup Response	0x11	ASRresult	none
Association Teardown	0x02	ASTreason	none
Config	0x03	LFBselect+	Config+
Config Response	0x13	LFBselect+	Config Response+
Query	0x04	LFBselect+	Query+
Query Response	0x14	LFBselect+	Query Response+
Event Notification	0x05	LFBselect	Report
Packet Redirect	0x06	Redirect	Meta Data+, Redirect Data+
Heartbeat	0x0F	none	none

ETNA uses the following LFB class IDs: manager, forwarding database, OAM, QoS, tunnel control, topology discovery, and status monitor. The corresponding ID values for these can be found in Appendix C. In the ETNA implementation the LFB instance ID is set to 1 as there is only one active instance of each LFB at a time.

ETNA has specified eleven FE capability type values used in FE Capabilities TLV, fifteen Config data types used in Config Data TLV, three extra result codes used in Result TLV, 27 Config parameter values used in Parameter Field TLV, and 12 event codes used in Event Notification TLV. The values are chosen so that they do not overlap with the values in the IETF specification. All these values are listed in Appendix C.

Modifications to the IETF specification

The PATH-DATA-TLV layer was removed from the ForCES model for simplicity as it was of no relevant use. Also the FULLDATA-TLV and SPARSEDATA-TLV was left unimplemented. All other TLVs were implemented.

4.3 Summary and Conclusions

The ETNA network element architecture view has three logical planes: the management plane, the control plane and the data (forwarding) plane. The control plane can have many control elements but only one forwarding element associated with each of the control elements. In the control plane there is also a management system which is controlled by the user interface in the management plane. The ForCES protocol is used between the CE-FE communication and the CE-MS communication utilizes mRSVP. RSVP-TEEth is used for the CE-CE communication. The MP-CP interface is utilizing SOAP.

The format of the common header is the same in the ETNA specification as in the IETF ForCES specification. ETNA uses the following LFB class IDs: manager,

forwarding database, OAM, QoS, tunnel control, topology discovery, and status monitor. In addition to these, ETNA has specified a bunch of IDs, data types, event and result codes, and other values that do not overlap with the values in the IETF specification. The PATH-DATA-TLV layer, FULLDATA-TLV and SPARSEDATA-TLV were removed from the ForCES model.

The ETNA ForCES model has some modifications differing from the IETF specification. It has been a bit simplified but not much. A few dozens of new values have been defined in ETNA ForCES specification for the needs of the ETNA implementation.

Chapter 5

Software Implementation of the ForCES Protocol

This chapter presents the software implementation of the ForCES protocol. The first section is about the implementation and the second section explains the code structure. The third section describes the class inheritance and the fourth is about the library use. The development environment is described in the fifth section. The sixth section tells about the coding process and the seventh section has the summary and contents.

5.1 About the Implementation

The implementation of the forwarding and control element separation (ForCES) protocol was done following the ETNA requirements specification for the ForCES Messages [26].

The programming language used in ETNA was C++, so the object-oriented programming (OOP) paradigm [33] was used from the beginning in the design of the software. This includes data abstraction, encapsulation, inheritance, and polymorphism. With data abstraction the functions are defined, but the implementation itself is hidden. By encapsulation, the data inside an object is not directly accessible. Instead of that, object member variable access functions must be used. Inheritance is used to put the same kind of objects together to achieve better code structure and reduce the amount of code by placing common code to all inherited classes to base class. Polymorphism makes possible call functions with different parameter types and the number of parameters can vary.

The main function of the implementation is to provide an interface for message creation and accessing, so it was sensible that the software compiles to a library. The most convenient way to do this was to compile it to a static library with a makefile. The ForCES message library provides classes and functions for creating and accessing ForCES messages to be transported between the CE and FE. The

complete library includes 31 classes and 15 436 lines of source code.

In addition to the ForCES Message library, a common test program was written by the author to test all the messages with different inputs. The tests create and modify message and TLV/ILV objects in several ways with varying input parameters. They also print the contents of the message to the standard output (i.e. screen) both in the hex and ASCII format. Finally, the created objects are removed to release the allocated memory. The idea of the test program is to run through all the essential parts of the library. This covers are constructors, destructors, class member variable accessing functions, byte array reading and writing functions, message validation functions, message length calculation functions, and message printing functions.

5.2 Code Structure

There are three base classes in the ETNA ForCES Message library: the ForCES Message base class, TLV base class, and ILV base class. Every object that is created runtime either belongs to or is inherited from one of these base classes. There is no main program, because the code compiles as a library. Every class has its own source and files also.

All of the ten ForCES message types are derived from the base class ForCES Message. Because the common header is the same for every ForCES message, all the common header data is placed in the base class member variables. Also functions that handle these variables are located in the *ForCES Message* base class. The difference between different types of ForCES messages is visible by checking which top-level TLV and sub-TLVs it has.

The top-level TLV and the sub-TLV classes are derived from the base class TLV. The type and the length fields are the same for all of the classes, so these fields and the accessing functions for them are placed in the TLV base class. Also the value field is stored as a byte array in the base class for uniformity. The derived classes have the value data in a more detailed way, i.e. it is stored in their class member variables which are different depending on the derived class. The ILV classes are derived the same way from the base class ILV, although there is currently only one ILV structure specified in the IETF ForCES specification which is the Meta Data ILV [16, p. 84].

In addition to the specific class member variable accessing functions, the author has implemented a couple of generic form class member functions that are included in all of the ForCES Message, the TLV and ILV classes. These are functions for byte array reading and writing, a function that checks the validity of an object and a funtion that calculates the actual length of the data stored in an object in bytes. That is, the amount of data in the message structure to be sent to the network. The code is specific for each of the classes, so it is located with the derived classes.

The ForCES message base class has also a function that prints the contents of the message to the standard output in both a hexadecimal and a clear text formats

for content checking and testing purposes.

5.3 Class Inheritance

There are three kinds of objects that can be created: ForCES Messages, TLVs, and ILVs. All ForCES Messages are inherited from the ForCES Message base class. All TLV objects are inherited from the TLV base class and all ILV objects are inherited from the ILV base class, correspondingly. The TLV class inheritance principle is shown in Figure 5.1. The same principle applies to the ForCES Message and ILV base classes.

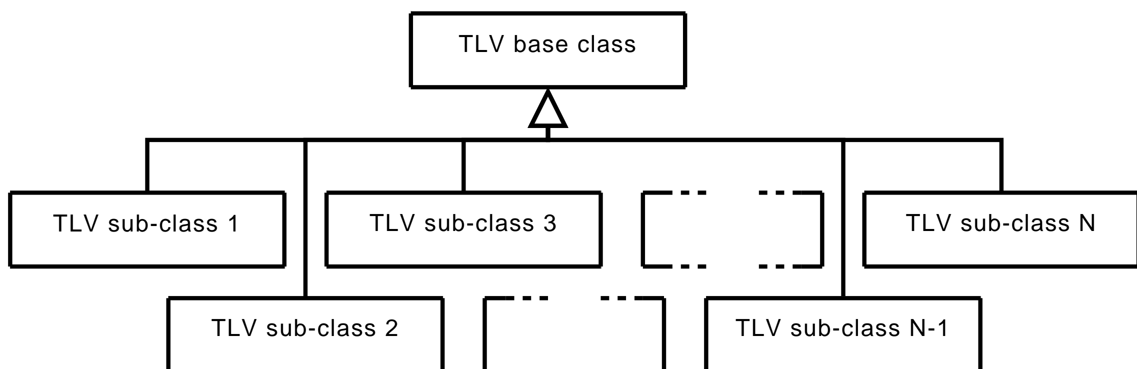

Figure 5.1: TLV class inheritance.

All implemented classes have their own constructors and a destructor. The classes have their member variables encapsulated as a private data type, so their values must be accessed using public member variable accessing functions. Depending on the type of the private data, the access functions can be setters and getters, but also functions that add or remove items in the case of a list or similar kind of data structure. These data structures are used in some message and TLV classes to store sub-TLVs in the other TLVs and the top-level TLVs in the message classes.

All classes have functions to read a raw network order data array to the object data and vice versa. There are also functions to validate an object and calculate it's length in bytes. Message classes have also a function to print the contents of the message to the standard output.

5.4 Library Use

There are two basic situations the ForCES message library is used: when a message arrives from the network and its contents must be read into a ForCES message object in order to access it and when the message contents must be written to a network order byte array containing the message that can be sent to the network.

The message can be easily modified when its data is read to a message object. After the modifications, the data can be written to a network order byte array.

5.4.1 ForCES Message Construction

There are five ways to create a message using the ETNA ForCES library:

(1) calling a specific message class constructor with no parameters and then filling the values separately,

(2) calling a specific message class constructor with all parameters separately given,

(3) calling a specific message class constructor with data stored in field format,

(4) calling a specific message class constructor with byte array as a parameter where the data is read to the class member variables, and

(5) calling a specific function that returns a base class type ForCES Message as a return value.

Method (1) is used when we want to create an empty ForCES message and afterwards set all the object member variables and fields. This method is also used when calling a default constructor for a ForCES message.

If we have all the needed values for creating a ForCES message already, we can use method (2) and call a constructor that takes all the parameters separately. The result is a complete ForCES message, if all the data is submitted and correct. After the construction, the calling program should validate the message to ensure all values were valid. Depending on the type of the message, there can be one or more TLVs after the common header. If the message has only one TLV that can be optional, the TLV is stored as a pointer in the message object and the memory must be allocated dynamically for the TLV. On the other hand, if the message can include several TLVs, they are stored in a pointer vector. In this case, the TLVs cannot be set in the message construction phase, but they must be added after creating the message using specific message class member functions.

Method (3) can be used, if we have the message input data as whole fields. For example, we can have the value of the whole 32-bit flag field, instead of each of the flags' values separately. The operation with TLVs is the same as in method (2).

When we have received a ForCES message from the network and want to construct a message object from the input data, we can use method (4) that takes a byte array as a parameter. The second parameter is the length of the array, since the parameter array is a pointer and has no size information in itself. There is, though, a message length field in the common header, but we cannot trust the value given in the input array is correct – the data can have been corrupted in the network

or it can be otherwise a wrong value. A wrong value can make the program read the data from outside of the allocated memory area for the input array causing a segmentation fault.

Method (4) can be used when we expect a certain type of a ForCES message, but when we do not know the type of the message for sure, we can use method (5) to construct the message. The parameters are the same as in method (4). The type of the return value is the base class ForCES Message, so the constructed message must be dynamically cast to the actual type of the message when operated.

Message Construction in Details

When creating a ForCES message using method (1) described in the previous subsection, the calling program must call a constructor of the desired message type, for example, in the case of Association Setup message, the constructor to be called is the default constructor *AssociationSetupMessage()* with no parameters. There are two ways to do this, depending whether the memory is to be allocated from the stack or from the heap. The memory is allocated from the stack using a default constructor call inside object declaration:

```
AssociationSetupMessage message;
```

The other way is to use the *new* command for allocating memory from the heap:

```
AssociationSetupMessage* message = new AssociationSetupMessage();
```

In this case, the allocated memory must be released, when not needed anymore, using the *delete* command:

```
delete message;
```

Now we have an empty Association Setup message object called *message*. After this, we can set the message object member variables using specific setter functions. The first field, the version field, can be set using the following message class member function call:

```
message->setVersion( 0x10 );
```

This function sets the first 8-bit common header field including the 4-bit version number with the value 0x10 in whole instead of only the version number. One must note that the version number is actually the four most significant bits of the eight bit value, so the version number bits must be shifted to the left by four before calling the function. The version number of the ForCES protocol is 1 in this example. The four least significant bits are reserved for future use and must be set to zero. When

using a constructor with all parameters separately given, the bit shifting is done in the costructor. In other cases, the whole field value must be given.

The message type field is set in the construction phase, so normally one does not have to set it anymore after construction. The message length behaves the same way: the value is updated after every size change automatically. The message should be validated using the class member function *isValid()* before sending it to network, though. The length can be calculated using the class member function *calculateLength()* and checked against the message length field to ensure the value is correct. The message length can be asked by using the class member getter function *getMessageLength()*. These functions are used this way:

```
bool valid = message->isValid();
uint16_t calculatedLength = message->calculateLength();
uint16_t storedLength = message->getMessageLength();
```

The source sub-ID and destination sub-ID fields can be set using either one 32-bit value as a parameter including both the source/destination sub-ID and sTS/dTS fields or giving them separately as two parameters:

```
message->setSourceSubID( 1, 0 );
message->setDestinationSubID( 0x40000001 );
```

The setter function for message correlator takes one 64-bit value as a parameter:

```
message->setCorrelator( 664ull );
```

The *ull* specifier just after the new correlator value stands for *unsigned long long* integer type, i.e. an unsigned 64-bit integer value, and must be included when working with larger than 32-bit unsigned integer values in a 32-bit processor architecture.

The flags field can be set also two ways: using the *setFlags()* class member function that takes one 32-bit value as a parameter to set all the flags, or setting each flag separately using the *setFlag()* class member function:

```
message->setFlags( 0b00001000010000000000000000000000 );
message->setFlag( Ack, 0b11 );
```

In here, the latter function takes the flag name and the new flag value as parameters. The flag name is an enumerated type defined as follows:

```
enum COMMON_HEADER_FLAG { Ack, Pri, EM, A, TP };
```

It is possible, for example, to create a Config message using method (2), a constructor that takes all the input values separately, and allocate memory from the stack, this way:

```
ConfigMessage configMsg( 0x10, 1, 1, 0, 1, 664ull, 0, 1, 1, 0, 0 );
```

An example of creating a Query message using method (3) and allocating memory from the stack:

```
QueryMessage queryMsg( 0x10, 0x40000001, 1, 664ull, 0xC8400000 );
```

A Packet Redirect message is created using method (4), giving an input byte array and it's length as parameters and allocating memory from the heap. The data in the array *arr* is copied from another Packet Redirect message *prm1*:

```
char* arr;
prm1->writeByteArray( &arr );
PacketRedirectMessage* prm2 = new PacketRedirectMessage( arr, 80 );
```

If we receive a message from the network and are not sure what the message type is, we can use method (5) to generate a ForCES base class object from the message and check if the result is a valid message:

```
char arr[ 4096 ];
ForCESMessage* fm = makeMessage( arr, 4096 );
```

If the return value is NULL, then the message construction was not successful. Otherwise, the result is a valid ForCES message the type of which can be checked by using a class member function *getMessageType()*:

```
uint8_t messageType = fm->getMessageType();
```

The size of the array *arr* can be larger than the actual length of the message. It is a buffer to store the message whatever the length of the content is.

5.4.2 ForCES Message Destruction

When a ForCES message object is to be created, a constructor of the class of the object is called. Likewise, if a ForCES message object is to be deleted, the destructor of the class of the object is called. This happens when a *delete* command is called to that object or the program exits from the current context and the memory for the object has been allocated from the stack. Destructors are normally called automatically in these two situations, so the program does not need to call them directly. If an object has dynamically allocated memory in other objects or variables, they must be also deleted in the object's destructor.

All ForCES messages, except for the Heartbeat message, call a top-level TLV destructor from their own destructors. Depending on the structure of the top-level TLV, several sub-TLV destructors may be called. This means that the object destruction functions recursively through all the objects belonging to the actual message.

5.4.3 Object Member Variable Accessing Functions

For each of the object member variables, there is a function that sets the variable to the desired value and a function that gets the value of the variable at the current moment. Setter functions take the new value as a parameter, while getter functions return the current value of the variable as a return value. Below are the function prototypes to set the version field and get the message type field:

```
virtual void setVersion( uint8_t version );
virtual uint8_t getMessageType() const;
```

The setter and getter functions for the common variables for all inherited classes are placed in the base class. That is why we must use the word *virtual* before the return type in the function prototype. The getter function does not change anyting inside the object, so it needs the word *const*. Normally there is only one getter for each of the member variables. An exception is, for example, the flags field where one can get the whole field or only one flag value by specifying the flag name as a parameter to the flag getter function. Here are the prototypes for these functions:

```
virtual uint32_t getFlags() const;
virtual int getFlag( COMMON_HEADER_FLAG flagName ) const;
```

If the value can contain more than one of the same kind of variable type, then there are functions to add an item to and remove an item from the list. Because of encapsulation, the list implementation is not visible to the outside of the object. There is also a function that returns the number of items in the list. Here are examples of these function prototypes:

```
ParameterFieldTLV* getItem( int itemNo );
int getNumberOfItems() const;
void addItem( ParameterFieldTLV* parameterField );
void removeItem( int itemNo );
```

In this case, the member variable list contains Parameter Field TLV objects and the list item getter function returns a pointer to the TLV object returned. The getter function also takes the number of the item as a parameter. The calling program must release the memory by deleting the object, when it is not needed anymore.

To ensure that the number is not out of bounds, the number of items in the list should be checked before getting the value by using the number of items getter function. The last two function prototypes are for adding an item to the list and removing an item from the list. The item adding function places the new object or value to the end of the list. The item removing function removes an item which number is specified as a parameter.

The list implementation uses the Boost[1] library and a pointer vector data structure for dynamic object storing. Pointer vectors are easy to use and the Boost library takes control of all memory allocation and releasing operations. When adding or removing a list item, the object length field value is updated automatically so the calling program need not calculate the new length value. Still, it is recommended to check the validity of the object after operations that change the object length with the object validation checking function.

5.4.4 Object Validation

A ForCES message or a TLV/ILV object can be validated using a specific class member function. The validation function checks, for example, that the object length is correct by calculating recursively all member variables and objects in the object being validated. The function prototype is:

```
virtual bool isValid();
```

The function returns a *true* value of boolean type, if the object is valid, otherwise it returns a *false* value. Depending on the type of the object tested, all the needed tests are driven against the object. If all tests are passed, the object tested is found to be a valid object.

Message validation starts by checking the common header. The code for this operation is located in the parent class ForCESMessage to minimize duplicate code. All the fields of the common header are tested for valid values as specified in ETNA requirements [26]. Lastly, the message length calculation function is called and the result is compared to the message length field.

The next operation is to check the top-level TLV: if it exists, the validation function of the top-level TLV is called. It calls then recursively all the sub-TLVs inside the top-level TLV. If any of the validation functions returns a *false* boolean value, the message is not valid and the *false* value is returned to the calling program. Moreover, if the message should have a top-level TLV and there is none, the validation function returns *false*.

The TLV validation function checks, if the type of the TLV and the message length are valid values. Depending on the type of the TLV, the validation function can perform additional tests, for example, some TLV types must not have a sub-TLV and others can have it. If there exists any sub-TLV, the sub-TLVs' validation functions are called recursively.

5.4.5 Object Length Calculation

The length of a ForCES message, TLV, or ILV object can be calculated using a specific class member function. The function prototype is:

[1] http://www.boost.org/

```
virtual uint16_t calculateLength();
```

The return value is an unsigned 16-bit integer type. The length of an object is calculated as (8-bit) bytes, ie. octets.

If the object being calculated has sub-objects, the total length is calculated recursively by calling the length calculation function in the sub-object and the returning value is added to the length counter, finally giving the total length of the object. This value corresponds to the valuable data of the object, not the actual amount of data allocated from the memory for the object.

5.4.6 ForCES Message Printing

There is a class member function in all the ForCES message classes to print the contents of a message to the standard output. The prototype for the print function is:

```
virtual void print();
```

All data of the message is printed in hex format. If the data values are text characters or other characters that can be printed on the screen, they are printed also in ASCII format next to the hex data values. Otherwise, the period character is printed instead of the corresponding ASCII character. The printed line length is limited to 78 characters, so that the message can be printed using a 80 column terminal.

5.5 Development Environment

This section describes the development environment used along with the coding process.

5.5.1 Platform

The project chosed the Ubuntu Linux operating system and x86 processor architecture as the development environment platform. Also FreeBSD[2] was considered as an alternative. Also the Debian[3] distribution of Linux could have been a choice, but the Ubuntu[4] distribution of Linux was ultimately chosen, because it is easy to setup and use and there are more up-to-date libraries available for Ubuntu than for other Unix-like operating systems.

[2] The FreeBSD Project, http://www.freebsd.org/
[3] Software in the Public Interest, Inc., http://www.debian.org/
[4] Canonical Ltd., http://www.ubuntu.com/

5.5.2 Testing and Debugging

The testing of the source code was performed mainly by unit testing. A separate test program was written by the author which has a number of tests and a menu to choose a test from. The test program uses standard output to print every meaningful step of the running program and the contents of the message in a byte array format as a hex dump to check that the modifications to the message have been affected in the right places.

There is at least one test case for each of the message types. The test cases first create a new message object of the desired type, for example PacketRedirectMessage, and then add or set using a specific function the needed Type-Length-Value (TLV) structures inside the message object depending if there is a fixed area of memory reserved for the TLV to be set or the memory must be allocated dynamically by adding a new TLV. There can be one or multiple TLVs inside another TLV depending on the message structure. The aim is to use all TLV levels even though they are not necessarily needed to compose a valid message to test the right functional operation. The composed message is printed to the standard output using the print function.

The test cases then call a function that writes the message content to a byte array format and then generates a second instance of the message using a function that takes the byte array and the length of the message in bytes as arguments. The function returns a message of a ForCESMessage base class type that must be dynamically casted to the actual message type in order to access the data inside the object other than common header, because the message structure depends on the type of the message. Lastly, the generated message is tested so that it contains the correct values.

The second way to generate a message from a byte array is to call the constructor of the desired message class with an input byte array and the length of the array as arguments. This method can be used when we know the type of the message already. Some of the test cases utilize this method as well. The generated message is printed at the end of the test case to compare the contents of the second instance to the first.

There are also three debug levels that can be activated in the source code. There is a definition *DEBUG_LEVEL* in the ForCES Message base class header file *ForCESMessage.h* that can have three values: *DEBUG_LEVEL_OFF*, *DEBUG_LEVEL_NORMAL* and *DEBUG_LEVEL_ALL*. Debugging level *DEBUG_LEVEL_OFF* prints no debug prints. Some essential debug information is printed with level *DEBUG_LEVEL_NORMAL*. The most accurate debug level, when all the debug prints are printed, is *DEBUG_LEVEL_ALL*. The idea behind this was to have a simple and fast way to add debug prints to the source code with the ability to disable them when not needed anymore. Also the GNU Debugger[5] was used a few times when there were more cumbersome bugs.

For memory leakage inspection, the Valgrind[6] Memcheck tool was used to reveal

[5] The GDB developers, http://www.gnu.org/software/gdb/

bugs that leak random access memory. The tool was found very valuable for locating many problems in source code, usually missing memory free calls.

5.6 Coding process

The work was started by designing the code structure: the needed classes, functions and variables. After that the basic implementation of classes were quite straightforward. However, there came a lot of small changes in the code during the project and many of the changes affected all or several classes. The number of TLV classes and their relations to other classes, and the number of details in the requirement specification needed special care while writing the code. All the files, classes and functions were commented using Doxygen style comments and that also took a lot of time to make a good documentation.

There were some problems with the way the Boost library handles the pointer vector memory allocations: adding an object to a pointer vector needs the object memory to be allocated from the heap and no stack allocation was possible. The Boost library releases the memory allocated for the pointer vector objects when the objects are removed from the pointer vector or the object containing the pointer vector is deleted, so memory allocations and deallocations are not done in the same context. The author did not have a relevant solution to this problem so this functioning was decided to be accepted and documented.

5.7 Summary and Conclusions

The implementation of the ForCES protocol was done following the ETNA requirements specification for the ForCES messages. A C++ library was implemented and a separate test program written by the author to test the library functionality. The library has 31 classes with 15 436 lines of source code.

The object-oriented programming paradigm was used from the beginning in the design of the software: the data in the objects is stored in the private member functions and can be accessed using specific functions. There are functions to convert the object data to a byte array form to be transported to the network and to read a byte array, that contains a message from the network, and write that data to a message object. In addition, there are object validation and length calculation functions. In the ForCES Message class there is also a print function to print contents of the message to the screen.

Testing was performed using the common test program that has different tests to cover the functionality of the library. There are three debug levels in the code to print run-time debug information. Also the GNU Debugger was used a few times. Comments were written using Doxygen documentation system.

[6] The Valgrind developers, http://valgrind.org/

The implementation is working well and has all the needed functionality to generate ForCES messages and process them. There are still some things that can be done to improve the usability of the library such as adding functions to insert an item to a selected place on the pointer vector. The coding work was first estimated much less than it was finally, because there were many small and bigger changes to the source code and one change needed modifications to other classes also, if not all of them at a time.

Chapter 6

Evaluation of the Implementation

In the first section of this chapter the used evaluation methods are described. The second section contains the performance measurements: the planning of the measurements, the tools and the environment used in the measurements and the actual measurements in details. The results are discussed in the third section and the fourth section has the summary and conclusions.

6.1 Evaluation Methods

Because the implementation is used in the message transport, the speed of the message construction and the object data reading is important. Two things can be measured from these: the time elapsed in the message construction and how much memory was used during an operation. These facts describe the performance of the implementation.

The tools used measuring the performance are the test program *tester.cpp* written by the author and the Valgrind Memcheck tool. The ForCES message construction time was measured with the test program and the memory usage with the Valgrind Memcheck tool.

6.2 Performance Measurements

The performance measurements were done in two ways: the one with the normal compiling options and the other with code execution speed optimizing with the *-O2*-flag given to the compiler to optimize the code as much as possible. The processor's other load was kept steady and as minimal as possible, so that the measurements would be comparable together. A few times there were slightly divergent values measured. When this happened, the divergent values were excluded from the results not to distort them.

Four different message types were selected in the measurements. ForCES test message 1 is a small message with only a common header and no Type-Length-Value (TLV) data structures. ForCES test message 2 is a short Config message with only a few TLVs and minimal data. ForCES test message 3 is a long Config message with a few TLVs and much data. ForCES test message 4 is a long Config message with lots of TLVs included. The structures of the test messages are designed so that the test messages 2 and 3 have the same amount of TLVs included and the test messages 3 and 4 have the same length in bytes.

The unoptimized ForCES message construction time measurements and memory allocations done during the code execution are shown in Tables 6.1, 6.3, 6.5, and 6.7. The corresponding code execution time measurements and memory allocations of the optimized code are shown in Tables 6.2, 6.4, 6.6, and 6.8. The tables have the values normalized to individual messages. The original measurement results can be seen on Appendix D.

The values for the memory allocations include only the essential memory allocations: other memory allocations, such as temporary arrays used and the memory allocated for the program start are excluded from the values.

6.2.1 Planning of the Measurements

The ForCES message construction operation was selected to be measured in the tests because it is more complicated operation than the object data reading. That is because the Boost library is used to store dynamically TLV objects inside upper level TLVs or message objects in the message construction phase and it is expectable that these Boost operations take more time than if we are only reading the object data values. The message destruction operations were left out from the time measurements also. The test cases handle several messages at a time and an average value can be counted from the results to omit errors resulting from unessential causes.

The designed performance tests have the amount of messages to be constructed set so that the total processing time is about a few seconds or less to keep the testing time reasonable. The tests are done five times with five different lineal increasing amounts of messages. The results were then normalized to correspond to the values for one message to five messages so that the individual message processing times and memory allocations can be easily seen.

6.2.2 Tools Used in the Measurements

The ForCES message library was performance tested with the test program *tester.cpp* written by the author. It is a common test program containing also functional tests that were run when the general functionality of the ForCES message library was tested.

The message construction time was measured by specific tests in the test pro-

gram. The tests utilize *clock()* function that resides in the C time library [34, p. 906]. The function prototype is:

```
clock_t clock(void);
```

The return value of type *clock_t* is the processor time at the moment the function was called. The current time, when the test starts, is stored and compared to the time after the code to be measured is executed. The difference of these two values is the execution time of the measured code.

The Valgrind Memcheck tool was utilized for checking the number of memory allocations and the total amount of memory allocated in bytes per program run. Valgrind can be executed from the command line with the needed arguments by typing:

```
valgrind --leak-check=full --max-stackframe=2000016 ./tester
```

The *--leak-check=full* argument tells Valgrind to search for memory leaks at the program exit and print the number of the memory allocations done, the total amount of memory allocated from the heap in bytes, and the amount of memory released during the program execution. The *--max-stackframe=2000016* argument was needed to increase the stack frame limit because the limit was not high enough for the fourth measurement with lots of TLVs. Without this argument, there will not be enough space in the stack during the program execution and a stack overflow will result [35, p. 197].

6.2.3 Measurement Environment

The operating system used in measurements was Ubuntu Release 8.04 with Linux Kernel 2.6.24-21-generic. The hardware was a standard PC compatible with an Intel® Core™ 2 Duo E6600 2.40 GHz processor with two cores and two gigabytes of random access memory. The used test program did not utilize threads so only one central processing unit was used at a time.

6.2.4 Measurements with ForCES Test Message 1

The first ForCES message used for measurements was selected to be as short as possible. The only ForCES message that has no TLVs, just a common header, is the Heartbeat message. The length of the common header is 24 bytes, so that will be the length of this message also.

The average message construction time measurements and memory allocations from the heap using non-optimized code are shown in Table 6.1. The corresponding measurements for optimized code are shown in Table 6.2.

Table 6.1: ForCES test message 1 construction time measurements normalized and related memory allocations with non-optimized code.

Average time (μs)	Standard deviation (μs)	Number of messages	Number of memory allocations	Memory allocated (bytes)
8.16	0.06	1	1	32
16.48	0.56	2	2	64
24.26	0.70	3	3	96
31.78	0.57	4	4	128
39.88	0.43	5	5	160

Table 6.2: ForCES test message 1 construction time measurements normalized and related memory allocations with optimized code.

Average time (μs)	Standard deviation (μs)	Number of messages	Number of memory allocations	Memory allocated (bytes)
5.54	0.11	1	1	32
10.86	0.21	2	2	64
15.78	0.16	3	3	96
21.12	0.74	4	4	128
26.00	0.37	5	5	160

6.2.5 Measurements with ForCES Test Message 2

The second ForCES message used for measurements was selected to be a short message with a few TLVs in the payload. For clarity, this message and the rest of the messages used for measurements would be good to be the same type. Because a Config message can have several TLVs with multiple levels, that message type was selected for the rest of the measurements.

The Config message used for this measurement has an LFBselect TLV as the

Table 6.3: ForCES test message 2 construction time measurements normalized and related memory allocations with non-optimized code.

Average time (μs)	Standard deviation (μs)	Number of messages	Number of memory allocations	Memory allocated (bytes)
304	5.5	1	19	308
522	8.4	2	38	616
722	4.5	3	57	924
946	13.4	4	76	1232
1166	11.4	5	95	1540

Table 6.4: ForCES test message 2 construction time measurements normalized and related memory allocations with optimized code.

Average time (μs)	Standard deviation (μs)	Number of messages	Number of memory allocations	Memory allocated (bytes)
130	7.1	1	19	308
212	8.9	2	38	616
284	11.4	3	57	924
364	11.4	4	76	1232
442	11.0	5	95	1540

top-level TLV. Inside that, there is one Config TLV with two Config Data TLVs in the value field. The first Config Data TLV has two Parameter Field TLVs of length eight and the second Config Data TLV has two Parameter Field TLVs of length four, that is the value field of them being empty. This is not a reasonable situation in practise, but for testing purposes it is a good choice, because it is a special case that is good to be tested also. The length of this message is 80 bytes.

The average message construction time measurements and memory allocations from the heap using non-optimized code are shown in Table 6.3. The corresponding measurements for optimized code are shown in Table 6.4.

6.2.6 Measurements with ForCES Test Message 3

A long Config message with a few TLVs was used in this measurement. The structure of this Config message is the same as in the previous measurement, but the length of the first two Parameter Field TLVs is 480 bytes instead of eight. The total length of this message is 1024 bytes.

The average message construction time measurements and memory allocations from the heap using non-optimized code are shown in Table 6.5. The corresponding measurements for optimized code are shown in Table 6.6.

Table 6.5: ForCES test message 3 construction time measurements normalized and related memory allocations with non-optimized code.

Average time (μs)	Standard deviation (μs)	Number of messages	Number of memory allocations	Memory allocated (bytes)
352	13.0	1	19	1252
624	9.0	2	38	2504
896	9.0	3	57	3756
1162	11.0	4	76	5008
1438	14.8	5	95	6260

Table 6.6: ForCES test message 3 construction time measurements normalized and related memory allocations with optimized code.

Average time (μs)	Standard deviation (μs)	Number of messages	Number of memory allocations	Memory allocated (bytes)
156	13.4	1	19	1252
268	11.0	2	38	2504
368	14.8	3	57	3756
488	11.0	4	76	5008
596	8.9	5	95	6260

6.2.7 Measurements with ForCES Test Message 4

The last measurement was done using a long Config message with lots of TLVs. The structure of this Config message is almost the same as in the second measurement, but there are in total 120 Parameter Field TLVs of the length eight instead of only two. The total length of this message is 1024 bytes which is the same length used in the third measurement to enable easier comparison of the time and memory usage between the measurements.

Table 6.7: ForCES test message 4 construction time measurements normalized and related memory allocations with non-optimized code.

Average time (μs)	Standard deviation (μs)	Number of messages	Number of memory allocations	Memory allocated (bytes)
5020	83	1	437	6232
9320	217	2	874	12464
13500	71	3	1311	18696
17760	114	4	1748	24928
22000	283	5	2185	31160

Table 6.8: ForCES test message 4 construction time measurements normalized and related memory allocations with optimized code.

Average time (μs)	Standard deviation (μs)	Number of messages	Number of memory allocations	Memory allocated (bytes)
2200	122	1	437	6232
3980	84	2	874	12464
5560	55	3	1311	18696
7360	89	4	1748	24928
9020	45	5	2185	31160

The average message construction time measurements and memory allocations from the heap using non-optimized code are shown in Table 6.7. The corresponding measurements for optimized code are shown in Table 6.8.

6.3 Results of the Measurements

The results of the measurements are handled from two points of view: the amount of memory allocated in the message construction phase and the measured time elapsed during the operation.

6.3.1 Analysis of the Memory Allocations

It can be seen from the values from the ForCES test message 1 measurements (Table 6.1) that there is one memory allocation from the heap for each message. Memory is allocated 32 bytes for one Heartbeat message, so if we take into consideration that the length of the Heartbeat message is 24 bytes, there is eight bytes more memory allocated than the length of the message is as can be seen in Table 6.9. The number of the memory allocations and the memory allocated grow linearly as the number of the test messages increase.

The ForCES test message 2 has eight TLVs. From Table 6.3 we can see that there were 19 memory allocations for that message. That is because the Boost library does one extra memory allocation per each of the TLVs. Two memory allocations are for the parameter values and one for the message itself. That makes in total nineteen memory allocations. The difference between the memory allocated for the ForCES test message 2 and the message length, ie. overhead, is 228 bytes (Table 6.9). The extra bytes are from the memory allocations made by the test program and Boost library. The exact amounts of bytes allocated in each of the operations are beyond the scope of this thesis and will not be investigated more here.

The memory usage for the ForCES test message 3 (Table 6.5) is quite similar to the previous message. The length of the message is only bigger, while the number of the memory allocations and the amount of memory allocated minus the length of

Table 6.9: Comparison between message length and memory allocated in measurements with ForCES test messages 1–4.

Test message name	Number of TLVs in a message	Message length (bytes)	Memory allocated (bytes)	Overhead (bytes)	Overhead (%)
ForCES test message 1	0	24	32	8	33
ForCES test message 2	8	80	308	228	285
ForCES test message 3	8	1024	1252	228	22
ForCES test message 4	185	1024	6232	5208	509

the message are the same as with the ForCES test message 2 (see Table 6.9).

The ForCES test message 4 has 185 TLV data structures and that increases the memory usage significantly (Table 6.7). The difference between the memory allocated for each of the messages and the message length is now 5208 bytes (Table 6.9). That is over five times the length of the actual message (1024 bytes). The memory allocations overhead compared to the message network length is calculated in the last column of Table 6.9.

6.3.2 Analysis of the Message Construction Times

In Figure 6.1, there is a bar diagram showing the difference between the construction times for one ForCES test message.

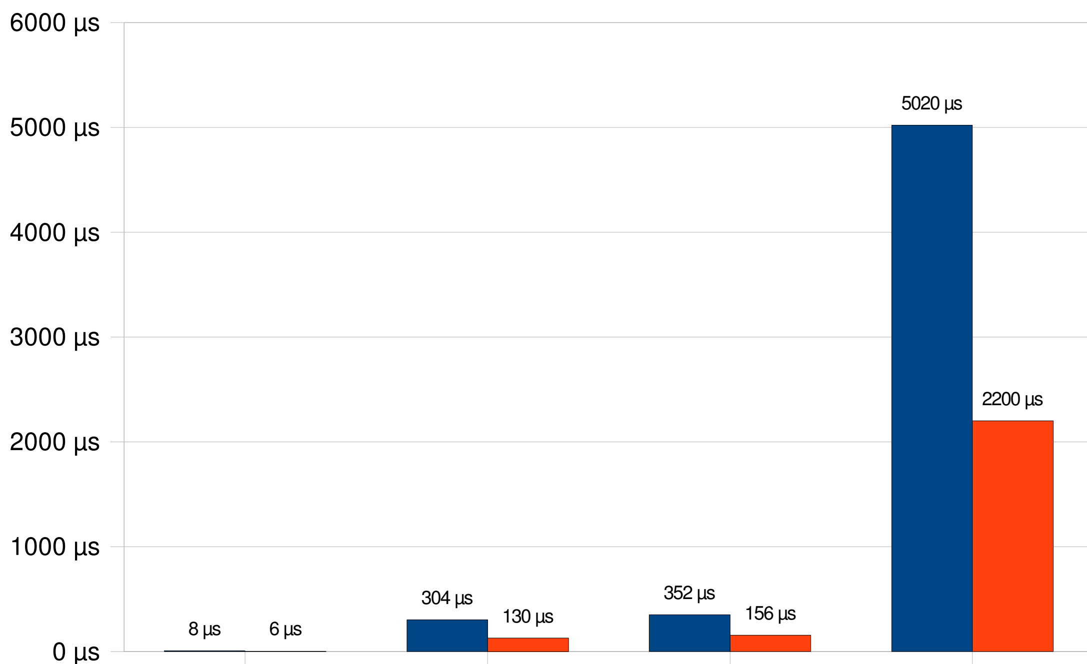

Figure 6.1: ForCES test messages 1–4 construction times with one message compared.

Figure 6.2 shows the same bars zoomed so that the small differences between messages 2 and 3 are seen better from the diagram. The blue bar shows the result for the unoptimized code and the red bar shows the result for the optimized code. The first almost invisible bar pair is for the Heartbeat message and the rest are for the Config messages. From the picture it can be seen that a long message with lots of TLVs takes significantly more time than a long message with a few TLVs even though they are of the same length. This is because the memory for the objects is allocated dynamically by the test program and the Boost library, and that seems to take much time.

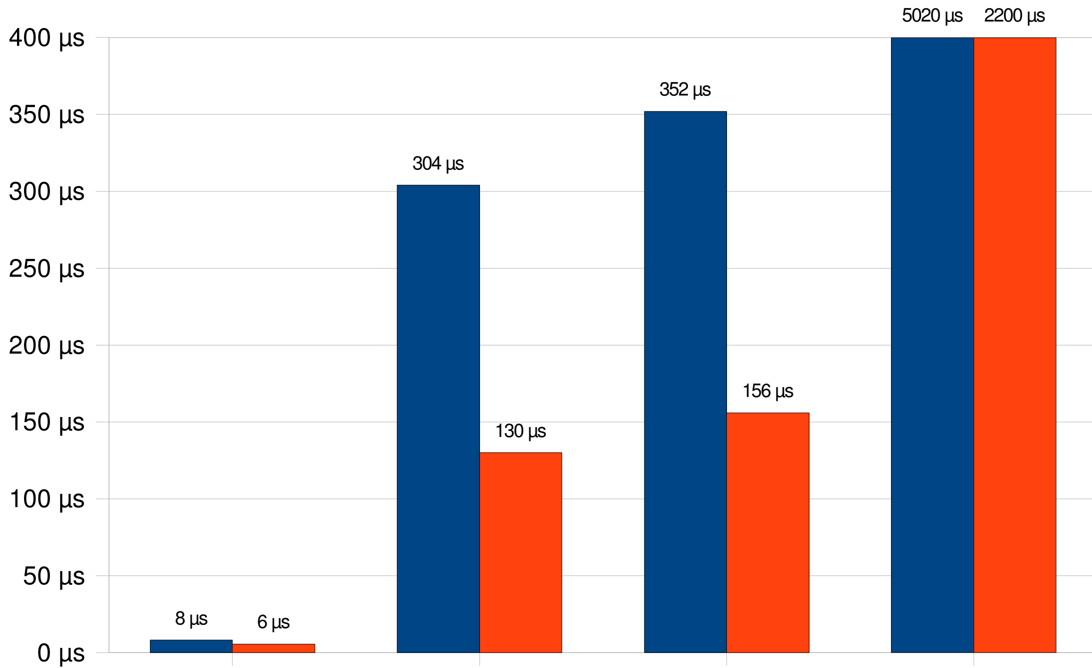

Figure 6.2: ForCES test messages 1–4 construction times with one message compared (zoomed).

The code optimization with the *-O2* parameter given to the compiler reduces the message construction times to a half or less compared to the unoptimized code. This can be seen from all of the four measurements.

From Figure 6.3 it can be seen from the graph that the ForCES test message processing times increase linearly. If we continue the line one more step to the left (x=0) by approximating, the gap between the origin and the line in y-axis direction is the offset time.

The offset time and the percentage of the offset compared to the average of the differences of the neighbor values for the ForCES test messages 1–4 are calculated in Table 6.10. Δ_1 is the distance of the first value from the x-axis, ie. the value itself. Δ_2–Δ_5 are the differences of the other values compared to the previous value. The average value is calculated for only Δ_2–Δ_5, because Δ_1 includes the offset so it

Table 6.10: ForCES test message construction time offsets calculated.

Msg #	Δ_1 (μs)	Δ_2 (μs)	Δ_3 (μs)	Δ_4 (μs)	Δ_5 (μs)	Average of Δ_2–Δ_5 (μs)	St. Dev. (μs)	Offset (μs)	Offset (%)
1	8.16	8.32	7.78	7.52	8.10	7.93	0.35	0.23	2.9
2	304	218	200	224	220	216	11	89	41
3	352	272	272	266	276	272	4	81	30
4	5020	4300	4180	4260	4240	4245	50	775	18

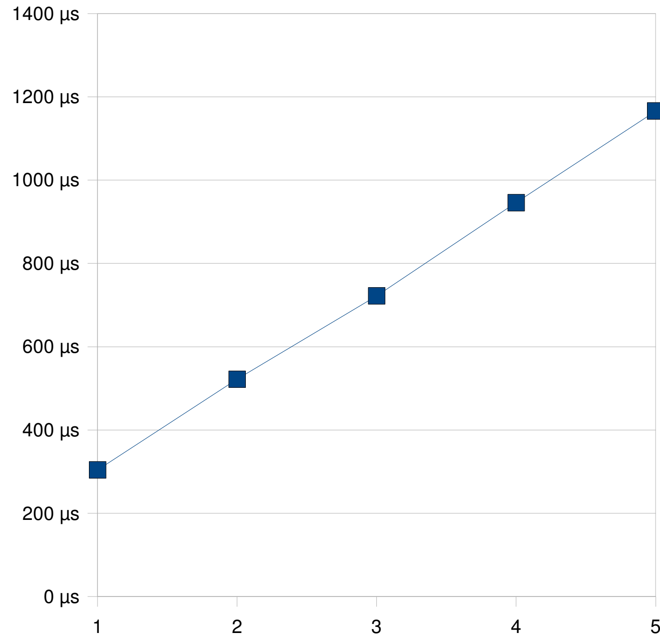

Figure 6.3: ForCES test message 2 construction times with 1–5 messages processed.

must be excluded. The next column shows the standard deviation for the calculated average values.

From the offset percentage column of this table we can see that for the Heartbeat messages the message construction offset time is nearly negligible. The relative offset is the largest for the short Config message with a few TLVs and gets smaller as the message complexity increases.

6.4 Summary and Conclusions

Four different kinds of ForCES test messages have been used in the performance testing in the evaluation of the ForCES message library. The first ForCES test message was a short one with only the common header. The second one was a message with a few short TLVs. The third message was a much longer message with a few TLVs and the fourth message was a long message with lots of TLVs.

The memory usage was calculated using the Valgrind memory checking tool and compared between different test messages. This includes both the number of memory allocations and the amount of memory allocated during the message construction. Also the message construction time was measured using the *clock()* function that is included in the C time library.

There were significant differences between the results. Both the construction time and the amount of memory allocated increased significantly as the number of

TLVs increased. The reason for this phenomenon is the memory allocation operations that take time. The code optimization reduced the processing times to a half or less compared to the unoptimized code.

As the number of TLVs increase in a ForCES message, the processing time and the memory usage increase significantly. However, that would be not a problem because the absolute time and memory used are still quite insignificant compared to the capacity of the computers today. The nature of the ForCES messages is that they are, for example, used to make associations or configuration changes so they do not need so much capacity in processing.

The results were quite reasonable and it was interesting to see how much the optimization of the code reduced the processing times of the tested messages.

Chapter 7

Summary and Conclusions of the Study

In this chapter there is a summary of all the work done and conclusions. Finally, some future work ideas are presented.

7.1 Summary

This Master's thesis has dealt with a software implementation and evaluation of a network element control protocol that has been a part of the Ethernet Transport Networks, Architectures of Networking (ETNA) project which was co-funded by the European Commission in the Seventh EU Framework Program of Research and Technological Development (FP7). The network element control protocol chosen to be implemented was the Forwarding and Control Element Separation (ForCES) protocol. Two other network element control protocols were also investigated and their features compared with the protocol utilized by ETNA and with each other. These protocols were general switch management protocol (GSMP) and OpenFlow.

The original ForCES protocol specification written by the IETF network working group was modified for ETNA needs: the PATH-DATA-TLV layer was removed to simplify the implementation and several new TLV type values were introduced. The implementation was done against the modified specification and compiled to a C++ library. The ForCES protocol implementation was part of the ETNA proof-of-concept network element model that was tested together with BGU's forwarding element part ForCES implementation in the project integration phase during autumn 2009. The evaluation of the implementation included some performance measurements.

7.2 Conclusions

The ETNA project requirements for the ForCES protocol implementation were well specified, so the coding process was quite straightforward. The implementation was found to be working well without problems in the project integration phase where all the software modules from Aalto University and Ben Gurion University were working together.

As a result, it can be said that the protocol chosen was a good decision eventually. It has all the functionality for the CE–FE communication needed in ETNA.

The implementation of the basic framework for the ForCES protocol took a few months and the commenting, debugging, and modifying of the code for better matching up to the ForCES protocol specification adapted for the ETNA project took also a lot of time because of the high amount of classes in the C++ code. The commenting of all the code was done using the Doxygen[1] documentation system.

The ForCES message library has now the basic functionality so that it can be used to construct a ForCES message from a byte array containing a message from the network and read a ForCES message object data to a byte array to be transported to the network. The library can handle all ten ForCES message types that are specified in the IETF ForCES specification.

7.3 Future Work

If an object can have more than one of the same kind of TLV, now TLVs can be added to the end and the desired TLV number can be removed. Some future work could be to make adding possible to any place in the data structure. A function that finds the first or next item with a given value or type could be useful also. The Boost library has also other useful features that can be utilized in making better functions that handle Boost pointer vector items.

The PATH-DATA-TLV was not utilized in the ETNA implementation. That could be implemented in the future versions if it seems to be practical. For multi-core processor systems thread utilizing could be useful. That will need some additional code to use and control multiple threads.

[1] http://www.doxygen.org/

Bibliography

[1] The Miniwatts Marketing Group, *Internet World Stats – Usage and Population Statistics*, updated: 2009-09-30, accessed: 2009-11-17, available: http://www.internetworldstats.com/stats.htm.

[2] International Telecommunication Union (ITU), *ICT Statistics Newslog – Internet traffic is growing fast — but capacity is keeping pace*, updated: 2008-09-05, accessed: 2009-11-17, available: http://www.itu.int/ITU-D/ict/newslog/Internet+Traffic+Is+Growing+Fast+But+Capacity+Is+Keeping+Pace.aspx.

[3] Comer, D. E., *Internetworking with TCP/IP – Volume I: Principles, Protocols, and Architecture, Third Edition*, Prentice-Hall, Inc., 1995, ISBN 0-13-216987-8.

[4] Hudson, D., *Next-generation Ethernet and network intelligence: Foundations for future networks*, Nortel Technical Journal, Issue 4, pp. 1–6.

[5] Tanenbaum, A. S., *Computer Networks*, Prentice-Hall International Editions, Prentice-Hall, Inc., 1981, ISBN 0-13-164699-0.

[6] Allard, R., *Ethernet OAM and resiliency: Making Ethernet suitable for carrier operations*, Nortel Technical Journal, Issue 4, pp. 28–29.

[7] ETNA Consortium, *ETNA*, ETNA Consortium WWW home page, accessed: 2009-11-04, available: http://www.ict-etna.eu/.

[8] ETNA Consortium, *Ethernet Transport Networks, Architectures of Networking, Work Package 6 Deliverable 6.1, Showcase Report*, 2010-01-15, accessed: 2010-05-10, available: http://www.ict-etna.eu/documents/ETNA%20D6.1%20Showcase%20Report%20V1.4.pdf.

[9] ETNA Consortium, *Ethernet Transport Networks, Architectures of Networking, Work Package 1 Deliverable 1.1, Requirements, specification and analysis*, 2008-06-30, accessed: 2010-05-10, available: http://www.ict-etna.eu/documents/ETNAWP1FinalD1.1.pdf.

[10] Kantola, R., Luoma, M., and Ilvesmäki, M., *Routed End to End Ethernet – RE2EE*, 2007-02-26.

[11] Ryynänen, J., *Routed End-to-End Ethernet Network – Proof of Concept*, Master's thesis, Aalto University School of Science and Technology, Department of Communications and Networking, Espoo, 2008.

[12] Toropainen, T., *A Routing Protocol for Ethernet Transport*, Master's thesis, Helsinki University of Technology, Espoo, 2008.

[13] Khosravi, H., and Anderson, T., Eds., *Requirements for Separation of IP Control and Forwarding*, RFC 3654, November 2003.

[14] Allan, D., and Bragg, N., *Taking control: The evolving role of the control and data planes*, Nortel Technical Journal, Issue 4, pp. 25–32.

[15] Calzia, S., Bonsall, L., *Building a highly available enterprise network*, Lightwave, Issue April 2010, a network magazine, pp. 13–16, PennWell Corporation, accessed: 2010-05-10, available: http://online.qmags.com/LW0410/.

[16] Doria, A., Haas, R., Hadi Salim, J., Khosravi, H., and Wang, W. M., *ForCES Protocol Specification*, Internet-Draft draft-ietf-forces-protocol-22 (work in progress), March 2009.

[17] Yang, L., Dantu, R., Anderson, T., and Gopal, R., *Forwarding and Control Element Separation (ForCES) Framework*, RFC 3746, April 2004.

[18] Jin, R., and Wang, W, *Research and Implementation of SNMP in ForCES Framework*, IEEE publication, 2007.

[19] Halpern, J., and Salim, J. H., *Forwarding and Control Element Separation (ForCES) Forwarding Element Model*, RFC 5812, March 2010.

[20] Dutton, H. J. R., Lenhard P., *Asynchronous Transfer Mode (ATM) Technical Overview, Second Edition*, Prentice-Hall, Inc., 1995, ISBN 0-13-52044-5.

[21] Doria, A., Hellstrand, F., Sundell, K., and Worster, T., *General Switch Management Protocol (GSMP) V3*, RFC 3292, June 2002.

[22] Doria, A., and Sundell, K., *General Switch Management Protocol (GSMP) Applicability*, RFC 3294, June 2002.

[23] McKeown, N., Anderson, T., Balakrishnan, H., Parulkar, G., Peterson, L., Rexford, J., Shenker, S., and Turner, J., *OpenFlow: Enabling Innovation in Campus Networks*, OpenFlow whitepaper, 2008-03-14, accessed: 2009-12-17, available: http://www.openflowswitch.org/documents/openflow-wp-latest.pdf.

[24] Das, S., Parulkar, G., and McKeown, N., *Simple Unified Control for Packet and Circuit Networks*, accessed: 2010-05-10, available: http://www.openflowswitch.org/wp/wp-content/uploads/2009/05/openflow_ucp_submitpaper.pdf.

[25] Sherwood, R., Gibb, G., Kok-Kiong, Y., Appenzeller, G., Casado, M., McKeown, N., and Parulkar, G., *FlowVisor: A Network Virtualization Layer*, 2009-10-14, accessed: 2009-11-19, available: http://openflowswitch.org/downloads/technicalreports/openflow-tr-2009-1-flowvisor.pdf.

[26] ETNA Consortium, *ForCES Messages*, The ETNA Control Plane ForCES Messages requirements specification, updated: 2008-09-15.

[27] Gray, J., *Notes on database operating systems. In Operating Systems: An Advanced Course. Lecture Notes in Computer Science, Vol. 60*, pp. 394–481, Springer-Verlag, 1978.

[28] Lamminen, O.-P., Luoma, M., Nousiainen, J., and Taira, T., *Control Plane for Carrier-Grade Ethernet Network*, submitted for BIPN'09.

[29] ETNA Consortium, *Ethernet Transport Networks, Architectures of Networking, Work Package 4 Deliverable 1.1, Requirements, Implementation, Architecture, and Functionality*, 2008-11-30, accessed: 2010-05-10, available: http://www.ict-etna.eu/documents/ETNA%20Report%20of%20the%20requirements,%20implementation%20%20architecture%20and%20models%20-D4.1-R1.pdf.

[30] Nousiainen, J., *Management of Carrier Grade Intra-Domain Ethernet*, Master's thesis, Aalto University School of Science and Technology, Department of Communications and Networking, Espoo, 2010.

[31] Awduche, D., Berger, L., Gan, D., Li, T., Srinivasan, V., and Swallow, G., *RSVP-TE: Extensions to RSVP for LSP Tunnels*, RFC 3209, December 2001.

[32] Taira, T., *A Signaling System for Ethernet Transport*, Master's thesis, Aalto University School of Science and Technology, Department of Communications and Networking, Espoo, 2010.

[33] Schach, S. R., *Object-Oriented & Classical Software Engineering, 6th Edition*, International Edition, McGraw-Hill, 2005, ISBN 0-07-111191-3.

[34] Stroustrup, B., *The C++ Programming Language, Special Edition*, Addison Wesley, 2000, ISBN 0-201-70073-5.

[35] Hoffman, A., *PC Assembly Language: Step by Step, A complete beginners guide to learning and applying assembly language*, A Micro Application Book, Abacus Software Inc, 1990, ISBN 1-55755-096-4.

Appendix A

Code Snippets

ForCESMessage.h

```
/** \file    ForCESMessage.h
 *  \brief  ForCES Message base class declaration.
 *  \author Jan Grondahl
 *  \date    19.11.2008
 *
 * This file contains class declaration of ForCES Message base class. All
 * ForCES Messages are derived from this class.
 */

#ifndef FORCESMESSAGE_H_
#define FORCESMESSAGE_H_

/// Message type values
#define ASSOCIATION_SETUP_MESSAGE_TYPE             0x01
#define ASSOCIATION_SETUP_RESPONSE_MESSAGE_TYPE    0x11
#define ASSOCIATION_TEARDOWN_MESSAGE_TYPE          0x02
#define CONFIG_MESSAGE_TYPE                        0x03
#define CONFIG_RESPONSE_MESSAGE_TYPE               0x13
#define QUERY_MESSAGE_TYPE                         0x04
#define QUERY_RESPONSE_MESSAGE_TYPE                0x14
#define EVENT_NOTIFICATION_MESSAGE_TYPE            0x05
#define PACKET_REDIRECT_MESSAGE_TYPE               0x06
#define HEARTBEAT_MESSAGE_TYPE                     0x0F

/// Common header field offset values.
#define COMMON_HEADER_VERSION_OFFSET             0x00
#define COMMON_HEADER_MESSAGETYPE_OFFSET         0x01
#define COMMON_HEADER_MESSAGELENGTH_OFFSET       0x02
#define COMMON_HEADER_SOURCESUBID_OFFSET         0x04
#define COMMON_HEADER_DESTINATIONSUBID_OFFSET    0x08
#define COMMON_HEADER_CORRELATOR_OFFSET          0x0C
#define COMMON_HEADER_FLAGS_OFFSET               0x14
#define FIRST_TOPLEVELTLV_OFFSET                 0x18
#define COMMON_HEADER_SIZE                       0x18

/// Common header field bit offset values
#define COMMON_HEADER_VERSION_BIT_OFFSET    0x04 ///< Version
#define COMMON_HEADER_STS_BIT_OFFSET        0x1E ///< sTS
#define COMMON_HEADER_DTS_BIT_OFFSET        0x1E ///< dTS
#define COMMON_HEADER_ACK_FLAG_BIT_OFFSET   0x1E ///< Ack flag
#define COMMON_HEADER_PRI_FLAG_BIT_OFFSET   0x1B ///< Pri flag
#define COMMON_HEADER_EM_FLAG_BIT_OFFSET    0x16 ///< EM flag
#define COMMON_HEADER_A_FLAG_BIT_OFFSET     0x15 ///< A flag
```

```
#define COMMON_HEADER_TP_FLAG_BIT_OFFSET  0x13 ///< TP flag

/// Common header field bit size values
#define COMMON_HEADER_STS_BIT_SIZE       0x02 ///< sTS
#define COMMON_HEADER_DTS_BIT_SIZE       0x02 ///< dTS
#define COMMON_HEADER_ACK_FLAG_BIT_SIZE 0x02 ///< Ack flag
#define COMMON_HEADER_PRI_FLAG_BIT_SIZE 0x03 ///< Pri flag
#define COMMON_HEADER_EM_FLAG_BIT_SIZE  0x02 ///< EM flag
#define COMMON_HEADER_A_FLAG_BIT_SIZE   0x01 ///< A flag
#define COMMON_HEADER_TP_FLAG_BIT_SIZE  0x02 ///< TP flag

/// Current software version
#define CURRENT_VERSION 1

/// "Invalid" values used for initializing variables
#define INVALID_VALUE_8BIT  0xFFu                  ///< uint8_t (char)
#define INVALID_VALUE_16BIT 0xFFFFu                ///< uint16_t (short)
#define INVALID_VALUE_32BIT 0xFFFFFFFFu            ///< uint32_t (int/long)
#define INVALID_VALUE_64BIT 0xFFFFFFFFFFFFFFFFull  ///< uint64_t (long long)

/** \brief Debug levels for testing and debugging purposes.
 *
 * Debug levels for testing and debugging purposes.
 *
 * DEBUG_LEVEL_OFF    = No test prints.
 * DEBUG_LEVEL_NORMAL = Normal test prints.
 * DEBUG_LEVEL_ALL    = All test prints.
 */
#define DEBUG_LEVEL_OFF    0 ///< No debug prints.
#define DEBUG_LEVEL_NORMAL 1 ///< Some debug prints.
#define DEBUG_LEVEL_ALL    2 ///< All debug prints.

/// Current debug level
#define DEBUG_LEVEL DEBUG_LEVEL_OFF

#include <inttypes.h>
#include <stdint.h>
#include "bit_ops.h"

/** \namespace forcesmessage
 *  \brief ForCES message creation and accessing functions.
 *
 * Forcesmessage includes classes and functions to create and access
 * different types of ForCES messages used to transport between ETNA Control
 * Plane and Forwarding Plane.
 */
namespace forcesmessage {

/// Common header flags
enum COMMON_HEADER_FLAG { Ack, Pri, EM, A, TP };

/** \brief This class specifies the base class ForCESMessage.
 *
 * This class specifies the base class ForCESMessage.
 */
class ForCESMessage {

public:

    /// Constructors

    /** \brief The default constructor. */
    ForCESMessage();

    /** \brief A constructor that takes every single value as it's own
     *         parameter. */
    ForCESMessage(
            int version,
```

```
            int sTS,
            uint32_t sourceSubID,
            int dTS,
            uint32_t destinationSubID,
            uint64_t correlator,
            int flag_Ack,
            int flag_Pri,
            int flag_EM,
            int flag_A,
            int flag_TP );

    /** \brief A constructor that takes values as whole bytes, words, long
     *         words and long long words. */
    ForCESMessage(
            uint8_t version,
            uint32_t sourceSubID,
            uint32_t destinationSubID,
            uint64_t correlator,
            uint32_t flags );

    /** \brief A constructor that reads a byte array. */
    ForCESMessage( const char* array, int size );

    /// Destructor

    /** \brief Destructor. */
    virtual ~ForCESMessage();

    /// Getters

    /** \brief Returns the version number. */
    virtual uint8_t getVersion() const;

    /** \brief Returns the message type. */
    virtual uint8_t getMessageType() const;

    /** \brief Returns the message length. */
    virtual uint16_t getMessageLength() const;

    /** \brief Returns the source sub-ID. */
    virtual uint32_t getSourceSubID() const;

    /** \brief Returns the destination sub-ID. */
    virtual uint32_t getDestinationSubID() const;

    /** \brief Returns the correlator. */
    virtual uint64_t getCorrelator() const;

    /** \brief Returns the flag field. */
    virtual uint32_t getFlags() const;

    /** \brief Returns the flag specified. */
    virtual int getFlag( COMMON_HEADER_FLAG flagName ) const;

    /// Setters

    /** \brief Sets the version number. */
    virtual void setVersion( uint8_t version );

    /** \brief Sets the message type. */
    virtual void setMessageType( uint8_t messageType );

    /** \brief Sets the message length. */
    virtual void setMessageLength( uint16_t messageLength );

    /** \brief Sets the source sub-ID. */
    virtual void setSourceSubID( uint32_t sourceSubID );
```

```
    /** \brief Sets the source sub-ID. */
    virtual void setSourceSubID( uint32_t sourceSubID, int sTS );

    /** \brief Sets the destination sub-ID. */
    virtual void setDestinationSubID( uint32_t destinationSubID );

    /** \brief Sets the destination sub-ID. */
    virtual void setDestinationSubID( uint32_t destinationSubID, int dTS );

    /** \brief Sets the correlator. */
    virtual void setCorrelator( uint64_t correlator );

    /** \brief Sets the flag field. */
    virtual void setFlags( uint32_t flags );

    /** \brief Sets the flag specified. */
    virtual void setFlag( COMMON_HEADER_FLAG flagName, int value );

    /// Message content read from and write to Byte Array functions

    /** \brief Reads the contents of a byte array received from the network
     *        and sets the corresponding fields in an object. */
    virtual int readByteArray( const char* array, int size );

    /** \brief Generates a byte array corresponding the actual message to be
     *        transported to network. */
    virtual int writeByteArray( char** array ) const;

    /** \brief Writes object data to a byte array corresponding the actual
     *        message to be transported to network. */
    virtual int writeByteArray( char* array, int size ) const;

    /** \brief Checks the validity of a message. */
    virtual bool isValid();

    /** \brief Calculates the length of a message. */
    virtual uint16_t calculateLength();

    /** \brief Prints the message to the standard output. */
    virtual void print();

private:
    uint8_t m_version;              ///< Bits 0-3: reserved, bits 4-7: version
    uint8_t m_messageType;          ///< Message type
    uint16_t m_messageLength;       ///< Message length in DWORDS (4 byte inc.)
    uint32_t m_sourceSubID;         ///< Bits 0-29: Src. sub-ID, bits 30-31: sTS
    uint32_t m_destinationSubID;    ///< Bits 0-29: Dest.sub-ID, bits 30-31: dTS
    uint64_t m_correlator;
    uint32_t m_flags;               ///< Flags: Ack, Pri, EM, A, TP
};

/** \brief Makes a new ForCES Message */
ForCESMessage* makeMessage( const char* array, int size );

}

#endif /* FORCESMESSAGE_H_ */
```

tester.cpp

This is the test program the author has used to test the functionalities of the ForCES protocol implementation and to ensure the code works as it was specified. Due to the length of the source code in the test program, only one test function is shown here.

The following function is for the performance test of the ForCES test message 4. This function takes the test count number as a parameter which denotes how many times the test loop is executed. One test loop run constructs one Config message with total of 120 Parameter Field TLVs in a Config Data TLV which is stored inside a Config TLV. Again, the resulting Config TLV is stored inside an LFBselect TLV which is the top-level TLV of the Config message. The whole constructed test message length will be 1024 bytes.

```
void test2200( int testCount )
{
    int version = 1;
    int sts = 0;
    uint32_t sourceSubID = 1;
    int dts = 1;
    uint32_t destinationSubID = 1;
    uint64_t correlator = 0ull;
    int flag_Ack = 0;
    int flag_Pri = 1;
    int flag_EM = 1;
    int flag_A = 0;
    int flag_TP = 0;
    ConfigMessage* messages[ testCount ];

    cout << "Creating new Config message objects..." << endl;

    try
    {
        ParameterFieldTLV* parameterFieldTLV1 = NULL;
        ParameterFieldTLV* parameterFieldTLV2 = NULL;
        ConfigDataTLV* configDataTLV1 = NULL;
        ParameterFieldTLV* parameterFieldTLV3 = NULL;
        ParameterFieldTLV* parameterFieldTLV4 = NULL;
        ConfigDataTLV* configDataTLV2 = NULL;
        LFBselectTLV* topLevelTLV1 = NULL;
        int i, j;
        clock_t initTime;

        ConfigTLV* configTLV = NULL;
        initTime = clock();
        for( i = 0; i < testCount; i++ ) {
            configTLV = new ConfigTLV( 5 );
            for( j = 0; j < 60; j++ ) {
                parameterFieldTLV1 = new ParameterFieldTLV( 1 );
                parameterFieldTLV1->setParameterValue( 0x12345678 );

                parameterFieldTLV2 = new ParameterFieldTLV( 2 );
                parameterFieldTLV2->setParameterValue( 0xFEDCBA98 );

                configDataTLV1 = new ConfigDataTLV();
                configDataTLV1->setType( CONFIG_SET_PROP_TYPE );
                configDataTLV1->setIdentifier( 6 );

                configDataTLV1->addItem( parameterFieldTLV1 );
                configDataTLV1->addItem( parameterFieldTLV2 );

                configTLV->addItem( configDataTLV1 );
            }
            parameterFieldTLV3 = new ParameterFieldTLV( 3 );
            parameterFieldTLV4 = new ParameterFieldTLV( 4 );

            configDataTLV2 = new ConfigDataTLV( 2, 6 );

            configDataTLV2->addItem( parameterFieldTLV3 );
            configDataTLV2->addItem( parameterFieldTLV4 );

            configTLV->addItem( configDataTLV2 );
```

```
            topLevelTLV1 = new LFBselectTLV( CONFIG_MESSAGE_TYPE );

            topLevelTLV1->setLFBClassID( LFB_CLASS_ID_MANAGER );
            topLevelTLV1->setLFBInstanceID( LFB_INSTANCE_ID );
            topLevelTLV1->addItem( configTLV );

            messages[ i ] = new ConfigMessage( version, sts, sourceSubID, dts,
                    destinationSubID, correlator, flag_Ack, flag_Pri, flag_EM,
                    flag_A, flag_TP );

            messages[ i ]->addItem( topLevelTLV1 );
        }
        cout << (double)( ( clock() - initTime ) / (double)CLOCKS_PER_SEC ) << endl;
        cout << "Config messages created ok!" << endl;

        for( i = 0; i < testCount; i++ )
            delete messages[ i ];

    } catch (const std::logic_error &e)
    {
        cout << e.what();
    }
}
```

Appendix B

Test Runs

```
jgrondah@g2n-3:~/subversion/ce/ForCESMessage_tester/bin$ ./tester
ForCES Message library tester version 0.7a //JLG
Test programs:
1 - Association Setup Message
2 - Association Setup Response Message
3 - Association Teardown Message
4 - Config Message
5 - Config Response Message
6 - Query Message
7 - Query Response Message
8 - Event Notification Message
9 - Packet Redirect Message
10 - Heartbeat Message
11 - Association Setup Message, using makeMessage()
100 - Example tests

Select test program. Enter number and press enter: 9
Test program 9 selected!
Creating new packet redirect message object...
MetaDataILV created ok!
MetaDataTLV created ok!
MetaDataILV added to MetaDataTLV ok!
RedirectDataTLV created ok!
RedirectTLV created ok!
PacketRedirectMessage created ok!
message.print():
Message content is:
10 06 00 10 : 00 00 00 01 : 40 00 00 01 : 00 00 00 00   ........@.......
00 00 00 00 : 08 40 00 00 : 00 01 00 28 : 01 15 00 14   .....@.....(....
00 00 00 05 : 00 00 00 10 : 67 61 72 62 : 61 67 65 00   ........garbage.
01 16 00 10 : 01 02 03 04 : 05 06 07 08 : 09 00 00 00   ................
Message content is:
10 06 00 10 : 00 00 00 01 : 40 00 00 01 : 00 00 00 00   ........@.......
00 00 00 00 : 08 40 00 00 : 00 01 00 28 : 01 15 00 14   .....@.....(....
00 00 00 05 : 00 00 00 10 : 67 61 72 62 : 61 67 65 00   ........garbage.
01 16 00 10 : 01 02 03 04 : 05 06 07 08 : 09 00 00 00   ................
Printed. Program end. Deleting objects...
jgrondah@g2n-3:~/subversion/ce/ForCESMessage_tester/bin$

jgrondah@g2n-3:~/subversion/ce/ForCESMessage_tester/bin$ ./tester
ForCES Message library tester version 0.7a //JLG
Test programs:
1 - Association Setup Message
2 - Association Setup Response Message
3 - Association Teardown Message
4 - Config Message
5 - Config Response Message
6 - Query Message
```

```
7 - Query Response Message
8 - Event Notification Message
9 - Packet Redirect Message
10 - Heartbeat Message
11 - Association Setup Message, using makeMessage()
100 - Example tests

Select test program. Enter number and press enter: 100
Test program 100 selected!
Message content is:
10 01 00 0c : 00 00 00 01 : 40 00 00 01 : 00 00 00 00   ........@.......
00 00 00 00 : 08 40 00 00 : 10 00 00 18 : 00 00 00 01   .....@..........
00 00 00 01 : 00 0b 00 0c : e7 01 00 08 : 00 00 02 2b   ...............+
Message content is:
10 03 00 14 : 00 00 00 01 : 40 00 00 01 : 00 00 00 00   ........@.......
00 00 00 00 : 08 40 00 00 : 10 00 00 38 : 00 00 00 01   .....@.....8....
00 00 00 01 : 00 05 00 2c : 00 02 00 18 : 00 00 00 06   .......,........
00 01 00 08 : 12 34 56 78 : 00 02 00 08 : fe dc ba 98   .....4Vx........
00 02 00 10 : 00 00 00 06 : 00 03 00 04 : 00 04 00 04   ................
Message content is:
10 06 00 10 : 00 00 00 01 : 40 00 00 01 : 00 00 00 00   ........@.......
00 00 00 00 : 08 40 00 00 : 00 01 00 28 : 01 15 00 14   .....@.....(....
00 00 00 05 : 00 00 00 10 : 67 61 72 62 : 61 67 65 00   ........garbage.
01 16 00 10 : 01 02 03 04 : 05 06 07 08 : 09 00 00 00   ................
Message content is:
10 04 00 11 : 00 00 00 01 : 40 00 00 01 : 00 00 00 00   ........@.......
00 00 00 00 : 08 40 00 00 : 10 00 00 2c : 00 00 00 01   .....@.....,....
00 00 00 01 : 00 05 00 20 : 00 02 00 14 : 00 00 00 06   .......  ........
21 2c 37 42 : 4d 58 13 88 : cb fe 00 00 : 00 02 00 08   !,7BMX..........
00 00 00 07                                             ....
jgrondah@g2n-3:~/subversion/ce/ForCESMessage_tester/bin$
```

Appendix C

Numerical Values Used in ETNA

This appendix contains the numerical values for type names and IDs used in ETNA. The values that are defined in the IETF ForCES protocol specification are not listed here. Instead, they can be found in the IETF ForCES protocol specification draft [16].

Unique ID numbers for the Logical Function Block (LFB) Classes used in ETNA are listed in Table C.1. The LFB Class ID value is used in the LFB Class ID field of the LFBselect TLV which is included in ForCES Association Setup message, Config message, Config Response message, Query Message, Query Response message, and Event Notification message.

Table C.1: LFB Class IDs used in ETNA.

LFB Class ID	Value
Manager	0x0001
Forwarding Database	0x0002
OAM	0x0003
QoS	0x0004
Tunnel Control	0x0011
Topology Discovery	0x0012
Status Monitor	0x0013

Table C.2 shows the FE capability values used in ETNA. The FE capability value is used in the type field of the FE Capabilities TLV which is placed in an optional Report TLV in the ForCES Association Setup message.

Depending on the type of the operation in Config TLV in ForCES Config messages, Config TLV may have one or more Config Data TLVs. The type field of the Config Data TLV contains the type of the config data. The config data types and their corresponding values used in ETNA are listed in Table C.3.

Inside the Config Data TLV, there are one or more Parameter Field TLVs. The type field of the Parameter Field TLV contains the type of the parameter and the value of the parameter is stored in the TLV value field.

Table C.2: FE capabilities used in ETNA.

FE capability	Value
point-to-multipoint forwarding	0x0001
multiple forwarding tables	0x0002
port trunking	0x0004
OAM support	0x0010
link level OAM	0x0020
tunnel level OAM	0x0040
QoS support	0x0100
shaping	0x0200
scheduling	0x0400
port typing	0x1000
frame typing	0x2000

Table C.3: Config data types used in ETNA.

Config Data Type	Abbreviation	Value
LFB activation	LA	0x0001
LFB deactivation	LD	0x0002
LFB reset	LR	0x0003
LFB configuration	LC	0x0004
Device settings	DS	0x0011
Interface settings	IS	0x0012
Port settings	PS	0x0013
Port Type settings	PT	0x0014
Frame Format specification	FF	0x0021
Frame Mapping setup	FM	0x0022
Event Subscription	EV	0x0031
Forwarding Database setup	DBS	0x0041
Forwarding Database entry	DBE	0x0042
OAM setup	OAS	0x0051
OAM update	OAU	0x0052

The parameter types and their corresponding values are listed in Table C.5. The third column tells the length of the parameter value in octets. An asterisk means that the length of the parameter value can vary. If the length of the parameter value is not divisible by four, it must be padded with zero to the 32-bit boundary. The applicability column shows in which of the config data types the parameter can be applied. The abbreviations used in this column refer to the config data types shown in Table C.3. The description column has more detailed information on the usage of the parameter.

The ForCES Config Response message contains one or more Result TLVs. The result code is stored in the result code field of the Result TLV. In addition to the IETF specified result codes [16, p. 99], there are a few result codes specified by

ETNA for use with Config Response. These codes and their corresponding values are listed in Table C.4.

Table C.4: Config Response result codes used in ETNA.

Result Code	Value	Definition
E_C_IDENTIFIER_UNKNOWN	0x30	Identifier is unknown
E_C_PARAMETER_MISMATCH	0x31	Parameter not supported by config type
E_C_PARAMETER_NOT_SUPPORTED	0x32	Setting of parameter not supported by receiver

Event codes are used in the Event Notification TLV in the ForCES Event Notification message. The ETNA specified event codes and their corresponding values are listed in Table C.6. The data column in the table indicates which event types allow extra data. The extra data is stored in Event Data TLVs after the Event Code field in the Event Notification TLV.

Table C.5: Config parameter values used in ETNA.

Parameter	Value	Len	Applicability	Description
ID	0x0001	8	LC, DS, IS, PS, PR, FF, FM, DBS, DBE	Unique identifier (32 bit integer)
Address	0x0002	12	DS, IS, PS	NSAP/MAC address (6 octets + padding)
Name	0x0003	*	DS, IS, PS, PT, FF, DBS	User defined name (string)
Speed	0x0004	8	IS	Speed in bits per second (32 bit integer)
Duplexity	0x0005	8	IS	Interface duplexity: 0x01 = half duplex, 0x02 = full duplex (32 bit integer)
State	0x0006	8	IS, PS	Interface / port state: 0x00 = down, 0x01 = up (32 bit integer)
Type	0x0007	8	PS	Port Type identifier (32 bit integer)
Interfaces	0x0008	*	PS	List of interface IDs bound to port (array of 32 bit integers)
InFrames	0x0009	*	PS, PT	List of incoming frame format IDs allowed by port / port type (array of 32 bit integers)
OutFrames	0x000A	*	PS, PT	List of outgoing frame format IDs allowed by port / port type (array of 32 bit integers)
OpMode	0x000B	8	PS, PT	Default operating mode for port / port type: 0x00 = deny all traffic, 0x01 = allow InFrames, 0x02 = allow OutFrames, 0x03 = allow all known frames, 0x0F = allow all traffic (32 bit integer)
FrameStruct	0x000C	*	FF	Frame structure in byte encoded XML format (bXML)
FrameMappingIDs	0x000D	12	FM	Frame format IDs mapped together (2 32 bit integers)
FrameMappingDir	0x000E	8	FM	Direction of frame mapping: 0x01 = unidirectional, 0x02 = bidirectional (32 bit integer)
FrameMappingStruct	0x000F	*	FM	Byte encoded XML description of frame field mapping (bXML)
Subscription	0x0010	8	EV	Subscription status: 0x00 = not subscribed, 0x01 = subscribed (32 bit integer)
FDBfields	0x0011	*	DBS	Byte encoded XML describing the structure of forwarding database (bXML)
FDBtimeout	0x0012	8	DBS, DBE	Forwarding DB entry timeout in milliseconds (32 bit integer)
FDBkey	0x0013	*	DBE	Byte encoded XML describing the field matches for incoming traffic (bXML)
FDBmatch	0x0014	*	DBE	Byte encoded XML describing the set fields for outgoing traffic (bXML)
FDBout	0x0015	*	DBE	Frame mapping and outgoing port ID pairs for outgoing traffic (array of 2 32 bit integers)
FDBstate	0x0016	8	DBE	Forwarding entry state: 0x00 = deactive , 0x01 = active (32 bit integer)
OAMstate	0x0017	8	OAS	OAM instance state: 0x00 = deactive , 0x01 = active (32 bit integer)
OAMscope	0x0018	8	OAS	OAM level: 0x00 = link, 0x01 = tunnel (32 bit integer)
OAMrate	0x0019	8	OAS	OAM packet interval, matches values in CCM interval field (32 bit integer)
OAMout	0x001A	8	OAS	Outgoing interface (link) or FDB entry (tunnel) ID (32 bit integer)
OAMlinkdst	0x001B	12	OAS	NSAP/MAC address of the neighbor across this link (6 octets + padding)

Table C.6: Event codes used in ETNA.

Code	Value	Data	Description
LFB_ERROR	0x0001	Yes	LFB detected an unrecoverable internal error.
LFB_DOWN	0x0002	Yes	LFB going down because of something.
LFB_NEW	0x0003	Yes	New LFB started.
IF_DOWN	0x0011	No	Network interface went down (disconnected).
IF_UP	0x0012	No	Network interface came up (reconnected).
IF_STATUS	0x0013	Yes	Other change in interface status.
PORT_DOWN	0x0015	No	Port went down (disconnected).
PORT_UP	0x0016	No	Port came up (reconnected).
PORT_STATUS	0x0017	Yes	Other change in port status.
OAM_HB_FAILURE	0x0021	No	OAM heartbeats not received on tunnel.
OAM_DELAY	0x0022	Yes	OAM delay/jitter notification.
STATS_UPDATE	0x0031	Yes	Periodical update in subscribed statistics group.

Appendix D

Measurements

This appendix contains the original values from the measurements explained in this thesis. The first five columns are the measured time values. Then comes their average value and standard deviation. The next column tells how many messages was processed. Then the number of memory allocations and how many bytes of memory has been allocated totally from the heap.

Table D.1: Heartbeat message construction time measurements and related memory allocations with non-optimized code.

Val. #1 (s)	Val. #2 (s)	Val. #3 (s)	Val. #4 (s)	Val. #5 (s)	Ave. time (s)	St. dev. (s)	Number of messages	Number of mem. alloc.	Memory allocated (bytes)
0.82	0.81	0.82	0.82	0.81	0.816	0.005	100000	100000	3200000
1.64	1.63	1.57	1.68	1.72	1.648	0.056	200000	200000	6400000
2.40	2.40	2.38	2.55	2.40	2.426	0.070	300000	300000	9600000
3.21	3.15	3.18	3.10	3.25	3.178	0.057	400000	400000	12800000
3.99	3.99	4.05	3.98	3.93	3.988	0.043	500000	500000	16000000

Table D.2: Heartbeat message construction time measurements and related memory allocations with optimized code.

Val. #1 (s)	Val. #2 (s)	Val. #3 (s)	Val. #4 (s)	Val. #5 (s)	Ave. time (s)	St. dev. (s)	Number of messages	Number of mem. alloc.	Memory allocated (bytes)
0.55	0.54	0.57	0.55	0.56	0.554	0.011	100000	100000	3200000
1.07	1.08	1.12	1.07	1.09	1.086	0.021	200000	200000	6400000
1.59	1.56	1.59	1.59	1.56	1.578	0.016	300000	300000	9600000
2.24	2.05	2.09	2.09	2.09	2.112	0.074	400000	400000	12800000
2.61	2.59	2.66	2.57	2.57	2.600	0.037	500000	500000	16000000

Table D.3: Heartbeat message construction time measurements and related memory allocations normalized with non-optimized code.

Val. #1 (μs)	Val. #2 (μs)	Val. #3 (μs)	Val. #4 (μs)	Val. #5 (μs)	Ave. time (μs)	St. dev. (μs)	Number of messages	Number of mem. alloc.	Memory allocated (bytes)
8.2	8.1	8.2	8.2	8.1	8.16	0.055	1	1	32
16.4	16.3	15.7	16.8	17.2	16.48	0.563	2	2	64
24.0	24.0	23.8	25.5	24.0	24.26	0.699	3	3	96
32.1	31.5	31.8	31.0	32.5	31.78	0.572	4	4	128
39.9	39.9	40.5	39.8	39.3	39.88	0.427	5	5	160

Table D.4: Heartbeat message construction time measurements and related memory allocations normalized with optimized code.

Val. #1 (μs)	Val. #2 (μs)	Val. #3 (μs)	Val. #4 (μs)	Val. #5 (μs)	Ave. time (μs)	St. dev. (μs)	Number of messages	Number of mem. alloc.	Memory allocated (bytes)
5.5	5.4	5.7	5.5	5.6	5.54	0.114	1	1	32
10.7	10.8	11.2	10.7	10.9	10.86	0.207	2	2	64
15.9	15.6	15.9	15.9	15.6	15.78	0.164	3	3	96
22.4	20.5	20.9	20.9	20.9	21.12	0.736	4	4	128
26.1	25.9	26.6	25.7	25.7	26.00	0.374	5	5	160

Table D.5: A short Config message with a few TLVs construction time measurements and related memory allocations with non-optimized code.

Val. #1 (s)	Val. #2 (s)	Val. #3 (s)	Val. #4 (s)	Val. #5 (s)	Ave. time (s)	St. dev. (s)	Number of messages	Number of mem. alloc.	Memory allocated (bytes)
0.31	0.30	0.30	0.31	0.30	0.304	0.005	1000	19000	308000
0.53	0.52	0.51	0.53	0.52	0.522	0.008	2000	38000	616000
0.72	0.73	0.72	0.72	0.72	0.722	0.004	3000	57000	924000
0.94	0.94	0.94	0.97	0.94	0.946	0.013	4000	76000	1232000
1.16	1.17	1.18	1.15	1.17	1.166	0.011	5000	95000	1540000

Table D.6: A short Config message with a few TLVs construction time measurements and related memory allocations with optimized code.

Val. #1 (s)	Val. #2 (s)	Val. #3 (s)	Val. #4 (s)	Val. #5 (s)	Ave. time (s)	St. dev. (s)	Number of messages	Number of mem. alloc.	Memory allocated (bytes)
0.13	0.13	0.13	0.14	0.12	0.130	0.007	1000	19000	308000
0.20	0.21	0.22	0.22	0.21	0.212	0.009	2000	38000	616000
0.30	0.28	0.29	0.28	0.27	0.284	0.011	3000	57000	924000
0.36	0.38	0.37	0.36	0.35	0.364	0.011	4000	76000	1232000
0.44	0.44	0.46	0.43	0.44	0.442	0.011	5000	95000	1540000

Table D.7: A short Config message with a few TLVs construction time measurements and related memory allocations normalized with non-optimized code.

Val. #1 (μs)	Val. #2 (μs)	Val. #3 (μs)	Val. #4 (μs)	Val. #5 (μs)	Ave. time (μs)	St. dev. (μs)	Number of messages	Number of mem. alloc.	Memory allocated (bytes)
310	300	300	310	300	304	5.48	1	19	308
530	520	510	530	520	522	8.37	2	38	616
720	730	720	720	720	722	4.47	3	57	924
940	940	940	970	940	946	13.42	4	76	1232
1160	1170	1180	1150	1170	1166	11.40	5	95	1540

Table D.8: A short Config message with a few TLVs construction time measurements and related memory allocations normalized with optimized code.

Val. #1 (μs)	Val. #2 (μs)	Val. #3 (μs)	Val. #4 (μs)	Val. #5 (μs)	Ave. time (μs)	St. dev. (μs)	Number of messages	Number of mem. alloc.	Memory allocated (bytes)
130	130	130	140	120	130	7.07	1	19	308
200	210	222	220	210	212	8.88	2	38	616
300	280	290	280	270	284	11.40	3	57	924
360	380	370	360	350	364	11.40	4	76	1232
440	440	460	430	440	442	10.96	5	95	1540

Table D.9: A long Config message with a few TLVs construction time measurements and related memory allocations with non-optimized code.

Val. #1 (s)	Val. #2 (s)	Val. #3 (s)	Val. #4 (s)	Val. #5 (s)	Ave. time (s)	St. dev. (s)	Number of messages	Number of mem. alloc.	Memory allocated (bytes)
0.36	0.33	0.35	0.36	0.36	0.352	0.013	1000	19000	1252000
0.63	0.61	0.62	0.63	0.63	0.624	0.009	2000	38000	2504000
0.90	0.90	0.88	0.90	0.90	0.896	0.009	3000	57000	3756000
1.15	1.18	1.16	1.16	1.16	1.162	0.011	4000	76000	5008000
1.43	1.42	1.44	1.46	1.44	1.438	0.015	5000	95000	6260000

Table D.10: A long Config message with a few TLVs construction time measurements and related memory allocations with optimized code.

Val. #1 (s)	Val. #2 (s)	Val. #3 (s)	Val. #4 (s)	Val. #5 (s)	Ave. time (s)	St. dev. (s)	Number of messages	Number of mem. alloc.	Memory allocated (bytes)
0.15	0.14	0.17	0.15	0.17	0.156	0.013	1000	19000	1252000
0.27	0.27	0.27	0.25	0.28	0.268	0.011	2000	38000	2504000
0.37	0.39	0.37	0.36	0.35	0.368	0.015	3000	57000	3756000
0.49	0.50	0.49	0.47	0.49	0.488	0.011	4000	76000	5008000
0.60	0.60	0.60	0.58	0.60	0.596	0.009	5000	95000	6260000

Table D.11: A long Config message with a few TLVs construction time measurements and related memory allocations normalized with non-optimized code.

Val. #1 (μs)	Val. #2 (μs)	Val. #3 (μs)	Val. #4 (μs)	Val. #5 (μs)	Ave. time (μs)	St. dev. (μs)	Number of messages	Number of mem. alloc.	Memory allocated (bytes)
360	330	350	360	360	352	13.03	1	19	1252
630	610	620	630	630	624	8.94	2	38	2504
900	900	880	900	900	896	8.94	3	57	3756
1150	1180	1160	1160	1160	1162	10.95	4	76	5008
1430	1420	1440	1460	1440	1438	14.83	5	95	6260

Table D.12: A long Config message with a few TLVs construction time measurements and related memory allocations normalized with optimized code.

Val. #1 (μs)	Val. #2 (μs)	Val. #3 (μs)	Val. #4 (μs)	Val. #5 (μs)	Ave. time (μs)	St. dev. (μs)	Number of messages	Number of mem. alloc.	Memory allocated (bytes)
150	140	170	150	170	156	13.42	1	19	1252
270	270	270	250	280	268	10.95	2	38	2504
370	390	370	360	350	368	14.83	3	57	3756
490	500	490	470	490	488	10.95	4	76	5008
600	600	600	580	600	596	8.94	5	95	6260

Table D.13: A long Config message with lots of TLVs construction time measurements and related memory allocations with non-optimized code.

Val. #1 (s)	Val. #2 (s)	Val. #3 (s)	Val. #4 (s)	Val. #5 (s)	Ave. time (s)	St. dev. (s)	Number of messages	Number of mem. alloc.	Memory allocated (bytes)
0.51	0.50	0.50	0.49	0.51	0.502	0.008	100	43700	623200
0.92	0.92	0.92	0.97	0.93	0.932	0.022	200	87400	1246400
1.36	1.35	1.35	1.35	1.34	1.350	0.007	300	131100	1869600
1.77	1.78	1.76	1.79	1.78	1.776	0.011	400	174800	2492800
2.25	2.19	2.19	2.18	2.19	2.200	0.028	500	218500	3116000

Table D.14: A long Config message with lots of TLVs construction time measurements and related memory allocations with optimized code.

Val. #1 (s)	Val. #2 (s)	Val. #3 (s)	Val. #4 (s)	Val. #5 (s)	Ave. time (s)	St. dev. (s)	Number of messages	Number of mem. alloc.	Memory allocated (bytes)
0.24	0.22	0.21	0.21	0.22	0.220	0.012	100	43700	623200
0.41	0.39	0.40	0.39	0.40	0.398	0.008	200	87400	1246400
0.55	0.56	0.55	0.56	0.56	0.556	0.005	300	131100	1869600
0.74	0.73	0.75	0.73	0.73	0.736	0.009	400	174800	2492800
0.91	0.90	0.90	0.90	0.90	0.902	0.004	500	218500	3116000

Table D.15: A long Config message with lots of TLVs construction time measurements and related memory allocations normalized with non-optimized code.

Val. #1 (μs)	Val. #2 (μs)	Val. #3 (μs)	Val. #4 (μs)	Val. #5 (μs)	Ave. time (μs)	St. dev. (μs)	Number of messages	Number of mem. alloc.	Memory allocated (bytes)
5100	5000	5000	4900	5100	5020	83.67	1	437	6232
9200	9200	9200	9700	9300	9320	216.80	2	874	12464
13600	13500	13500	13500	13400	13500	70.71	3	1311	18696
17700	17800	17600	17900	17800	17760	114.02	4	1748	24928
22500	21900	21900	21800	21900	22000	282.84	5	2185	31160

Table D.16: A long Config message with lots of TLVs construction time measurements and related memory allocations normalized with optimized code.

Val. #1 (μs)	Val. #2 (μs)	Val. #3 (μs)	Val. #4 (μs)	Val. #5 (μs)	Ave. time (μs)	St. dev. (μs)	Number of messages	Number of mem. alloc.	Memory allocated (bytes)
2400	2200	2100	2100	2200	2200	122.47	1	437	6232
4100	3900	4000	3900	4000	3980	83.67	2	874	12464
5500	5600	5500	5600	5600	5560	54.77	3	1311	18696
7400	7300	7500	7300	7300	7360	89.44	4	1748	24928
9100	9000	9000	9000	9000	9020	44.72	5	2185	31160