EVH2 protocol

Performance analysis and Wireshark dissector development

STEFAN ÅHMAN
and
MARCUS WALLSTERSSON

KTH Information and Communication Technology

Degree project in
Communication Systems
First level, 15.0 HEC
Stockholm, Sweden

EVH2 protocol

Performance analysis and Wireshark dissector development

Stefan Åhman
Marcus Wallstersson

Bachelor of Science Thesis

Communication Systems
School of Information and Communication Technology
KTH Royal Institute of Technology
Stockholm, Sweden

30 June 2012

Examiner: Professor Gerald Q. Maguire Jr.

Abstract

EVH2 is a proprietary application layer protocol developed by Aptilo Networks and used in their software product. Currently the only way to inspect EVH2 traffic is by using their own application. This application inspects no traffic other than EVH2. Since Aptilo continuously develops this protocol it is important to see how changes in the protocol affect its performance. This thesis examines possible ways to facilitate the use and development of the protocol.

To analyse EVH2 network traffic along with traffic from other protocols another approach is needed. *Wireshark* is an application capable of inspecting traffic of multiple protocols simultaneously and uses dissectors to decode each packet. This thesis describes the development and evaluation of a Wireshark plugin dissector for inspection of EVH2 traffic.

Performance tests of EVH2 will provide feedback about protocol changes. This thesis creates a platform for performance evaluation by introducing a test suite for performance testing. A performance evaluation of EVH2 was conducted using the developed test suite.

Keywords: Wireshark, protocol, dissector, performance, analysis, EVH2, Aptilo Networks, test suite

Sammanfattning

EVH2 är ett proprietärt applikationslagerprotokoll utvecklat av Aptilo Networks som används i deras mjukvaruprodukt. För närvarande kan EVH2-trafik endast inspekteras med deras egenutvecklade applikation. Denna applikation har inte något stöd för att inspektera annan trafik än EVH2. Eftersom Aptilo kontinuerligt utvecklar detta protokollet är det viktigt att kunna se hur förändringar i protokollet påverkar dess prestanda. Detta examenarbete undersöker möjliga sätt att underlätta användningen och utvecklingen av protokollet.

För att kunna inspektera EVH2-trafik tillsammans med trafik från andra protokoll behövs en annan lösning än den nuvarande. Wireshark är en applikation som har stöd för att inspektera flera protokoll samtidigt där protokollpaketen avkodas med *dissectors* (dissektorer översatt till svenska). I detta examensarbete beskrivs och utvärderas utvecklingen av ett *Wireshark dissector plugin* för inspektion av EVH2-trafik.

Genom att prestandatesta EVH2 kan prestandaskillnader påvisas vid förändringar i protokollet. Detta examensarbete tar fram en plattform för prestandautvärdering genom att introducera en testsvit för prestandatestning. Den utvecklade testsviten användes för att utföra prestandtestning av EVH2.

Acknowledgements

We wish to express our sincere gratitude to our supervisors at Aptilo Networks, Pontus Söderström and Peter Eriksson, who gave us the chance to do this thesis. We also wish to express our appreciation of our examiner Professor Gerald Q. Maguire Jr. for his great support and feedback during this thesis. Many thanks to staff at the development department of Aptilo, especially Martin Hedenfalk for his help and advice during the development process.

Finally, we would like to thank everyone at Aptilo Netwoks for a great time at the office.

Contents

List of Figures

List of Tables

List of listings

List of Acronyms and Abbreviations

This thesis requires the reader to be familiar with terms and concepts of computer networking and computer science. In this section acronyms and abbreviations are described shortly before using them in the thesis.

BGP	Border Gateway Protocol
C	C programming language
CPU	Central Process Unit
CSV	Comma-separated value
DHCP	Dynamic Host Configuration Protocol
DNS	Domain Name System
EVH2	Event Handler Protocol
FTP	File Transfer Protocol
GUI	Graphical User Interface
HTTP	The Hypertext Transfer Protocol
ICMP	Internet Control Message Protocol
IEEE 802.3	Ethernet
IEEE 802.11	Wireless LAN
IANA	Internet Assigned Numbers Authority
IP	Internet Protocol
IPSec	Internet Protocol Security
IPv4	Internet Protocol version 4

IPv6	Internet Protocol version 6
LAN	Local Area Network
NFS	Network File System
OSPF	Open Shortest Path First
PCAP	Library for packet capture
PPP	Point-to-Point Protocol
PPTP	Point-to-Point Tunneling Protocol
QoS	Quality of Service
RIP	Routing Information Protocol
SCTP	Stream Control Transmission Protocol
SLIP	Serial Line Internet Protocol
SMTP	Simple Mail Transfer Protocol
STDIN	Standard input
STDOUT	Standard output
SVN	Apache Subversion
TCP	Transmission Control Protocol
TSN.1	Transfer Syntax Notation One
UDP	User Datagram Protocol
UNIX	Uniplexed Information and Computing System (UNICS)
XML	Extensible Markup Language

Chapter 1

Introduction

This chapter contains some background information about why this thesis was conducted, then describes the thesis project's goals, purpose, and limitations.

1.1 Background

EVH2 is an application layer protocol developed by the company Aptilo Networks and used for communication between their software components, but also it is used for system debugging. The protocol utilizes a binary encode, thus making analyzing protocol messages and dealing with implementation errors difficult and time-consuming for developers without proper tools.

The use of continuous testing and performance evaluations benefit the development process of a software product. A test suite for performance testing will give the developer an overview of how the product is affected by changes to the code. When incrementally improving code, testing is often only done by looking at a specific part of the application regardless of other parts that might be negatively affected by these changes. This kind of problem is often not detected due to lack of thorough regression testing. By continuously testing the code these problems can be quickly detected and fixed, saving the developer many hours. Today only a few performance tests for EVH2 have been implemented. Better testing will be implemented as part of a test suite to be used as a tool for evaluating the EVH2's performance.

There are no tools for inspecting EVH2 traffic together with other protocols. This complicates the improvement of the protocol as well as complicating troubleshooting during installation of new components. To facilitate development and improvement a EVH2 dissector for Wireshark is needed.

1.2 Goals

The goals of this thesis project are to implement two tools: A EVH2 Wireshark dissector and a EVH2 test suite. These tools will be used for a performance evaluation of EVH2 and to facilitate the developer's daily work with the protocol.

The Wireshark dissector will be able to dissect EVH2 and present the contents of a message, translated with the EVH2 dictionary. The dissector will provide developers with the ability to inspect and evaluate EVH2, along with other protocols in the same application.

The test suite will be a tool for developers and can be used to perform regression testing. The tool will test and measure relevant performance areas such as:

- Runtime of functions used to handle EVH2 messages,
- CPU-load when processing EVH2 messages at sender or receiver, and
- The round trip time of a EVH2 message

1.3 Thesis purpose

Implementing tools for performance analysis of EVH2 will facilitate development of the protocol by providing developers with performance feedback following protocol changes. A Wireshark dissector for EVH2 will save developers time as they will be able to inspect multiple protocols including EVH2 using one application.

1.4 Limitations

This thesis will address testing and performance evaluation of EVH2. Potential performance improvements to EVH2 detected during the evaluation will not be implemented (but might be noted in internal memos to the developers).

Only dissectors on the application layer will be addressed. The implemented dissector will not be contributed to the Wireshark source code, but used only by Aptilo as EVH2 is a protocol proprietary to Aptilo.

1.5 Thesis outline

The thesis is structured with the following chapters:

Chapter 1
Introduction to the problem and its context as well as goals, purpose, and limitations.

Chapter 2
Protocols in general and application layer protocols will be described. A brief explanation of the EVH2 protocol will be given covering the protocol's structure and how the protocol can be used. Previous evaluations will be mentioned, as well as the existing tools for testing and analysing EVH2.

Chapter 3
Information about Wireshark, dissectors, and implementation of dissectors.

Chapter 4
Information about applications for evaluation of test data, previous evaluations of EVH2, and relevant areas to be tested.

Chapter 5
Describes the methods, tools, and environments used in this thesis project.

Chapter 6
Explains the implementation of the Wireshark dissector and test suite.

Chapter 7
Presents results of the dissector, test suite, and test suite measurements.

Chapter 8
The analysis of the Wireshark dissector, test suite, and performance evaluation of EVH2 are discussed.

Chapter 9
Conclusions of this thesis. Presents obtained goals and suggests future work that can be done.

Chapter 2

EVH2

The reader of this thesis is expected to have basic knowledge of computer networking. This chapter will briefly review some of the relevant protocols. The EVH2 protocol will be explained in detail including its structure and usage. Different tools for testing and analysis will be briefly described.

2.1 Protocols in general

When you communicate with another person, the conversation usually is initiated with a greeting followed by a question or a statement. One requirement is that the other person speak the same language, otherwise your question can not be answered. This is similar to the case of network protocols as the communicating hosts must speak a protocol at the same time in order to exchange data. The protocols operate on different layers of the protocol stack, where each layer has a special role in the communication between endpoints. A parable of a network protocol and a human conversation are shown in Figure 2.1.

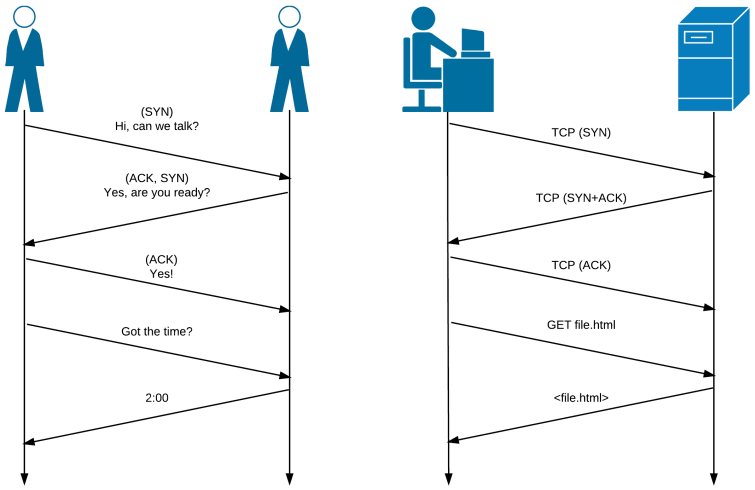

Figure 2.1: A human protocol and a computer network protocol

2.1.1 Protocol stack

The TCP/IP protocol stack is separated into five layers (from top to bottom): application layer, transport layer, network layer, data link layer, and physical layer. A brief summary of where different protocols belong (*protocol stack*) is found below:

Application layer
EVH2, DNS, FTP, HTTP, NFS, DHCP, SMTP, RIP, BGP

Transport layer
TCP, UDP, SCTP

Network layer
IP, IPsec, ICMP, OSPF

Data link layer
PPP, SLIP, PPTP

Physical layer
IEEE 802.3, IEEE 802.11

2.1.2 Application layer protocols

The application layer protocol is the richest layer due to the amount of different networking applications. Every application has a unique way to communicate between endpoints. The application data is passed to the layer beneath it.

An application layer protocol is used for example when browsing the web. HTTP is used to retrieve a web page to be rendered for viewing in the client's browser (see Figure 2.2). The conversation begins with a 3-way TCP handshake. A GET request is sent specifying the requested HTML that is used to realize a web page. The server responds by sending a response message containing the requested data. This procedure is repeated to get the resources needed to render the page in the client's web browser.

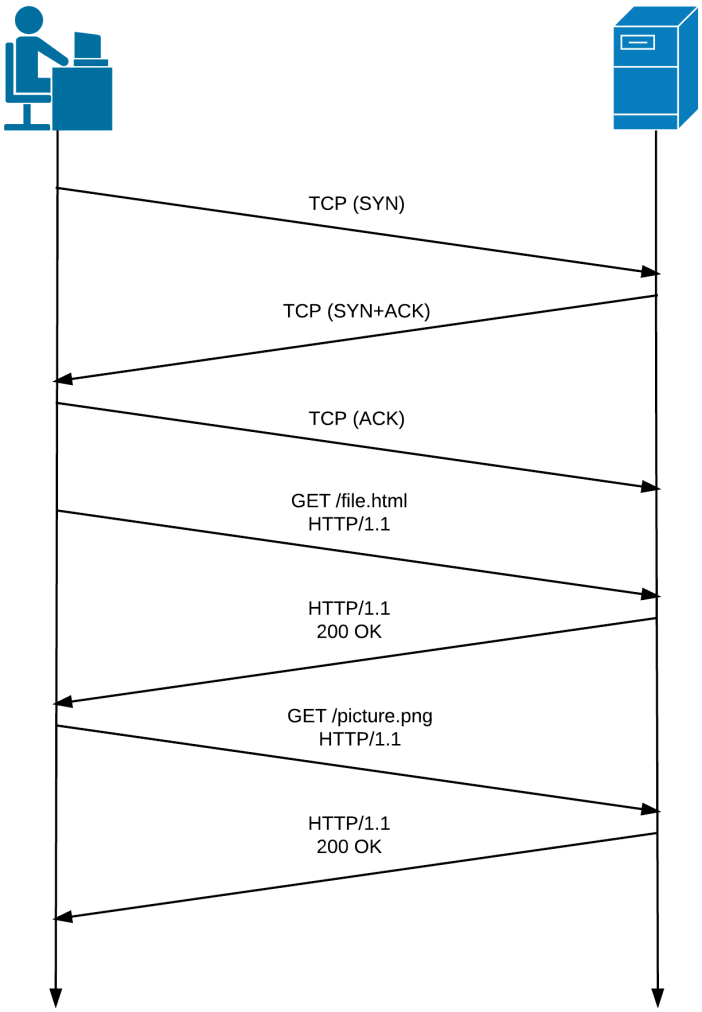

Figure 2.2: HTTP message conversation

2.2 What is EVH2?

EVH2 is a request-response protocol implemented in the application layer of the protocol stack. The protocol is used in Aptilo's software to send messages among different components. These messages are exchanged via the localhost and remote hosts. EVH2 is a binary protocol where requests and responses have the same format. This section will describe the packet structure and the implementation of the EVH2 protocol.[2]

2.3 Message format

Each protocol has a predefined structure, in the case of EVH2 it is structured as a header with optional attributes. This section explains the fields of the header and attributes.

2.3.1 EVH2 Header

The current message format of EVH2 version 3 is shown in Figure 2.3.

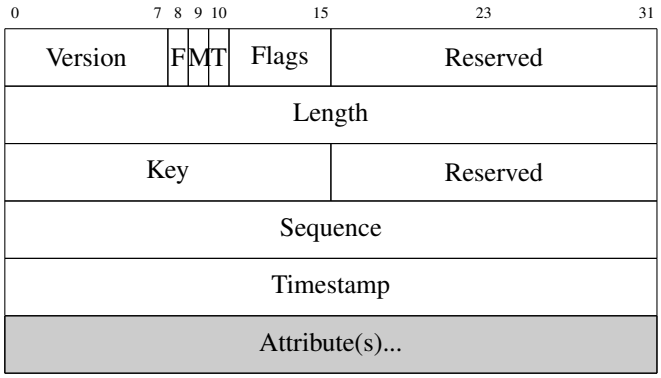

Figure 2.3: EVH2 protocol header. Attribute(s) are optional

Version
The version of the protocol (currently 3). Unsigned 8-bit integer.

F
Flag specifying that the event has been forwarded from another machine.

M
Flag specifying that multiple responses are being sent. The last response has this flag cleared.

T
Flag specifying that the event should be traced. The meaning of "traced" is not part of the EVH2 protocol specification.

Flags
5-bit flag field (currently unused). Unused bits must be set to zero.

Length
Total length of message including header and all attributes. Unsigned 32-bit integer, network byte order.

Key
Unique identifier specifying what kind of message this is. Unsigned 16-bit integer, network byte order.

Sequence
Number used to correlate requests with responses. Unsigned 32-bit integer, network byte order.

Timestamp
Set by the sending part when message is sent. Unsigned 32-bit integer, network byte order.

Reserved
Reserved for future use. Must be set to zero when sending and ignored on receive.

Attribute
Field containing one or more attributes, described in section 2.3.2

2.3.2 EVH2 Attribute

Each message can contain zero or more attributes, all with the same format (illustrated in Figure 2.4). Attributes can be *grouped* since the value of these attributes can contain multiple attributes in a tree structure. Attributes only containing values are referred to as *ungrouped*.

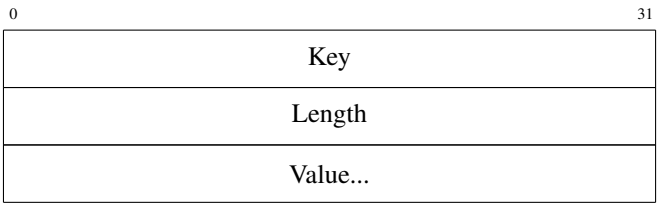

Figure 2.4: EVH2 protocol attribute header

Key
Unique identifier specifying the attribute. Unsigned 32-bit integer, network byte order.

Length
Total length of the attribute including key, length and value. Unsigned 32-bit integer, network byte order.

Value
Value of attribute. Array of length - 8 octets.

The interpretation of the Value depends on the Key, since the protocol itself does not specify the meaning of any values. Possible types of attributes are:

Octets
Array of bytes.

String
C style strings.

IPv4 / IPv6 address

Boolean
Single byte, non-zero value representing TRUE, while zero represents FALSE

Integer
8, 16, 32, and 64-bit signed integers

Time
Unix time stamp

Radius
RADIUS attribute

Diameter
Diameter attribute

2.4 Architecture

EVH2 messages can be sent in two ways, in a client-server fashion or by broadcast using the EVH2 daemon (evh2d).

2.4.1 Server/Client model

When sending messages using a client-server architecture, a client makes a request, for example for some data. The server will respond to the request with the data within the attributes (see Figure 2.5).

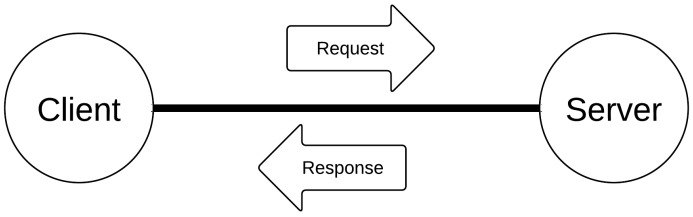

Figure 2.5: EVH2: Server/Client model

2.4.2 Daemon model

The evh2d daemon broadcasts messages produced by a certain server. Clients can connect to the daemon and subscribe to different types of messages. When a server sends a message to evh2d, the daemon will broadcast this message to all its subscribers (see Figure 2.6).

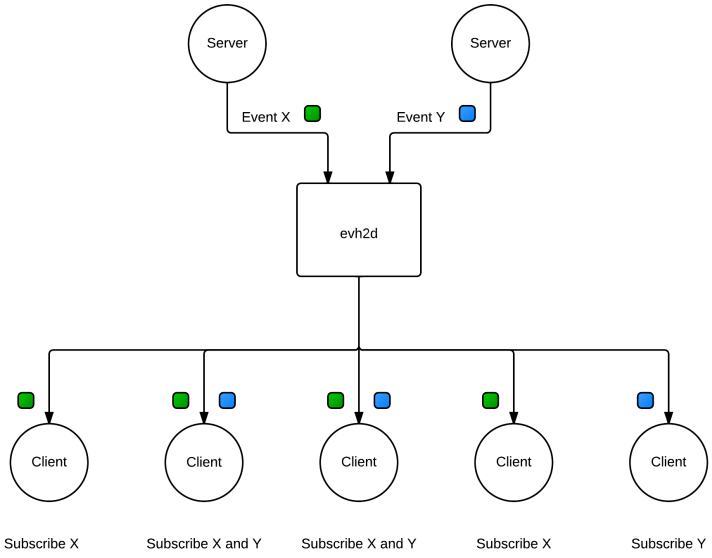

Figure 2.6: EVH2d: Daemon model

2.5 Evh2tools

Today there are four application forming a toolkit called *evh2tools*. These four applications are used to test EVH2. The protocol and all the EVH2 applications were developed by Aptilo Networks. The toolkit can send, receive, listen to message traffic, and perform very limited performance testing. All applications are written as command line applications in C and use an EVH2 library to create, send, receive, and manipulate EVH2 messages. These applications are described in the following paragraphs.

Evh2receive connects to a server specified by hostname and port number. All the incoming messages on this connection will be received and displayed. There are no performance tests made when receiving messages.

Evh2send is used to send events directly to a client running evh2receive by specifying the destination hostname and port number. If no host is specified, then events will be sent to evh2d on port 9090. When sending a message, a timer will measure the time it takes to send the message and print it on the console. Note that this time only includes the actual message sending and the print out of the sent message, the time the application waits for the ACK is not included. It is possible to provide the application with predefined messages on STDIN. These messages are separated by newline and will be sent sequentially.

Evh2listen is used to subscribe and listen for specific messages sent by evh2d. The application will by default listen on TCP port 9090. It is possible to listen for a specified number of messages by adding an option flag at runtime. When specified number of messages have been received, the application will terminate and print the measured receive time.

Evh2sniff is a packet capture application developed by Aptilo and used internally within the company. Like most other analysis tools, this application uses pcap to implement features such as filters and selecting interfaces. A negative aspect of this tool is its incompatibility with other protocols, hence the developers are limited to capturing only EVH2 traffic.

These tools makes use of a dictionary application that can translate numerical values to explanatory strings and vice versa. This makes it easier for the user to send and receive messages in textual form. All messages received by *evh2listen* and *evh2sniff* can be recorded and written to STDOUT in the format used for input to *evh2send*.

2.6 EVH2 dictionary

The EVH2 dictionary is a text file and has to be converted before it can be used with an application. The translation data from the text file is parsed and header files containing the translations are created. To use the dictionary the header file *evh2parse* must be included. This header file defines functions for dictionary lookups. The function (*dictionary_init*) defined in *evh2parse* will instantiate the dictionary by copying the translations from the header files into several red-black trees. A red-black tree is a self balanced binary tree. Once it has been instantiated the dictionary can be used to lookup translations. Lookups can be done to either concert a key to a string or vice versa. The dictionary can also lookup the data type of an attribute. To use the dictionary functionality in an application some proprietary packages developed by Aptilo need to be installed. The dictionary depends on these packages since it uses functionality provided by them, therefore it can only be used at Aptilo.

Chapter 3

Wireshark

This chapter will describe Wireshark in general, dissectors, and possible ways to implement dissectors.

3.1 What is Wireshark?

Wireshark is the world's foremost network protocol analyser[3]. The application is open source and is widely used for network analysis, protocol development, protocol analysis, and education. To capture packets, Wireshark uses the libpcap library developed by the Tcpdump team for UNIX [4]. The Windows version of the packet capturing library is called WinPcap and was developed by Riverbed Technology[5]. Wireshark can be downloaded fromthe Wireshark web page as an application[6] or as source code from the SVN-repository[7].

Wireshark currently has support for more than 850 protocols and the number is growing with each update. These protocols range from common protocols such as IP and DHCP to more advanced protocols such as like AppleTalk and BitTorrent. As Chris Sanders says in his book Practical Packet Analysis, "In the unlikely case that Wireshark doesn't support a protocol you need, you can code that support yourself and submit your code to the Wireshark developers for inclusion in the application"[8].

3.2 Usage of Wireshark

This subsection explains how Wireshark is used.

3.2.1 Capturing packets

Since Wireshark has a user-friendly graphical user interface (GUI) it is really simple to capture packets and then analyse them. The application is capable of capturing packets by monitoring different interfaces or reading pcap-files containing pre-captured data (see Figure 3.1). Unlike evh2sniff, with Wireshark it is possible to get a clear overview of the captured packets and to generate graphs, as well as flowcharts of traffic. Wireshark's main GUI consists of three different panes: Packet List, Packet Details, and Packet Bytes. The network traffic will be shown in the Packet List pane and visualized with different colors, which gives the user a good overview as shown in Figure 3.2. The packet list by default only shows basic information about each packet, such as: source, destination, type of protocol, and a timestamp. It is possible to add custom fields or remove current fields. The Packet Details view is shown by selecting a row in the packet list (i.e., by selecting a packet for which you want to see the details).

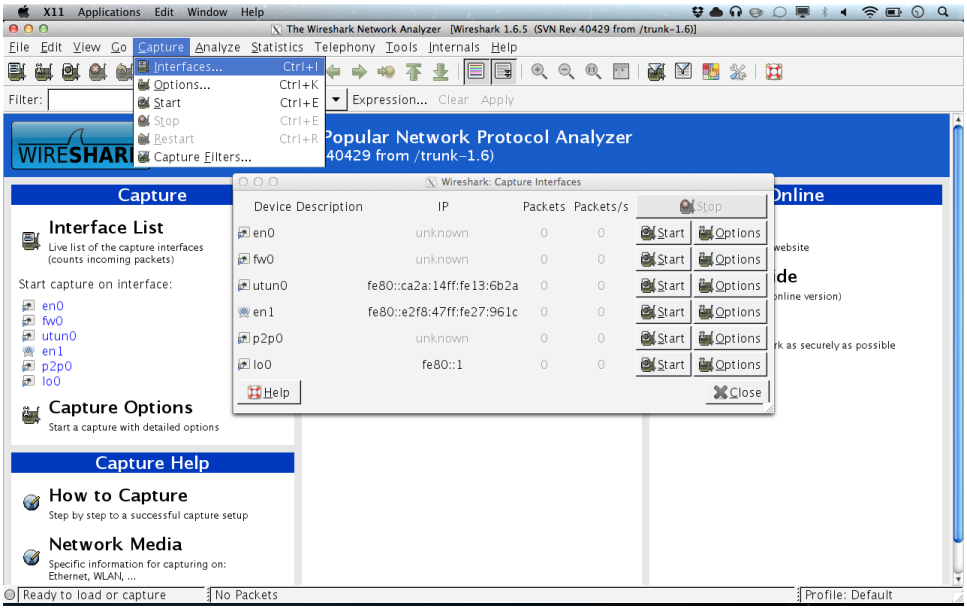

Figure 3.1: Wireshark: Capture interface

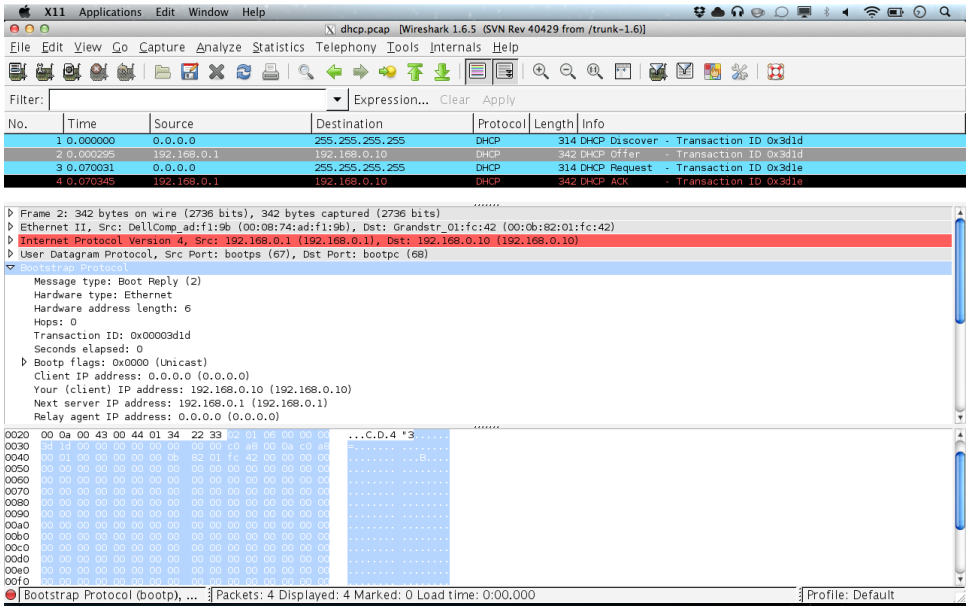

Figure 3.2: Wireshark: Inspection interface

3.2.2 Analysing packets

Dissectors will decode the corresponding protocol and its fields of the selected packet and present it in the Packet Details pane (see Figure 3.2). The protocols and fields are displayed in a tree view and can be expanded and collapsed. Without the corresponding dissector the protocol's content would simply be presented as a string of bytes. The raw data is always visible in the Packet Bytes pane and Wireshark is configurable to show the data in either hexadecimal or bits.

3.3 Dissectors

To use Wireshark for analysis of a certain protocol the Wireshark application must know how to interpret the protocol. This interpretation is implemented in C as a *dissector* and each protocol has its own dissector. The dissector translates the raw data from a packet, then presents the packet's content in the Packet Details pane.

A packet of a certain layer in the *protocol stack* is encapsulated in packets of each underlying layer. Due to this encapsulation multiple dissectors needs to cooperate when dissecting a captured packet. An example of an encapsulated EVH2 message is shown in Figure 3.3.

The dissection steps are:

- Ethernet dissector is used to dissect the Ethernet header (step 1),
- The payload of Ethernet is dissected by the IPv6 dissector (step 2),
- The payload of IPv6 is dissected by the TCP dissector (step 3), and
- The payload of TCP is a EVH2 packet and should be dissected by the EVH2 dissector (to be implemented) (step 4).

If no dissector for EVH2 is available in Wireshark when inspecting a EVH2 capture, the EVH2 packet will not be recognized and therefore not dissected. The packet will be dissected with EVH2 as an unknown payload during the dissection of the transport protocol. The payload will only be shown as bytes (see Figure 3.4).

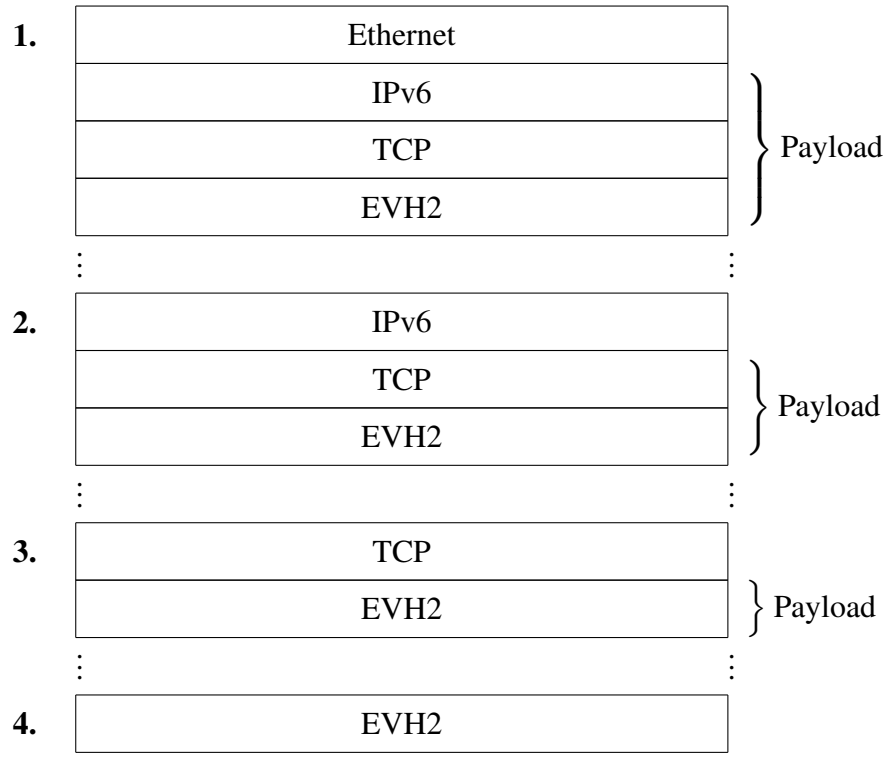

Figure 3.3: Encapsulation of a EVH2 packet

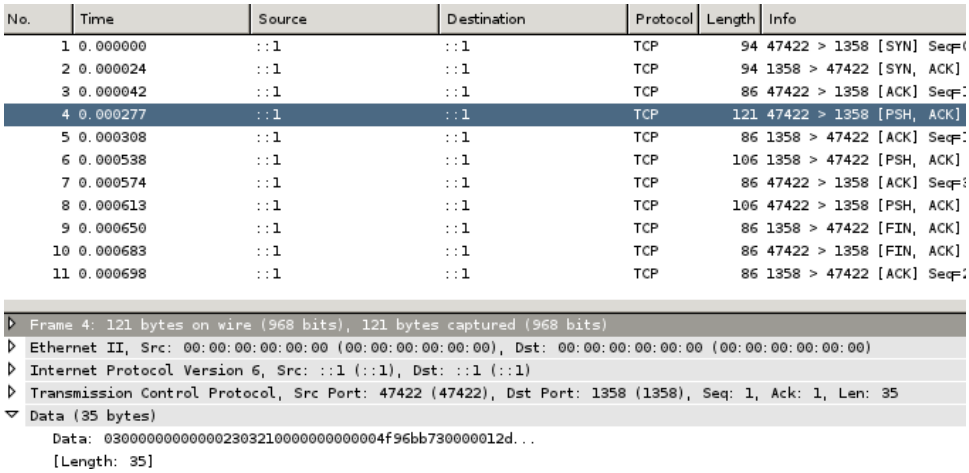

Figure 3.4: EVH2 message in Wireshark without a EVH2 dissector

3.3.1 Protocol dissection

The data (bytes) of the packet to be dissected are passed to the dissector in the form of a buffer. This buffer contains the complete packet and all the protocol fields will be represented by the bytes in the buffer. The dissection function will retrieve fields in the form of bytes from the buffer and add them to a protocol tree. The protocol tree is a tree structure containing the dissected data to be presented in the Packet Details pane. Wireshark has built in functions to retrieve a field and add it to the tree as an item (`proto_tree_add_item`) or subtree (`proto_tree_add_subtree`) (see Wireshark documentation[9], section 1.6.2). Example of items and subtrees are shown in Figure 3.5.

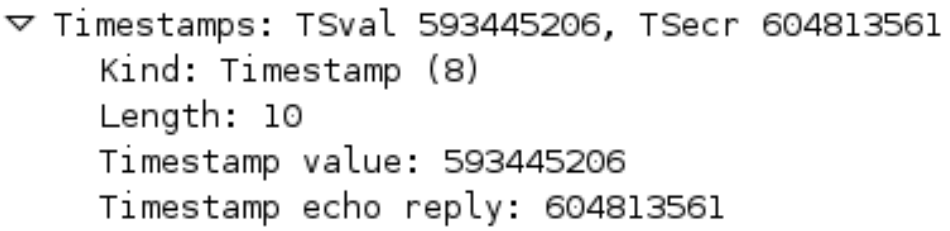

Figure 3.5: Example of a subtree containing four items

When a dissector is done dissecting all its fields, it will either stop or call another dissector. The call to another dissector is needed to dissect the payload as was described in Figure 3.3.

3.3.2 Requirements

Any implementation of a dissector requires a minimum of three functions to work with the Wireshark application. These functions are described briefly below. `PROTOABBREV` symbolizes the abbreviation of the protocol to be dissected. During application startup the dissector has to be registered by calling `proto_register_PROTOABBREV` and `proto_reg_handoff_PROTOABBREV`.

The function `proto_register_PROTOABBREV` will first register all the fields of the protocol (name, data type, and description). This registration is needed for Wireshark to add the fields as items or subtrees during dissection and to specify how the fields will be displayed (see Wireshark documentation[9], section 1.6). All possible subtrees must be registered, this is done to remember if a subtree was expanded or collapsed when changing the view of the packet. Additionally the complete name of the protocol and its abbreviation are registered.

With a call to `proto_reg_handoff_PROTOABBREV` the dissector registers how packets from this protocol should be recognized. The transport protocol in combination with a port number can be used in many cases to detect a specific application layer protocol. If the port number is not fixed, then a *heuristic dissector* should be used instead of the previous mentioned non-heuristic dissector (see section 3.3.4).

The actual dissection functionality is implemented in `dissect_PROTOABBREV`. The buffer mentioned in section 3.3.1 is passed to this function. The dissection starts with an offset set to the beginning of the buffer and reads the first field of the packet. This offset is used as a reference to the dissector to enable it to know which byte in the buffer comes next. The offset increases by the number of bytes a dissected field spans. The field will be added as an item in the protocol tree to display its content. Some fields can be added as subtrees in which case they can contain multiple items. A flag field are commonly added as a subtree where a summary of all flags are shown. Each flag is added as an item in the flag subtree. If the payload of the packet contains a packet from another protocol, then the corresponding dissector must be called to dissect this payload.

3.3.3 Dictionaries

Some protocols use dictionaries to translate numerical values (keys) into more understandable strings. The dictionaries are used to facilitate the analysis of the protocol's packets. For example, the Diameter dissector uses a combination of two methods a XML dictionary and value strings[10].

The XML-dictionary can be read by the dissector to translate each key to an explanatory string and then adding the string as an addition to the key in the protocol tree. Some smaller sets of key to string representations are stored in

arrays of *value strings* which are part of the dissector's source code. A *value string* is a struct (represented as "`value_string`" in C) containing a key and a string. Wireshark uses the value string array to translate a field of the protocol and presents the translated field instead of the key. A field using a value string array must register its use in the function `proto_register_PROTOABBREV` (see Wireshark documentation[9], section 1.6).

3.3.4 Protocol detection

In the function `proto_reg_handoff_PROTOABBREV` the dissector registers how the packets will be detected. There are three alternatives to do this, either with a non-heuristic dissector, a heuristic dissector, or a combination of both.

A non-heuristic dissector registers a transport protocol in combination with a port number. This alternative can be used for protocols with a reserved port number assigned by IANA[11]. Even with a port number reservation, protocol traffic from a different protocol can appear at a reserved port number. Wireshark will then attempt to dissect the packet with a incompatible dissector and the dissection will fail. The user can manually choose which dissector to use.

To solve this problem Wireshark has heuristic dissectors[12]. These are only registered with the associated transport protocol(s). In addition the function `dissect_PROTOABBREV` must implement some heuristic tests to check if the captured data belongs the chosen protocol. This can be done by checking for combinations of fields that are unique for the protocol. When Wireshark receives for example, TCP packet data, the application will lookup the non-heuristic dissector registered to TCP and the port number of the traffic. If no suitable dissector was found, then the application will go through all the heuristic dissectors until one which fulfills all the heuristic conditions is found.

3.3.5 Dissector types

Dissectors can be implemented in two ways, as a built in or a plugin[13]. Both are included in the Wireshark source code, but the way to include them differs. The built in dissector is the most common and its source code is distributed with the source code of Wireshark. If a built in dissector is not contributed to the Wireshark source code, then a custom build is needed to include it. A custom build is made by manually compiling the Wireshark source code with the additional specific code that you want to include. The plugin dissector is built as shared library file, in UNIX as a so-file and a dll-file in Windows. This shared library file can be distributed as a plugin to an already installed version of Wireshark. To make Wireshark read the plugin users running UNIX-systems need to place the plugin file in a specified location (`~/.wireshark/plugins`).

Chapter 4

Performance evaluation

When working with protocols several tools/methods can be used for performance evaluation. By providing measured data from performance tests, an application suitable for data analysis can be used to evaluate the protocol's performance. Different tools for testing and analysis will be described in this chapter.

4.1 Tools for data analysis

When test data needs to be analysed there are many suitable tools for the job. These tools range from spreadsheets to more advanced programming environments for analysis. Advanced programming tools are more powerful for analyzing data and generating graphical presentations of this data.

4.1.1 Spreadsheet

Spreadsheets are simple and provide a common tool for calculation. These are typically manipulated with an application that is easy to work with. Some of the spreadsheet applications are Microsoft Excel[14], OpenOffice Calc[15], and iWork Numbers[16]. All of them provide the user with tools for statistical calculations. Excel and Numbers require a (paid) license, but OpenOffice Calc is free software.

4.1.2 Matlab

Matlab is a programming environment developed by MathWorks. The application is designed to be a powerful tool for data analysis, visualization, algorithm development, simulations, and numerical computations. Matlab offers a large library of built-in functions which makes it a great tool for all forms of calculations[17].

Matlab is a commercial program and some of the packages are also commercial software.[18]

4.1.3 R

The R programming language and environment was developed for statistical computation and graphics. The R website says: "Many users think of R as a statistics system. We prefer to think of it of an environment within which statistical techniques are implemented"[19]. R is an open source program and built upon a similar language called S [20]. The programming language makes it possible for users to write functions in addition to using the built in ones. If necessary the user can manipulate R objects by using C[21], although this is not as simple as writing a R Script.

A R Script is a text file containing basically the same command as the user would input via the command line. To run a R Script file after having already launched R from a command line you say:

```
source(directory/file.R)
```

or run the script directly from the command line with an (optional) input file by typing:

```
R --slave file.R < [some_file]
```

4.2 Previous evaluations

EVH2 has not been evaluated before, although functions of the EVH2 library has been added with hope to improve performance. No tests have been made to ensure that these added functions actually improved performance. A performance evaluation will provide the necessary data in order for quality of service (QoS) limits to be set.

4.3 Performance questions

To get the packets delivered and acknowledged faster between the software components the round trip time (RTT) needs to be minimized. Different data types and message sizes may affect the RTT between hosts. The runtime of EVH2 library functions are relevant to measure from a regression testing perspective.

During a meeting with Aptilo a number of questions about the performance of EVH2 were discussed. Some of these questions are:

- **Can the number of integer types in EVH2 be reduced?**
  This would make the library functions less complex by handling fewer integer types.

- **How fast (in terms of execution time) are the EVH2 library functions for message creation?**
  Different messages could be tested with *grouped* or *ungrouped* attributes.

- **How big is the RTT difference when sending a boolean value versus an int with only one bit set?**

- **Is there a transmission time difference to send one message with multiple attributes or a stream of multiple messages with single attributes?**

- **Which of the two methods for reading *grouped* attributes from EVH2 messages has the fastest execution time? (`evh2_get_attr_grouped` and `evh2_cursor_*`)**
  The EVH2 library was first developed to support non-recursive access to attributes. Access to a *grouped* attribute is made by calling the function `evh2_get_attr_grouped`. This function creates and allocates memory of another EVH2 message and copies the attribute(s) into the new message. From this message the attribute(s) are accessed by the usual EVH2 library functions. If the accessed attribute in turn is grouped, then the function `evh2_get_attr_grouped` must be called again. To avoid memory allocation EVH2 library was extended with recursion supported functions (`evh2_cursor_*`) for traversing *grouped* attributes. There is no assurance that these functions are faster since no performance testing has been done. This "cursor" is capable of traversing the existing *grouped* attribute instead of allocating a new message.

Chapter 5

Method

When working with a team or a colleague the working method and planning are extremely important. A large part of the project concerns planning how things will be done. Choosing the right tool for the project is time-consuming since different projects require different tools. This chapter describes how this thesis was planned, along with a description of the tools that were used.

5.1 Work method

This subsection describes the method used and how it was applied to this thesis. In addition the tools that were used are briefly described.

5.1.1 Scrum

Scrum is an agile software development method for managing software projects and application development. The method is described in the following sections.

5.1.1.1 Roles

The *product owner* plays the role of the customer to the developer team and ensures that the team delivers what the customer expects. The product owner writes down the product parts in the form of items called tasks and prioritizes them. These tasks ends up in the *product backlog* (explained in section 5.1.1.2).

The *development team* usually consists of 3-9 people with different specification regions. These people are responsible for the product deliverable(s).

The *scrum master* is the person who is accountable for removing any impediments to the the team to deliver their sprint goal. An important part of the role is to be a supervisor to the *developer team* and keep the team focused on tasks in the current *sprint* (explained in section 5.1.1.3) .

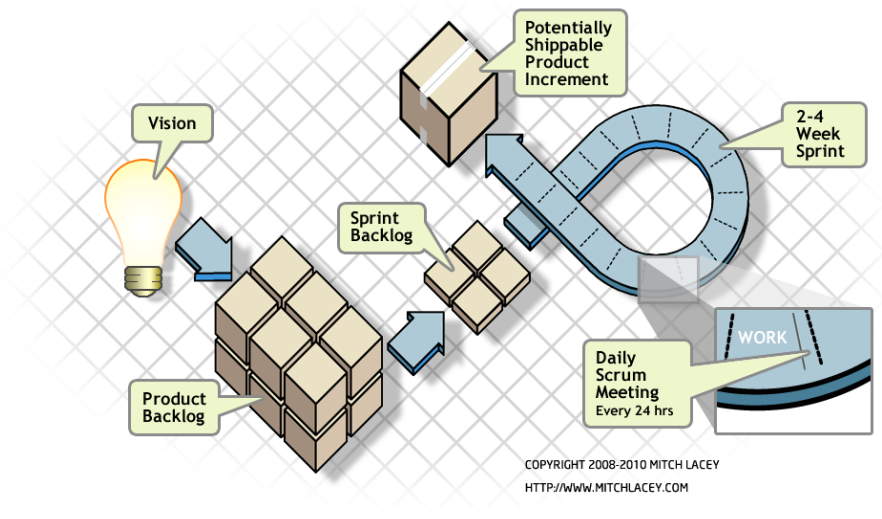

Figure 5.1: Overview of the scrum method [1]

5.1.1.2 Backlog

The backlog is divided into to a *product backlog* and a *sprint backlog* (see Figure 5.1). The *product backlog* contain all the tasks and requirements necessary to deliver the product. The *product owner* sorts these tasks according to their priority regarding risk, business value, and dependencies. Every 2-4 weeks the *product owner* sets up a new *sprint backlog* including the tasks to be done for that sprint.

5.1.1.3 Sprint

A sprint's duration varies normally from two weeks to a month depending on the project. Every sprint is introduced with a planning meeting where the tasks are identified and the time necessary for each task is estimated. During each sprint a daily *scrum meeting* is held, usually in the mornings (see Figure 5.1). The meeting is time fixed is usually and around 15 minutes long. The *scrum master* makes a brief check of what each team member did yesterday, how it went, and if any problems were occurred.

5.1.1.4 Burndown

Every task either gets rated by points or time by the *product owner*. This generates a *burndown chart*.

5.1.2 Redmine

To organize this project, a program called Redmine has been used[22]. Redmine is a web based project management system running on Ruby rails[23]. Tons of embedded features can be added and the tool can be customized to a specific project. With the Gantt feature (bar chart illustrating project schedule) time planning is clearly and easily arranged. A roadmap was setup where the "issues" were organized to get a better overview of the project. An "issue" in Redmine is equivalent to a Scrum task.

5.1.3 Applying the scrum method

Since this development team consisted of two persons the roles were not strict. The planning and development were done in a cooperative way. During the development process only a couple of *scrum meetings* were held because of the use of a shared workspace and continuous contact.

The project was divided into two parts. Each part was small enough to fit in one sprint which ended after a total of two sprints (see Table 5.1). The four weeks of development were splitted uniformly. The product backlog and the task-planning turned out to be trivial as each part of the project was placed in its own sprint backlog.

Table 5.1: Development sprints

Sprint	Description
1	Development of the EVH2 dissector
2	Development of test suite and corrections of the dissector

At the end of the first sprint a *demo meeting* was held with Aptilo Networks. The meeting began with a short presentation of the work achieved during the sprint. The presentation was followed by a hands on demonstration of the developed product to present its functionality. The implementation was presented by showcasing code where the structure of the code was explained and important parts were described. At the end of the meeting the participants were free to ask questions and encouraged to provide feedback. This demo gave an opportunity to get feedback before the next sprint.

In addition to the four weeks of development, two weeks were spent planning the project. Managing the tasks and the rest of the planning were done using

Redmine. The task board was reconstructed with a whiteboard and a task board-plugin for Redmine. Another four weeks were spent writing the report.

5.1.4 Git

Git is an open source distributed version control system used for handling development projects (more information is available at [24]). The reason why Git was used is because Aptilo manages their whole code base using this version control system. To participate in development a new *branch* was created from the existing code of *evh2tools* in the *master branch*. The *master branch* contains the original code of the project. A *branch* operation creates an exact copy of a *branch* and allows the developer to add new code and modify existing code without touching the *master branch*. When development is completed a *merge* operation is done to contribute the developed code to the *master branch*.

5.1.5 Documentation

During the literature study the main source of information was primarily Aptilo[2] and Wireshark documentation[25].

Information about the EVH2 protocol was found in Aptilo's internal documentation. This documentation can only be accessed from within the local domain at Aptilo.

During the development process of the dissector the Wireshark documentation was used frequently. The developer documentation provided guidelines of how to write code according to Wireshark's coding style (see Wireshark documentation[9], section 1.1). The code must follow this coding style since the Wireshark source code needs to be compiled on several platforms and using different compilers. The developer documentation provides basic skeleton code with example functions. This is a great starting point for dissector development (see Wireshark documentation[9], section 1.2).

Wireshark has a mailing list wireshark-dev[26] for developers. Questions about Wireshark development can be asked via this mailing list. All mail conversations are sent out to the subscribers and the mail conversations can be read online without a subscription. This mailing list was a very useful information source for dissector development.

5.2 Environments

Aptilo uses a custom operating system called *ALE*. The *ALE* operating system is their own product platform and is built upon a Linux distribution where their software components are running.

Another common operating system at Aptilo is Ubuntu, based on the Debian Linux distribution[27]. Since CentOS and Ubuntu are Linux distributions, they are both free unlike Microsoft Windows[28] operating system which is used by many developers at Aptilo.

5.3 Performance test setup

The performance testing were carried out on two *HP DL360 G6* servers running *ALE*. The specification of each server is a Quad-Core Intel(R) Xeon(R) E5504 @ 2.00 GHz processor and 4 GB of RAM. These severs were exlusively used for the performance testing in this thesis project.

Chapter 6

Implementation

This chapter will discuss the implementation of the dissector and test suite, how they were implemented, and why specific design decisions were made.

6.1 Dissector

The dissector was implemented to extend Wireshark to support *EVH2*. Using Wireshark as dissection application of *EVH2* traffic multiple protocols can be analysed together. Wireshark will facilitate the inspection of *EVH2* and other protocols, unlike *evh2sniff* which is only capable of inspecting *EVH2*.

6.1.1 Development process

The development of the dissector was split into two parts, a version without dictionary support following by a version containing dictionary support. This split was done to ensure a basic version capable of dissecting *EVH2* would be created. Since the dictionary was needed to lookup the data type of an attribute value the implementation of the basic version was skipped. This resulted into a single version of the dissector with the dictionary included from the beginning.

6.1.2 Wireshark source

The Wireshark source code provides the necessary libraries and functions for development of a generic dissector. The dissector's source code makes use of the Wireshark libraries and functions.

Wireshark is built with a series of makefiles generated by a configure script. The configure script generates makefiles for the dissector. The source code for Wireshark (version 1.6.7) was downloaded from the Wireshark web page[6].

6.1.3 Structure of the dissector

The dissector for EVH2 was implemented on Ubuntu 11.10 in C using the skeleton code in the Wireshark documentation (see Wireshark documentation[9], section 1.2). The skeleton code exemplifies the required functions (that were mentioned in section 3.3.2). The dissector implements the three required functions: `proto_reg_handoff_evh2`, `proto_register_evh2`, and `dissect_evh2`. In addition to these functions `dissect_evh2_attribute` was implemented for dissection of the attributes.

The function `proto_reg_handoff_evh2` registers the dissector in Wireshark for EVH2 detection (see section 3.3.4). All fields of the EVH2 protocol are registered by the function `proto_register_evh2`. This function defines the name, data type, and description for each field. The attribute value field is defined once for each possible data type. This multiple definition of the value field is done because Wireshark use these definitions to determine how to present the data in the Packet Details pane.

The function `dissect_evh2` dissects and adds the required header fields except for the flag field as items. The dissected flag field is added as a subtree containing each flag bit as an item. The flag field was implemented as a subtree to give a summary of all flags (in hexadecimal) in a collapsed view, but with details about each flag in an expanded view (see Figure 6.1).

During the dissection the offset is increased for each dissected field. When the offset reaches the length of the packet, then the dissection will end. If the header fields are dissected and the offset is less than the length, the EVH2 packet contains one or more attributes that needs dissection.

The function `dissect_evh2_attribute` is called from `dissect_evh2` for each attribute to be dissected. It will add the attribute as a subtree with the length field and key field added as items. If the attribute type is ungrouped, then the value field is added as an item according to its data type, which is registered in `proto_register_evh2` (as mentioned in section 3.3.2). In case of a grouped attribute `dissect_evh2_attribute` is recursively called to dissect the attribute(s) within the group. Each recursive call will add an attribute subtree instead of a value field item. The recursion will traverse and dissect the complete branch of the grouped attribute. When a attribute is dissected `dissect_evh2_attribute` returns the total number of dissected bytes.

The bytes returned from the function call to `dissect_evh2_attribute` inside `dissect_evh2` will be added to the offset. While the offset is smaller than the packet length, dissection of attributes will continue. When the offset reaches the packet length the dissection is complete (see figure 6.1).

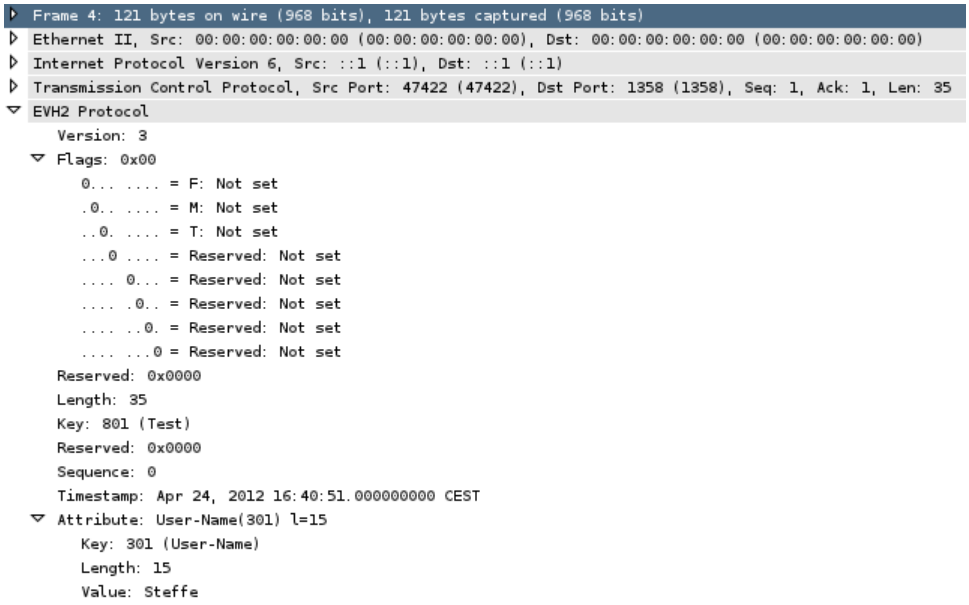

Figure 6.1: Dissected EVH2 fields presented in packet details pane

6.1.4 Dissector type

This dissector was first implemented as a built in dissector using the skeleton code. With this approach the complete Wireshark source needed to be recompiled to build the dissector. Since the dissector not will be contributed to the Wireshark source code it must be distributed in a custom build of Wireshark. When upgrading Wireshark to a newer version a custom build is always needed to include the dissector.

By developing a dissector plugin a custom build is not needed. The user can add the plugin to an existing version of Wireshark. The previously developed built in dissector code was reused to create the plugin dissector. In addition makefiles and code for plugin (a shared library file) creation was required. As the dissector plugin documentation[13] recommends the additional files were copied from the Gryphon plugin dissector folder in the Wireshark source code[29]. These files were edited according to the dissector plugin documentation to be compatible with the EVH2 dissector plugin. A dissector plugin can be built separately without recompiling the complete Wireshark source. Once its built, a shared library file containing the dissector is easily distributed to users with a binary copy of Wireshark.

6.1.5 Dictionary

A dictionary is needed to translate the keys of the events and attributes. Since Aptilo Networks had already developed a dictionary for EVH2, this dictionary was used in the dissector instead of writing a custom implementation using XML or value strings (see section 3.3.3). Using this dictionary only one external dictionary file needs to be maintained, e.g. when new keys are added they are added to this external dictionary file. The functions used by the dictionary are declared in a header file which is included in the dissector.

The dictionary is instantiated during the startup of Wireshark by a call from `proto_register_evh2`. The translated textual representation of a field will be appended to the item. The textual representation and the numerical are both presented in the Packet Details pane, as was shown for `key 301` in Figure 6.1.

6.1.6 Protocol detection

The dissector was at first developed as a non-heuristic dissector registered for TCP, UDP, and SCTP with the fixed port number 1358 in `proto_reg_handoff_evh2`. This port number was picked since the capture file used for testing had traffic on this port. EVH2 traffic may occur on port numbers other than the registered port number. When EVH2 traffic is captured on another port than the registered it will not be automatically recognized. Packets that are not automatically recognized can be dissected by manually specifying the packet's protocol type in Wireshark. Subsequently a heuristic dissector had to be implemented because the EVH2 traffic in general is not sent over a specific port number.

The code in `proto_reg_handoff_evh2` was changed to register the dissector as a heuristic dissector. Unlike the non-heuristic dissector this type of registration does not register a port number, but only the transport protocols that might be used, in this case UDP, TCP, and SCTP. Heuristic tests were implemented in the beginning of `dissect_evh2`:

- Packet size must be equal or greater than the size of the event header (20 bytes),
- Packet size must be greater than 28 bytes if attribute(s) are present,
- Version field must be set to the current version of EVH2 (currently 3),
- Reserved flags must be set to zero, and
- Reserved fields must be set to zero.

6.2 Test suite

A test suite was created to measure the protocol's current performance and to visualize changes in the protocol's performance. The following chapter will describe how this test suite is utilized. The following sections describe how the test suite was implemented.

6.2.1 Packet generator

To send lots of packets in a specific order and do this repeatedly a certain number of times, a lot of data needs to be sent. The packet generator is a bash script which generates a test case(s) in the form of an event(s) with an attribute(s) (see Listing 4 in Appendix B). The generator was implemented with the command line parameters: event type, attribute, number of events, and increase amount. The number of events is used to specify how many events it should generate. The size of each generated event is increased by increasing the number of attributes of each generated event (increases with the number of attributes specified in "increase amount"). Generated test cases can be stored in a file, this forms a test that can be used for performance testing. Figure 6.2 describes the structure of a test file containing multiple test cases.

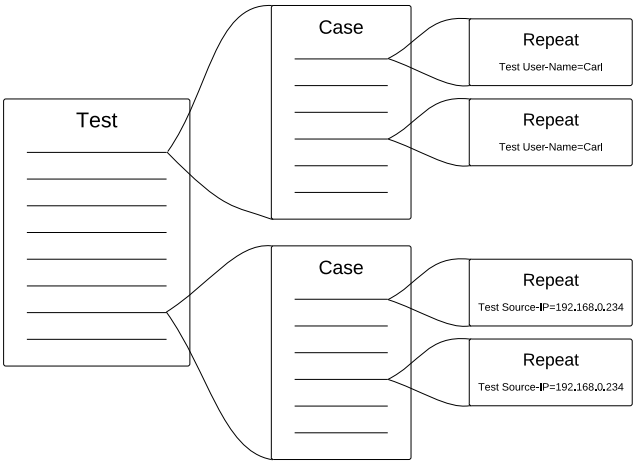

Figure 6.2: Structure of a test and its test cases

6.2.2 Evh2performance

A complementary application to the evh2tools was developed because of the present lack of performance testing. Evh2performance extends evh2send because the former tool's ability to send EVH2 messages. To answer the questions discussed in section 4.3 evh2performance was implemented with appropriate functionality for testing in these areas.

The application is written in C and can read EVH2 test cases by command line, STDIN, and from file. Each test case is passed to one of its three test functions that measures the RTT, message creation time, and grouped attribute access time for a test case. In addition to the test case time measurement a total test time is measured. This total test time is measured during the same period as a test case but it is incremented for each test case and therefore represents the sum of the test case measurements. All three test functions will measure the size of the tested EVH2 packages, providing an option to analyse the measured test times against the packet size. The three functions implemented to conduct the actual testing are described in sections 6.2.2.2, 6.2.2.3, and 6.2.2.4.

6.2.2.1 Accurate test data

To compare two different sets of test data the measurement must be performed with the same conditions, e.g. tests must be identical in the two measurements. Predefined tests were created to provide the application with tests usable for such a comparison. Comparing test data from different tests is pointless because of the lack of relation between the actual tests.

The accuracy of the measured data is increased with the number of measurements since a better mean value can be calculated. Multiple runs of the same test case will therefore increase the accuracy of the test data. This was achieved by implementing functionality to repeat each test case a given number of times during a test run. The number of repeats is specified by the flag `-R <num>` in Table 6.1, if the flag is not used repeat is set to one by default.

Table 6.1: The flags implemented in evh2performance

Flag	Full flag	Description
-h	--host	Select host
-p	--port	Select port
-d <val>	--debug	Set debuglevel, <val> 0 to 1
-r <val>	--showresponse	Output response packet information, <val> 0 to 4
-T <val>	--test	Choose the test type to run, <val> 0 to 2
-f	--files	Run predefined test(s) from file. Separate multiple files with ","
-C	--CSV	Output in CSV format
-s	--case-summary	Print summary of each case
-t	--table	Print table
-R <num>	--repeat	Repeat sending the same message <num> times
-u	--udp	Use UDP
-U	--unix	Use Unix socket
-S	--sctp	Use SCTP
-?	--help	Display help

6.2.2.2 Round trip time

To measure the round trip time of a EVH2 message evh2performance sends one or more EVH2 messages to a receiver (by default on TCP). After each message is sent the application will wait for an ACK to be received. The time from when the message is sent until the ACK is received is the measured RTT. Note that this time does not include any of the time required to create and read the packet size of the EVH2 message, only the round trip network delay, the processing time, and the time to generate the ACK. This test is run by using the test flag set to 0 (`-T 0`). Each test case will be repeatedly sent the number of times set by the flag `-R <num>`.

6.2.2.3 Creation time of EVH2 messages

This function tests the EVH2 library functions for EVH2 message creation, and is run with the test flag set to 1 (`-T 1`). This test measures the time is takes to create a evh2 message and add all its attributes. The time measurements exclude the time it takes to read the size of the created message, the size is not needed for message creation it is only measured for analytical purposes. This function will not send any of the created messages, it will only repeat the creation of each test case the number repeats set by the flag `-R <num>`.

6.2.2.4 Grouped attribute access

Access of grouped attributes in a EVH2 message is done by either using the `evh2_get_attr_grouped` or `evh2_cursor_*` functions. Due to time limitations in this thesis project this test function was only implemented with support to test attribute access with `evh2_get_attr_grouped`. The `evh2_cursor_*` functions are implemented in the EVH2 library but there are no functions that use them for access of grouped attributes. The library only provides functions for attribute access that use `evh2_get_attr_grouped` for grouped attributes. There was not enough time to write new library functions using `evh2_cursor_*` for comparison of the two methods.

The actual testing is done with the test flag set to 2 (`-T 2`). A EVH2 message is created from a test case and the attributes of the message are all accessed by the library functions for attribute access (library functions that use `evh2_get_attr_grouped`). This access will be repeated as many times as set by the flag `-R <num>`. The time measurements is done only for the access, creation time of the EVH2 message is not included.

6.2.2.5 Data output

The default output from evh2performace is a summary of the measurements done in the executed test (see Listing 7 in Appendix C). Regardless to the test function used it displays the number of messages, number of test cases, number of repeats and total runtime. Mean, max, and min values is presented accordingly to the measurements of the test function used e.g., RTT, message creation time, or traversal time.

To get a detailed results view the flags `-t` and `-s` are used. Evh2performance then prints measurements of every message sent in a user friendly table format, at the end of each test case a summary table is displayed (see Listing 8 in Appendix C). The table displays the following data:

- First column is an enumeration of the total number of repeats in a test (1, 2, 3, ...),
- Second column is the name of the file containing the test case(s),
- Third column is the size of the EVH2 message (in bytes),
- Fourth column is the measured time (Round trip, message creation or traversal. Measured in milliseconds), and
- Fifth column is the total test time of a test case (measured in milliseconds).

By generating this output to STDOUT the developer can pipe the output to another application, redirect it to a file, or simply view it in their console window. The output format can be changed by specifying the optional parameter "-C" (see Apendix). This option will format the output as comma separated values (i.e., as a CSV file). The CSV format was chosen due to both its simplicity and because it is frequently supported by the analysis packages of many companies[30]. The use of this format increases the opportunity to use individual analysis programs. At Aptilo Networks, the most common analysis program is Microsoft Excel (see Figure A.1 in Appendix A).

6.2.3 R analysis

R was partly chosen because of its command line support rather than the GUI based Matlab. As noted earlier R is an environment for statistical data analysis and graphics, while Matlab's origins are in numerical computation. Additionally, R is to preferable since it is free[19], unlike Matlab[31] The following sections describes how R was utilized to analyse the test results.

6.2.3.1 R Script development

To analyse the output data from evh2performance a R script was developed. A R script was suitable due to the frequent measurements that a developer will make and script's command line support. The script initializes the config parameters, e.g: the required external packages, paths, and measurement units. To facilitate the use of the script with external packages functionality was developed for the script to check if every package given in the config parameters is installed, if not - then the missing packages will thereafter be installed.

This script includes two packages, one called *ggplot2* (and possible dependencies) and is used for graph creation. Among the fifty functions in the 'ggplot2 reference manual' only a few were relevant when for visualizing the test data from evh2performance. Each function has serveral parameters called *aesthetics*[32]. These complementary functions were used in the script:

- `geom_points` - a scatter plot which displays the relationship between two continuous variables. The colors relative to the columns were set with the aesthetics. This function is used to get a more detailed overview of the test data than just a curve (An example of this is shown in Figure E.3 in Appendix E).
- `stat_smooth` - a curve fitting of the approximate values. The standard error (with a confidence interval of 0.95) relative to the curve was set with the aesthetics. This function is used to get a simple and more understandable overview of the actual test result (An example of this is shown in Figure 8.2 in Appendix E).

The second package included in the script is *gridExtra* and is used for table plotting.

6.2.3.2 Data analysis

The script parses formatted (comma-separated) CSV files and creates a *data set* from it. A *data set* is a common storage type in R and is often used when plotting. Plot-labels, such as titles, are set depending of the column names in the CSV-header:

```
[count],[case],[test],[packet_size],[test_times],[runtime],[timestamp]
```

There are three possible names in the `[test_times]`-column: `rtt` (Round trip time), `mct` (Message creation time), `tt` (Traversal time). Each abbreviation is mapped to corresponding name.

A feature in the script called *compare mode* was developed to compare multiple test runs. When this feature is used, all the CSV-files in the specified compare-path (the config parameters) will be compared and visualized in the same graph-plot. The test data will be grouped based upon their timestamp to distinguish the different runs. This feature is implemented in order to facilitate use of the test suite, since manual comparison of two different diagrams with (possibly) different axis scales is very time consuming. Two single graphs *without compare mode* can be seen in Appendix E (Figure 8.1 and Figure 8.2) together with a compounded graph in Appendix G (Figure 8.3).

The plot axes and other features are determined by user input (option arguments) and can be set from command line by saying: `--args [flags]`. The implemented flags can be seen in Table 6.2.

Table 6.2: The flags implemented in the R Script

Flag	Description
`p`	Set x-axis to packet size (default is time)
`c`	*Compare mode*
`d`	Visualizes each measured value as a dot in the plot.

6.2.3.3 Presentation

To present the test results from the R script a PDF-file is generated containing a graph based on a test run. The intention is for the developer to take a look at the graph and determine if changes to the code were an improvement or not. In addition to the graph several tables are generated using *gridExtra*. One table for each test contains the test cases and finally a table summarizes all tests. The tables will contain statistical calculations of the measured data such as: minimum, maximum, mean, and sum. The PDF-file will be created in the directory given in the config (default `./pdf/`) and the filename is automatically generated based on a template (`[y-axis]_[x-axis]_[timestamp].pdf`).

6.3 Usage of the test suite

The test suite is simple to use, with only three steps the developer will be able to notice potential performance differences.

Run the script with the current implementation of EVH2 and analyse the output:

```
R -slave -f suite.R < testdata --args (p d)
```

Make changes to the implementation, e.g. add a new feature or a bug fix.

Add the two test runs to the compare directory. Run the script again with *compare mode*:

```
R -slave suite.R --args c (p d)
```

Chapter 7

Results

This chapter will present the results from development of the Wireshark dissector and test suite. Additionally it presents results of performance testing done with the test suite.

7.1 EVH2 dissector

The development of a EVH2 dissector resulted in a plugin dissector for Wireshark, the plugin file is in the form of a shared object file (`.so` and `.dll`). The dissector is capable of automatically detecting EVH2 by heuristic tests that checks if the packet has the structure of a EVH2 packet. The dissection routines dissects the complete evh2 packet and translates key fields in the header and attributes with the application EVH2 dictionary. The dissector is capable of dissecting all attribute types mentioned in section 2.3.2. The only flaw is dissection of *RADIUS* and *Diameter* attribute types that are displayed as a byte string, all other types are displayed in appropriate formats (see Figure 7.1). To use the dissector in wireshark the plugin must be present, additionally the EVH2 dictionary and its dependent packges must be installed. Dissection in Wireshark of different protocols alongside with EVH2 is possible since Wireshark with the plugin has support for EVH2.

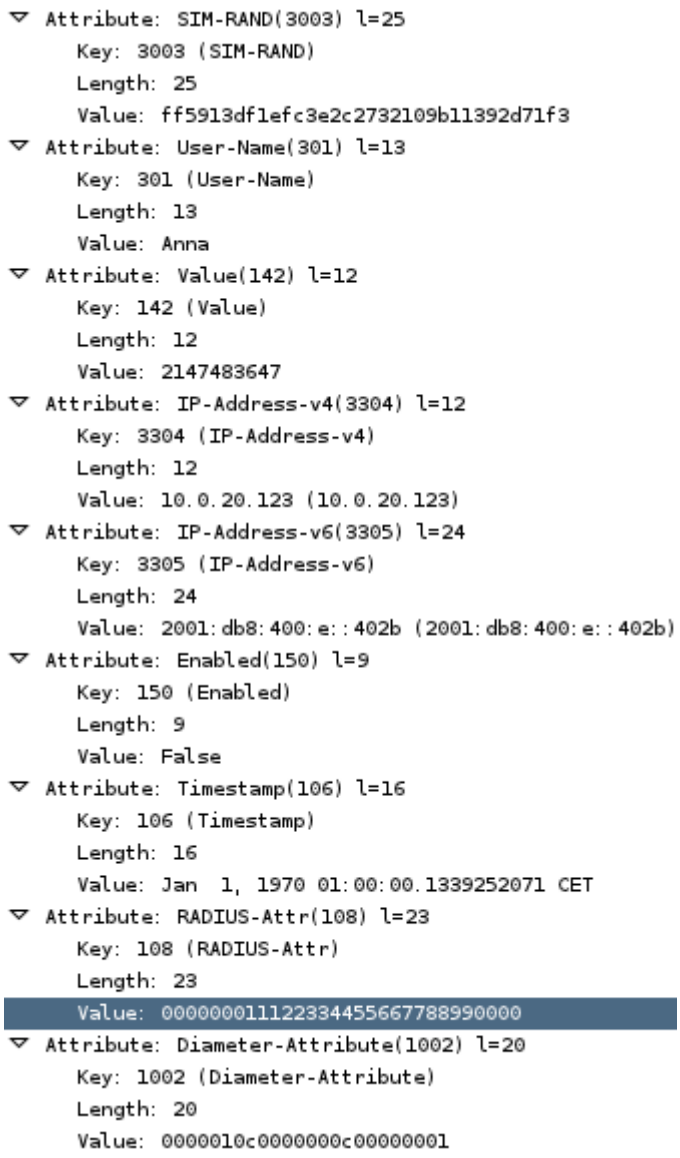

Figure 7.1: Different types of attributes

7.2 Test suite measurements

The test suite consists of three tools: Packet generator, evh2performance, and a R-script. Evh2performance is implemented to test round trip time, message creation time, and grouped attribute access time. Simple test cases can be created with the packet generator for use with evh2perfomance. The test results from evh2perfomance can be outputted in the form of a table or as CSV. CSV files are analysed with the R-script that generates PDF-reports with graphs of the

measurements and summary tables of the measurements. A comparison mode can be used to compare and present multiple CSV files in the same PDF-report.

The test suite was used to answer the questions in Section 4.3 by performing performance tests with evh2performance. CSV files with test data from evh2performance were analysed with the R-script, the different tests and the results are presented in the sections below.

7.2.1 Integer types

Can the number of integer types in EVH2 be reduced? To test the difference between integer types the round trip time was measured. It was done by sending 50000 messages of each of the following test cases:

`Test Status=127`
: Test case with int8 attribute (29 bytes)

`Test Protocol=127`
: Test case with int16 attribute (30 bytes)

`Test Value=127`
: Test case with int32 attribute (32 bytes)

`Test Duration-Left=127`
: Test case with int64 attribute (36 bytes)

The analysed test data is presented in Figure 7.2, the results shows that `int64` had fastest mean RTT followed by `int32`, `int16`, and `int8`. Since this was not expected six independent test runs of the same test case were made, the mean RTT differed between these tests. This independent testing was done for all test cases to get a more exact mean value. The graph and tables comparing the six independent runs of each test case are displayed in Appendix E and in Section F.1 in Appendix F. This resulted in new mean values shown in Table 7.1.

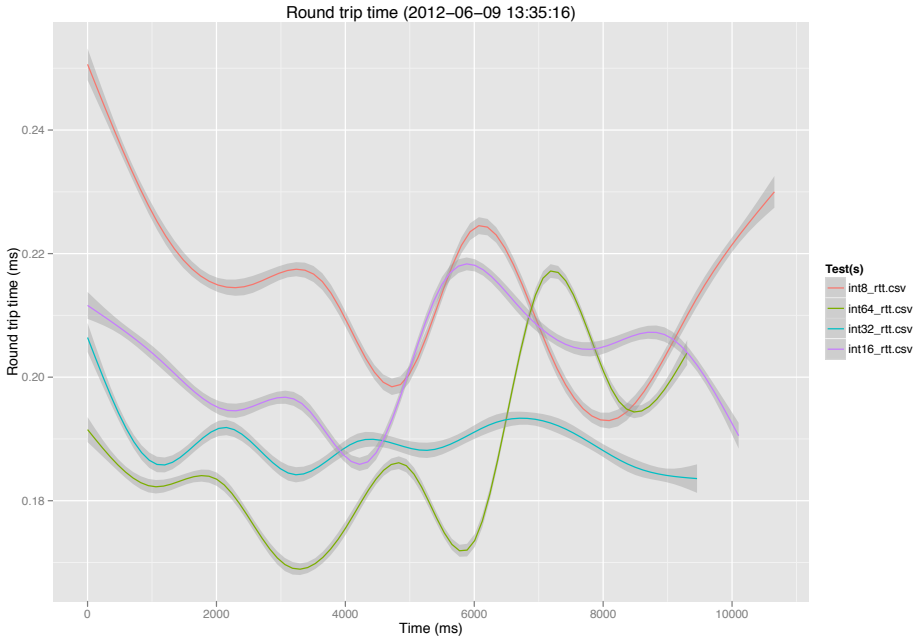

Figure 7.2: RTT comparison of integer types (int8, int16, int32, int64)

Table 7.1: The mean RTT of independent runs for the integer types

Data type	RTT
`int8`	0,183 ms
`int16`	0,176 ms
`int32`	0,188 ms
`int64`	0,195 ms

7.2.2 1 bit comparison

How big is the RTT difference when sending a boolean value versus an int with only one bit set? To test the difference between using boolean or integers for boolean values the round trip time was measured. It was done by sending 50000 messages of each of the following test cases:

`Test Status=1`
Test case with int8 attribute (29 bytes)

`Test Protocol=1`
Test case with int16 attribute (30 bytes)

`Test Value=1`
Test case with int32 attribute (32 bytes)

`Test Duration-Left=1`
Test case with int64 attribute (36 bytes)

`Test Enabled=1`
Test case with boolean attribute (29 bytes)

The RTT had the same behavior in this test as the test in section 7.2.1 where the mean RTT differed between independent runs. One run of each test case is compared in the graph in Figure E.5. Tests with six independent runs of each test case resulted in the mean RTT in Table 7.2. The test results of the six independent runs of each test case are displayed in section F.2 in Appendix F.

Table 7.2: The RTT of boolean and integer types (1 bit)

Data type	RTT
`int8`	0,203 ms
`int16`	0,197 ms
`int32`	0,197 ms
`int64`	0,191 ms
`boolean`	0,204 ms

7.2.3 Message creation

How fast (in terms of execution time) are the EVH2 library functions for message creation? Message creation time was tested by comparing the time it took to create messages with grouped attributes versus messages with ungrouped attributes. Each message was created 50000 times and the number of attributes in both tests were 27. The tests used are displayed in Listings 5 and 6. A graph of the results are shown in Figure E.6, and a summary of the results in Figure F.1.

7.2.4 Number of attributes

Is there a transmission time difference to send one message with multiple attributes or a stream of multiple messages with single attributes? This was tested by sending messages with a total of 100 attributes. Three tests were used: 100 messages with 1 attribute (see Listing 1 in Appendix B), 50 messages with 2 attributes (see Listing 2 in Appendix B), and 1 message with 100 attributes (see Listing 3 in Appendix B).

Each test was repeated 500 times to get accuracy of the measurements. The time it took for the different tests to send all events 500 times and wait for the ACK are displayed in Table 7.3 where the column Sum is a summary of all RTT measured. This test showed that sending 1 event with 100 attributes took the most time.

Table 7.3: Comparison of small versus big events

Test	Min (ms)	Max (ms)	Mean (ms)	Sum (ms)
`50event-2attr_rtt.csv`	0,099	0,487	0,16	3990,617
`1event-100attr_rtt.csv`	0,255	0,551	0,308	154,221
`100event-1attr_rtt.csv`	0,09	1,284	0,2	10023,423
`All tests`	0,09	1,284	0,223	14168,2261

Another test was conducted to see how the RTT is affected if the size of a packet exceeds the maximum transmission unit (MTU) of 1500 bytes. This was done by running a test containing events with an increasing number of attributes, the EVH2 packet sizes ranged from 32 bytes to 3140 bytes (see Listing 4). The result is displayed in Figure E.4.

7.2.5 Grouped attribute access

Which of the two methods for reading *grouped* attributes from EVH2 messages has the fastest execution time (`evh2_get_attr_grouped` and `evh2_cursor_*`)? Since the test function in evh2performance only supports access with the function `evh2_get_attr_grouped` no comparison could be done to answer this question.

Chapter 8

Analysis

This chapter will analyse the outcome of this thesis project. It will discuss the dissector plugin, test suite, and performance tests.

8.1 Analysing the dissector

Writing a EVH2 dissector plugin for Wireshark was a good solution to solve the problem of inspecting traffic from other protocols mixed with the EVH2 protocol. The personnel at Aptilo do not need to use evh2sniff in combination with Wireshark, because Wireshark now supports EVH2 through the plugin. This solution provides the necessary functionality from evh2sniff together with the additional powerful functionality of Wireshark as the analysis tools. Wireshark with the EVH2 plugin can be used for deeper analysis of EVH2 traffic.

The approach of developing the dissector as a plugin makes it easy to distribute and update. When a new version of the dissector is released the users simply update their plugin and can start to use the new version directly. The plugin was a better solution than a built-in dissector. The built-in dissector would require custom builds of Wireshark to be distributed to the users at Aptilo since the dissector code was not contributed to the source code of the Wireshark project.

The use of the evh2-dictionary in the dissector was a good solution from a development perspective. It saved development time since it was already implemented. However, from a user perspective this was not the best solution, the dictionary's dependencies on other packages made distribution and usage of the code a bit more complex. To use the dissector the user had to install all the packages on their workstation that this code depends upon. Because this plugin dissector was developed for internal use at Aptilo this was an acceptable solution even if it was a bit complex.

During the development process the dissector was implemented in two

versions: a non-heuristic and a heuristic version for EVH2 detection. Both solutions have advantages and disadvantages, the heuristic version automatically detects EVH2 packets on arbitrary port numbers using its heuristic tests. The non-heuristic version automatically detects packets sent on a fixed port number, but detection can manually be overridden. A disadvantage with the heuristic version is that it cannot be manually overridden, if the protocol is updated (e.g. there is a change in version number of the protocol) and if a packet fails a heuristic test, then Wireshark will not be able not detect that the packet in fact is a EVH2 packet. Such a change to the protocol will require that the heuristic tests must be adjusted to match the protocol changes, and a new plugin version must be distributed. The final solution was the heuristic version since EVH2 traffic is sent on arbitrary port numbers. Updating the heuristic tests and distributing a new version of the plugin in the event of protocol changes was considered to be more user friendly, rather than requiring the user to manually override the EVH2 detection every time they use the dissector.

The dissector plugin for Wireshark facilitates the inspection of EVH2 with a more user friendly application than evh2sniff. A comparison of the two application interfaces are shown in Appendix D (evh2sniff in Listing 12 and Wireshark in Figures D.1 to D.3). The traffic shown was captured at the *same* time using the *same* interface.

8.2 Analysing the test suite

There are a lot of different programmers and programmer types and it is rare that everyone has the same opinion. Evh2performance is a first step towards a platform, designed to suit as many developers as possible and to save them time when developing EVH2 applications. The command line output suits the programmer that only needs a quick performance test. Additionally, the CSV-file is perfect if the measurements are to be analysed in a graphical way. The test suite are already simple, but the usage can be simplified even more by creating bash scripts that perform the whole procedure (generate test, test, and plot results).

Evh2performance is capable of measuring the RTT, message creation time, and grouped attribute access time, but has the potential to expand the sets of test done since the developers can add further functions to this code.

One feature that made a big difference in the R-Script was the *compare mode*, both with aspect of time and simplicity. The script is run once, instead of once per test. The visual difference after studying Figure 8.1 and Figure 8.2 alone, followed by examing the combined graph (Figure 8.3). Since the ratio between the axes remains the same, the comparison is much simpler than trying to compare them separately. This will certainly save the developer a lot of time.

The tables were initially though to simply be a complement to the dot-graphs, but the tables turned out to be very valuable when an exact value is needed, rather than just dots in the plots. Further calculations can be performed manually using the data from the tables (e.g. in combination with the test data stored for longer periods and later comparing them).

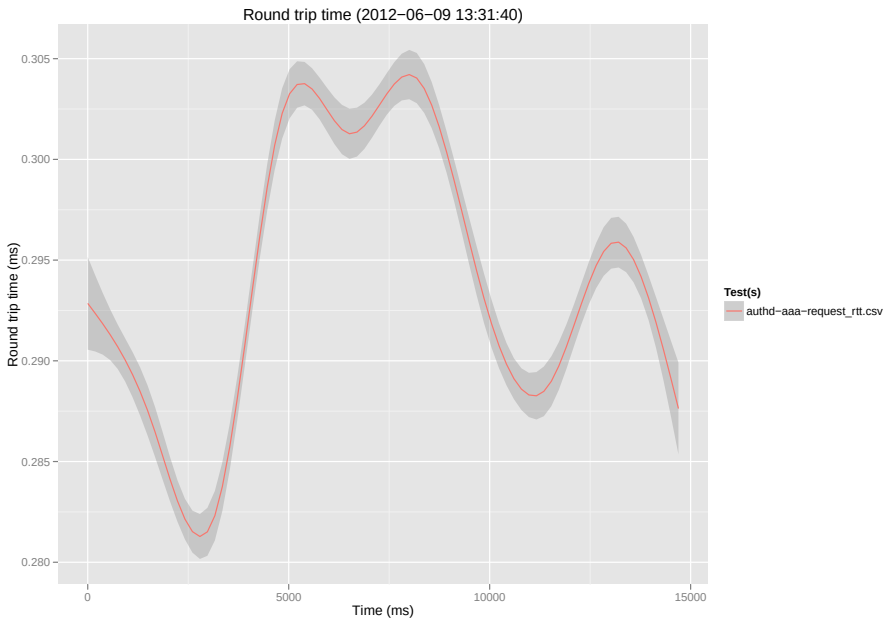

Figure 8.1: The graph of `authd-aaa-requeset_rtt.csv`

Figure 8.2: The graph of `grouped_rtt.csv`

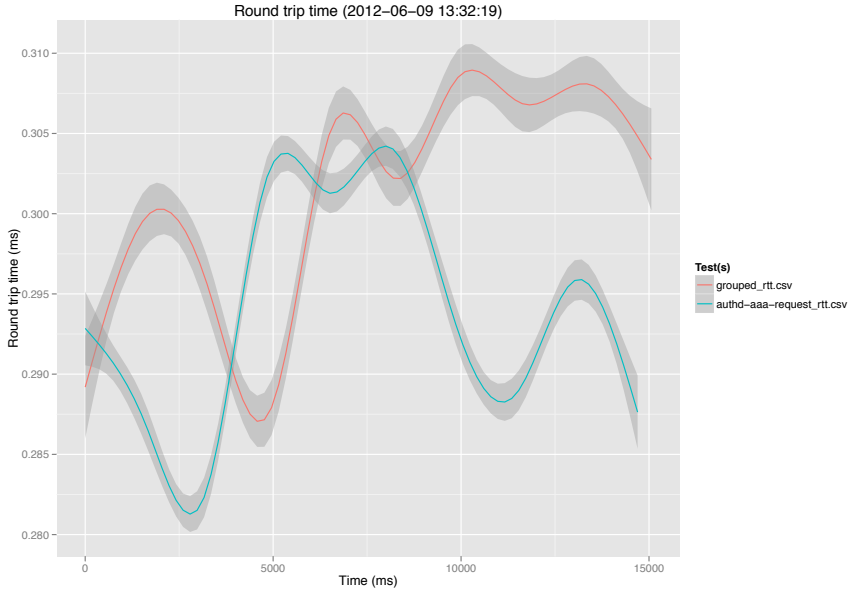

Figure 8.3: The graph of comparison between `grouped_rtt.csv` and `authd-aaa-requeset_rtt.csv`

8.3 Performance of EVH2

Performance tests were performed to gather measurement data to answer the questions in section 4.3. The data measured by evh2performance varied a lot between independent runs of the same test, this variation made it hard to get reliable data. To get a more reliable estimate of the actual performance the mean values of several independent runs were calculated manually. The tests results obtained from any single test run are not accurate enough to rely on for performance evaluation purposes.

- **Can the number of integer types in EVH2 be reduced?**
  Differences in RTT between the different integer types is small, the results show that there tends to be only a small difference, although int64 has the longest RTT. The test cases used to test the integer types were not big enough. The tests cases should contain more attributes of each type so that the packet size would vary greatly between an int8 and an int64 test case. Such a test case is expected to show a larger difference in RTT. Additional testing with better test cases is needed to answer this question.

- **How big is the RTT difference when sending a boolean value versus an int with only one bit set?**
  This test had the same problems as the previous were the test cases was not large enough. Hence this question also needs better testing in order to answer it. Analysing the packet sizes of int8 and boolean shows that the attributes value of int8 and boolean are both 1 byte in the EVH2 protocol. Boolean values may indeed have the same RTT as int8, but better testing is needed to verify this.

- **How fast (in terms of execution time) are the EVH2 library functions for message creation?**
  This test showed that creating EVH2 messages with grouped attributes takes longer time than to create ungrouped attributes. This was as expected since grouped attributes are more complex than ungrouped attributes. The differance in execution time is not very big, hence more extensive testing is necessary to get better results.

- **Is there a transmission time difference to send one message with multiple attributes or a stream of multiple messages with single attributes?**
  The testing showed that one attribute with multiple attributes had a significant faster transmission time to send each test 500 times (see Table 7.3). A message with 100 attributes has a size of 1200 bytes and in comparison 100 messages with one attribute each has a total size of 3200 bytes. The

overhead in packet size of sending a stream of messages affected the total transmission time. The conclusion of this test is that it is better to send messages with multiple attributes in a transmission time perspective. If the size of the transmitted packet exceeds the MTU of 1500 bytes the RTT is increased since the packet is fragmented into multiple packets. The graph E.4 shows that when the packet size gets close to 1500 bytes the RTT is increased, the same behavior can be seen at 3000 bytes when the packet is fragmented yet again. This must be considered when choosing the number of attributes to send in each EVH2 message.

- **Which of the two methods for reading *grouped* attributes from EVH2 messages has the fastest execution time? (`evh2_get_attr_grouped` and `evh2_cursor_*`)**
  This question could not be answered since the test function was not completed due to time constraints.

Chapter 9

Conclusion

During this thesis project a lot of new knowledge was gained about dissectors, Wireshark development and performance testing. Conclusions were made that test cases are extremely important when testing the performance of a protocol. The test cases in this thesis were too trivial to get useful knowledge about the EVH2 protocol from a performance perspective. To achieve reliable test results more complex (e.g. grouped and longer) test cases needs to be used for testing. Additionally the tests show that the conditions of the test system changes due to running the previous test cases. Hence it is necessary to avoid using data from the early part of a test run as these early events were likely to be processed when the system was largely idle (hence these results are not comparable with tests performed when the system is already busy).

When developing dissectors for Wireshark, it is *really* important to read the developer documentation[25]. This documentation was a big help during the development of the dissector. Another great information source about dissectors was the source code of other dissectors. By reading the code written by other developers a great deal of useful knowledge was gained. This knowledge gave an understanding of some of the problems encountered and helped to solve them. Generally both the developer documentation and the source code are valuable for anyone who wants to develop dissectors for Wireshark.

9.1 Goals

The main goal was the development of the two tools: a test suite and a dissector, were successfully achieved. The Wireshark dissector was more important to Aptilo than the test suite, but in the end it is up to developers to use these tools. The dissector can be used today and will probably be a great solution for Aptilo's software developers. The test suite on the other hand will need major improvement

before it can be used as a general performance analysis tool.

The Wireshark dissector is able to dissect EVH2 packets along with other protocols and present the content for the user. Because of this the inspection is much better and more detailed than before with evh2sniff.

The test suite was able to test the runtime of EVH2 library functions and round trip time of EVH2 messages. Unfortunately, due to time limitations, no CPU-loads were measured. Knowing the CPU load would be the next thing to measure during performance evaluation. First the library functions need to be implemented to measure the traversal time and of course an extensive test needs to be run in order to produce reliable data.

9.2 What has been left undone?

The only thing left undone in the EVH2 dissector is the presentation of RADIUS and Diameter attributes. In this version the code of the dissector is implemented to display these attributes as a byte string, this should be improved by adding functionality that parses these attributes serialized by EVH2, thus these attributes could be presented in a more understandable format.

In the evh2performance application the test function for attribute access was not completed with support for the `evh2_cursor_*` functions, when the EVH2 library is extended with these functions the test function should be completed. Additionally, a script could be developed that both runs the tests and analyses the results with the R-script.

9.3 Next obvious things to be done

Another dictionary implementation is an obvious thing to implement in a future version of the EVH2 dissector. A dictionary that uses the dictionary text file from the EVH2 dictionary and reads it into the dissector during start up of Wireshark. This version should be implemented to be independent of any of the packages developed by Aptilo. A new dictionary would make the distribution of the dissector less complicated and must be done if the dissector should be included in the source code of Wireshark.

The dissector could be implemented as a combination of a heuristic and non-heuristic dissector. This combination would allow packets to be automatically detected regardless of port number. Additionally, the non-heuristic part of the code would provide the support needed to manually override protocol detection. This would make it possible to manually detect EVH2 packets when a heuristic

test fails due to protocol changes. Information about this combination is described in the developer documentation [12].

Evh2performance must be improved with support to do multiple independent runs of the same test, so that test results could be computed from all these independent runs to provide higher accuracy of the test data. In addition to this larger and more relevant test cases must be developed that can be used for future performance analysis. Additional test functions could be added in evh2performance to extend its testing capabilities. As mentioned a warm up-timer needs to be implemented to avoid using data from early part of a test run. However, the most important challange is to run much longer test runs in order to achieve reliable test data.

9.4 Required reflections

This thesis project benefits Aptilo in economical aspects by reducing inspection time of EVH2 traffic, because a single application is less time consuming to use and therfore saves money. In addition a single traffic inspection tool minimizes the human error factor that may occur during use of two separate applications. Furthermore, the dissector and test suite can be used by Aptilo to find performance and functionality problems wich then can be corrected. This results in a better and more competitive software product. When problems are detected and corrected it will benefit the customers and product end users. User experience is enhanced especially if the performance is improved. Satisfied customers are very important since they will hopefully reccomend the product to others resulting in more sales.

The applications developed in this thesis project will help to reduce stress of the developers, since they provide an easy and quick way to inspect traffic and test performance. Test performance reports created by the R-script provides a base that is suitable for discussions during meetings due to its clear visualization of the measuements.

Bibliography

[1] Mitch Lacey. Scrum framework flow diagram - Mitch Lacey & Associates - scrum and agile training. [Accessed: May 23, 2012], 2010. URL http://www.mitchlacey.com/resources/scrum-framework-flow-diagram.

[2] Aptilo Networks AB. Aptilo documentation, April 2012.

[3] Wireshark Foundation. Wireshark documentation. [Accessed: April 12, 2012], n.d. URL http://anonsvn.wireshark.org/wireshark/trunk/doc/.

[4] Tcpdump/Libpcap. Tcpdump/Libpcap public repository. [Accessed: May 2, 2012], n.d. URL http://www.tcpdump.org/.

[5] Riverbed Technology. WinPcap - home. [Accessed: May 2, 2012], n.d. URL http://www.winpcap.org/.

[6] Wireshark Foundation. Wireshark · mailing lists. [Accessed: May 23, 2012], n.d. URL http://www.wireshark.org/lists/.

[7] Wireshark Foundation. Wireshark · about. [Accessed: May 2, 2012], n.d. URL http://www.wireshark.org/about.html.

[8] Chris Sanders. *Practical Packet Analysis, Second Edition*. No Starch Press, second edition, June 2011. ISBN 978-1-59327-266-1. URL http://my.safaribooksonline.com/9781593272661.

[9] Gerald Combs, James Coe, Gilbert Ramirez, James Foster, Olivier Abad, Laurent Deniel, Guy Harris, and Ulf Lamping. README.developer. [Accessed: April 12, 2012], n.d. URL http://anonsvn.wireshark.org/wireshark/trunk/doc/README.developer.

[10] David Frascone and Luis E. Garcia Ontanon. Diameter dissector - wireshark SVN. [Accessed: May 16, 2012], n.d. URL http://anonsvn.wireshark.org/wireshark/trunk/epan/dissectors/packet-diameter.c.

[11] Internet Assigned Numbers Authority. IANA — internet assigned numbers authority. [Accessed: May 23, 2012], n.d. URL http://www.iana.org/.

[12] Ulf Lamping. README.heuristic. [Accessed: April 12, 2012], August 2008. URL http://anonsvn.wireshark.org/wireshark/trunk/doc/README.heuristic.

[13] Guy Harris and Ed Warnicke. README.plugins. [Accessed: April 12, 2012], n.d. URL http://anonsvn.wireshark.org/wireshark/trunk/doc/README.plugins.

[14] Microsoft Corporation. Microsoft excel 2010 - office.com. [Accessed: May 2, 2012], n.d. URL http://office.microsoft.com/sv-se/excel/.

[15] The Apache Software Foundation. OpenOffice.org. [Accessed: May 2, 2012], n.d. URL http://www.openoffice.org/sv/product/#calc.

[16] Apple Inc. Apple - iWork - numbers - create perfect spreadsheets in minutes. [Accessed: May 2, 2012], n.d. URL http://www.apple.com/iwork/numbers/.

[17] The MathWorks Inc. Function reference (MATLAB®). [Accessed: June 4, 2012], n.d. URL http://www.mathworks.se/help/techdoc/ref/f16-6011.html.

[18] The MathWorks Inc. MathWorks nordic - online store - buy MATLAB, simulink, and other products. [Accessed: May 23, 2012], 2012. URL http://www.mathworks.se/store/default.do.

[19] The R Foundation. The R project for statistical computing. [Accessed: May 2, 2012], n.d. URL http://www.r-project.org/.

[20] John Chambers. The S system. [Accessed: May 2, 2012], January 2001. URL http://stat.bell-labs.com/S/.

[21] The R Foundation. R - about. [Accessed: May 2, 2012], n.d. URL http://www.r-project.org/about.html.

[22] Jean-Philippe Lang. Redmine. [Accessed: May 15, 2012], n.d. URL http://www.redmine.org/projects/redmine/wiki.

[23] David Heinemeier Hansson. Ruby on rails: Download. [Accessed: May 15, 2012], n.d. URL http://rubyonrails.org/download.

[24] Git. Git. [Accessed: May 23, 2012], n.d. URL http://git-scm.com/.

[25] Wireshark Foundation. Wireshark SVN repository. [Accessed: May 23, 2012], n.d. URL http://anonsvn.wireshark.org/wireshark/trunk/.

[26] Wireshark Foundation. Wireshark · download. [Accessed: May 23, 2012], n.d. URL http://www.wireshark.org/download.html.

[27] Canonical Ltd. Ubuntu and debian. [Accessed: May 15, 2012], n.d. URL http://www.ubuntu.com/project/about-ubuntu/ubuntu-and-debian.

[28] Microsoft Corporation. Windows 7. [Accessed: May 15, 2012], n.d. URL http://www.microsoftstore.se/shop/sv-SE/Microsoft/Windows/Windows-7.

[29] Steve Limkemann and Olivier Abad. Gryphon plugin - wireshark SVN. [Accessed: April 16, 2012], n.d. URL http://anonsvn.wireshark.org/wireshark/trunk/plugins/gryphon/.

[30] Y. Shafranovich. RFC 4180 - common format and MIME type for Comma-Separated values (CSV) files. [Accessed: May 16, 2012], October 2005. URL http://tools.ietf.org/html/rfc4180.

[31] The MathWorks Inc. MathWorks nordic - MATLAB and simulink for technical computing. [Accessed: May 2, 2012], n.d. URL http://www.mathworks.se/.

[32] Hadley Wickham. Ggplot. [Accessed: May 20, 2012], n.d. URL http://had.co.nz/ggplot/.

Appendix A

Surveys

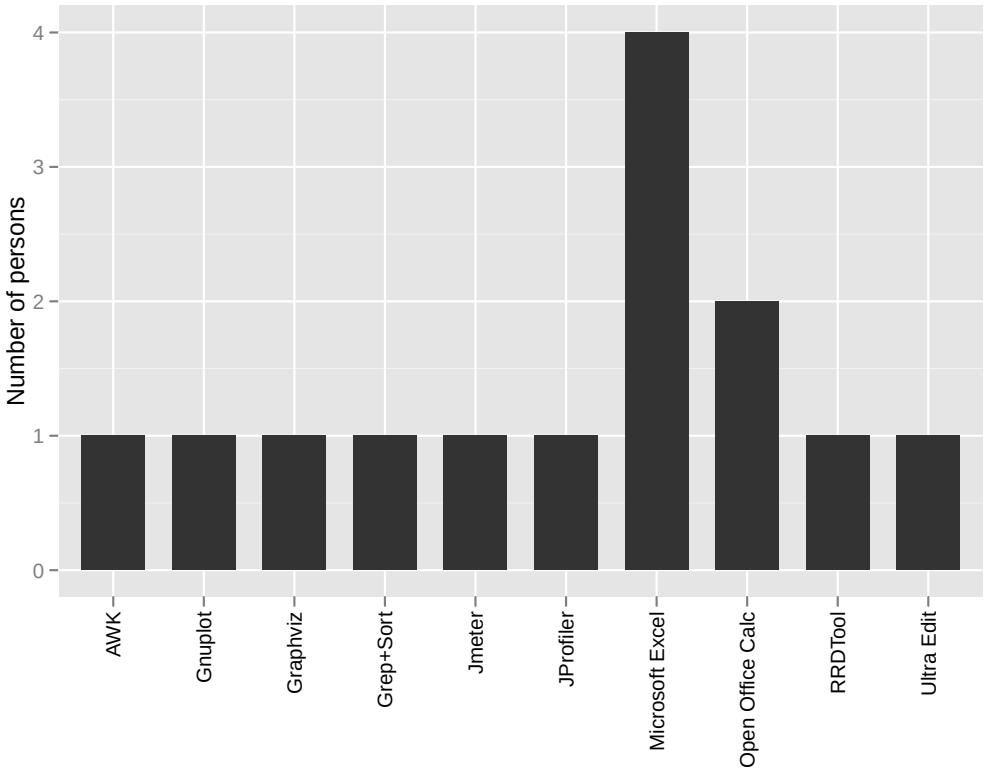

Figure A.1: Usage of analysis applications at Aptilo Networks

Appendix B

Tests

B.1 Ungrouped tests

```
Test Value=2147483647
Test Value=2147483647
Test Value=2147483647
.
.
.
```

Listing 1: Test for 100 messages 1 attribute (100 x 32 bytes = 3200 bytes)

```
Test Value=2147483647 Value=2147483647
Test Value=2147483647 Value=2147483647
Test Value=2147483647 Value=2147483647
.
.
.
```

Listing 2: Test for 50 messages 2 attributes (50 x 44 bytes = 2200 bytes)

```
Test Value=2147483647 Value=2147483647 Value=2147483647 ...
```

Listing 3: Test for 1 message 100 attributes (1 x 1200 bytes = 1200 bytes)

```
Test Value=127
Test Value=127 Value=127
Test Value=127 Value=127 Value=127
Test Value=127 Value=127 Value=127 Value=127
Test Value=127 Value=127 Value=127 Value=127 Value=127
Test Value=127 Value=127 Value=127 Value=127 Value=127 Value=127
.
.
.
```

Listing 4: Test for increasing size beyond MTU

```
Test DB-Field-Name="1"
  DB-Field-Name="1"
  DB-Field-Name="MSISDN"
  DB-Field-Value-String="46711223344"
  DB-Field-Name="1"
  DB-Field-Name="BILLINGPERIODST"
  DB-Field-Value-String="1"
  DB-Field-Name="1"
  DB-Field-Name="DATASPEED"
  DB-Field-Value-String="514"
  DB-Field-Name="1"
  DB-Field-Name="IPALLOCATION"
  DB-Field-Value-String="PUBLIC"
  DB-Field-Name="1"
  DB-Field-Name="SUBSCRIPTIONTYPE"
  DB-Field-Value-String="POSTPAID"
  DB-Field-Name="1"
  DB-Field-Name="COUNTRY"
  DB-Field-Value-String="SE"
  DB-Field-Name="1"
  DB-Field-Name="BWLPROFILE"
  DB-Field-Value-String="1"
  DB-Field-Name="1"
  DB-Field-Name="CUSTOMERTYPE"
  DB-Field-Value-String="CONSUMER"
  DB-Vclock=6bce61606060cc60ca05521c1f566d3ee27fa2d83a832991318f95e1
  DB-Key="46711223344"
```

Listing 5: Test containing one test case of an event with ungrouped attributes

B.2 Grouped tests

```
Test DB-Payload={
  DB-Field={
    DB-Field-Name="MSISDN" DB-Field-Value-String="46711223344"
  }
  DB-Field={
    DB-Field-Name="BILLINGPERIODST" DB-Field-Value-String="1"
  }
  DB-Field={
    DB-Field-Name="DATASPEED" DB-Field-Value-String="514"
  }
  DB-Field={
    DB-Field-Name="IPALLOCATION" DB-Field-Value-String="PUBLIC"
  }
  DB-Field={
    DB-Field-Name="SUBSCRIPTIONTYPE" DB-Field-Value-String="POSTPAID"
  }
  DB-Field={
    DB-Field-Name="COUNTRY" DB-Field-Value-String="SE"
  }
  DB-Field={
    DB-Field-Name="BWLPROFILE" DB-Field-Value-String="1"
  }
  DB-Field={
    DB-Field-Name="CUSTOMERTYPE" DB-Field-Value-String="CONSUMER"
  }
}
DB-Vclock=6bce61606060cc60ca05521c1f566d3ee27fa2d83a832991318f95e1
DB-Key="46711223344"
```

Listing 6: Test containing one test case of an event with grouped attributes

Appendix C

Command-line outputs

```
########################################################################

                    Summary of Round trip time test

########################################################################

Number of messages:               20000
Number of test cases:                 2
Number of repeats:                10000
Total runtime:              3963.681641 ms

Round trip time
-----------------------
Min:           0.0960 ms
Max:           0.4340 ms
Mean:          0.1981 ms
-----------------------

########################################################################
```

Listing 7: Summary output int evh2performance

```
   #              Test  Packet Size    Round trip time          Time
-------------------------------------------------------------------------
                              Test case 1
-------------------------------------------------------------------------
   1 authd-aaa-request    251 bytes           0.401 ms      0.401 ms
   2 authd-aaa-request    251 bytes           0.330 ms      0.731 ms
   3 authd-aaa-request    251 bytes           0.461 ms      1.192 ms
   4 authd-aaa-request    251 bytes           0.328 ms      1.520 ms
   5 authd-aaa-request    251 bytes           0.321 ms      1.841 ms
   6 authd-aaa-request    251 bytes           0.313 ms      2.154 ms
   7 authd-aaa-request    251 bytes           0.306 ms      2.460 ms
   8 authd-aaa-request    251 bytes           0.301 ms      2.761 ms
   9 authd-aaa-request    251 bytes           0.363 ms      3.124 ms
  10 authd-aaa-request    251 bytes           0.323 ms      3.448 ms

=========================================================================
                        Summary of Test case 1
=========================================================================
Number of messages:            10  Total runtime:        3.448000 ms
Round trip time
Min:        0.3010 ms   Max:        0.4610 ms   Mean:      0.3447 ms
=========================================================================

                                    .
                                    .
                                    .
```

Listing 8: Full table output in evh2performance (see Listing 9 and Listing 10)

```
                                .
                                .
                                .



                            Test case 2
------------------------------------------------------------------------
  11          grouped     426 bytes           0.324 ms      0.325 ms
  12          grouped     426 bytes           0.281 ms      0.606 ms
  13          grouped     426 bytes           0.318 ms      0.925 ms
  14          grouped     426 bytes           0.315 ms      1.240 ms
  15          grouped     426 bytes           0.278 ms      1.519 ms
  16          grouped     426 bytes           0.352 ms      1.871 ms
  17          grouped     426 bytes           0.273 ms      2.145 ms
  18          grouped     426 bytes           0.346 ms      2.491 ms
  19          grouped     426 bytes           0.311 ms      2.802 ms
  20          grouped     426 bytes           0.321 ms      3.123 ms

========================================================================
                        Summary of Test case 2
========================================================================
Number of messages:           10  Total runtime:         3.123000 ms
Round trip time
Min:        0.2730 ms   Max:        0.3520 ms   Mean:      0.3119 ms
========================================================================


                                .
                                .
                                .
```

Listing 9: Full table output in evh2performance (cont'd)

```
                                    .
                                    .
                                    .

######################################################################

                    Summary of Round trip time test

######################################################################

Number of messages:                  20
Number of test cases:                 2
Number of repeats:                   10
Total runtime:                6.571001 ms

Round trip time
-----------------------
Min:          0.2730 ms
Max:          0.4610 ms
Mean:         0.3283 ms
-----------------------
```

Listing 10: Full table output in evh2performance (cont'd)

```
"","case","test","packet_size","rtt","time","timestamp"
1,1,"authd-aaa-request",251,0.330000,0.330000,"2012-06-08 15:59:53"
2,1,"authd-aaa-request",251,0.346000,0.676000,"2012-06-08 15:59:53"
3,1,"authd-aaa-request",251,0.398000,1.074000,"2012-06-08 15:59:53"
4,1,"authd-aaa-request",251,0.438000,1.512000,"2012-06-08 15:59:53"
5,1,"authd-aaa-request",251,0.304000,1.816000,"2012-06-08 15:59:53"
6,1,"authd-aaa-request",251,0.321000,2.137000,"2012-06-08 15:59:53"
7,1,"authd-aaa-request",251,0.377000,2.514000,"2012-06-08 15:59:53"
8,1,"authd-aaa-request",251,0.362000,2.876000,"2012-06-08 15:59:53"
9,1,"authd-aaa-request",251,0.414000,3.290000,"2012-06-08 15:59:53"
10,1,"authd-aaa-request",251,0.317000,3.607000,"2012-06-08 15:59:53"
11,2,"grouped",426,0.335000,0.335000,"2012-06-08 15:59:53"
12,2,"grouped",426,0.363000,0.698000,"2012-06-08 15:59:53"
13,2,"grouped",426,0.281000,0.979000,"2012-06-08 15:59:53"
14,2,"grouped",426,0.286000,1.265000,"2012-06-08 15:59:53"
15,2,"grouped",426,0.338000,1.603000,"2012-06-08 15:59:53"
16,2,"grouped",426,0.361000,1.964000,"2012-06-08 15:59:53"
17,2,"grouped",426,0.289000,2.253000,"2012-06-08 15:59:53"
18,2,"grouped",426,0.261000,2.514000,"2012-06-08 15:59:53"
19,2,"grouped",426,0.352000,2.866000,"2012-06-08 15:59:53"
20,2,"grouped",426,0.273000,3.139000,"2012-06-08 15:59:53"
```

Listing 11: CSV output form evh2performance

Appendix D

Interfaces

```
[root@lab121-200 ~]# evh2sniff -i eth0 -f "port 1358"
Device: [eth0]
PCAP filter: [port 1358]
1: 192.168.121.201:48665 -> 192.168.121.200:1358
Event-Type = Session-Purged
Mon Jun 11 11:07:17 2012; Seq = 0
  Session-Time = 12312

2: 192.168.121.200:1358 -> 192.168.121.201:48665
Event-Type = Ack
Mon Jun 11 11:07:17 2012; Seq = 0

3: 192.168.121.201:48665 -> 192.168.121.200:1358
Event-Type = User-Account-Closed
Mon Jun 11 11:07:17 2012; Seq = 0
  User-Name = "Anna"

4: 192.168.121.200:1358 -> 192.168.121.201:48665
Event-Type = Ack
Mon Jun 11 11:07:17 2012; Seq = 0

5: 192.168.121.201:48665 -> 192.168.121.200:1358
Event-Type = Stop
Mon Jun 11 11:07:17 2012; Seq = 0
```

Listing 12: evh2sniff interface (displaying captured EVH2 data)

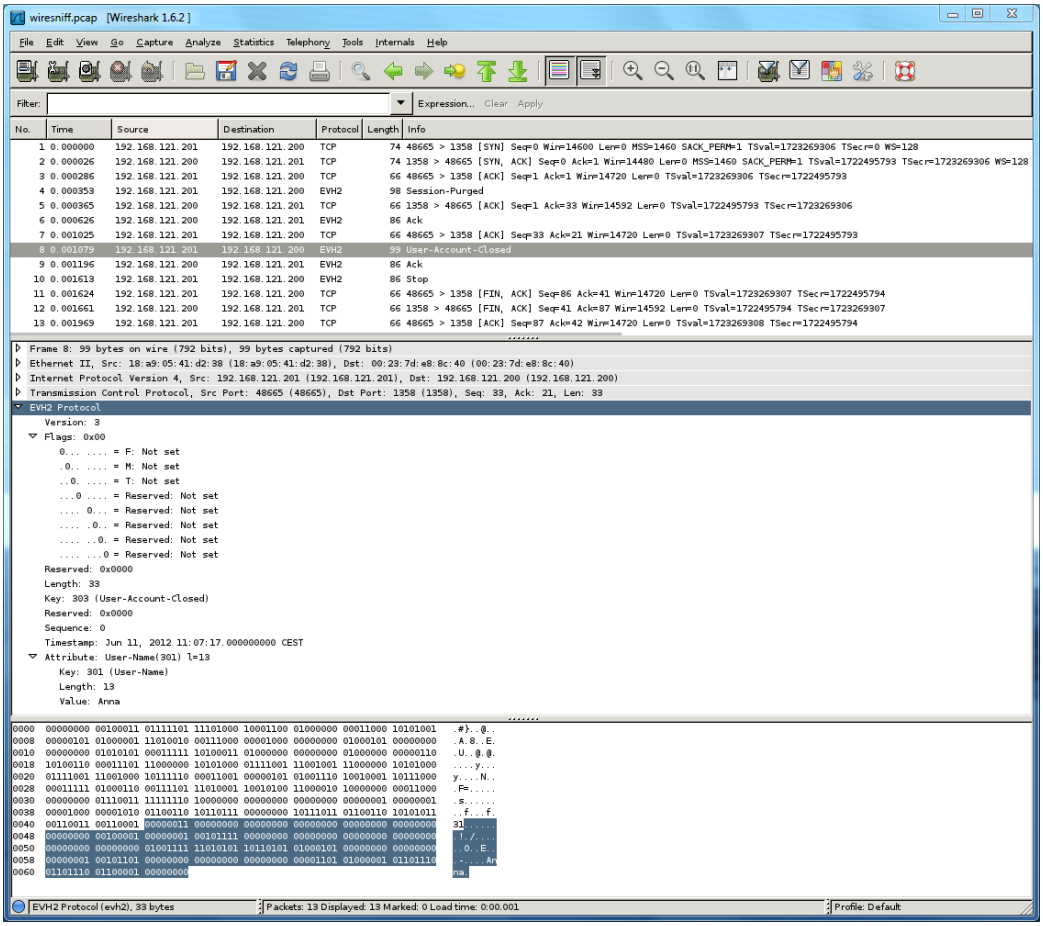

Figure D.1: Wireshark interface overview (displaying captured EVH2 data)

No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000	192.168.121.201	192.168.121.200	TCP	74	48665 > 1358 [SYN] Seq=0
2	0.000026	192.168.121.200	192.168.121.201	TCP	74	1358 > 48665 [SYN, ACK] S
3	0.000286	192.168.121.201	192.168.121.200	TCP	66	48665 > 1358 [ACK] Seq=1
4	0.000353	192.168.121.201	192.168.121.200	EVH2	98	Session-Purged
5	0.000365	192.168.121.200	192.168.121.201	TCP	66	1358 > 48665 [ACK] Seq=1
6	0.000626	192.168.121.200	192.168.121.201	EVH2	86	Ack
7	0.001025	192.168.121.201	192.168.121.200	TCP	66	48665 > 1358 [ACK] Seq=33
8	0.001079	192.168.121.201	192.168.121.200	EVH2	99	User-Account-Closed
9	0.001196	192.168.121.200	192.168.121.201	EVH2	86	Ack
10	0.001613	192.168.121.201	192.168.121.200	EVH2	86	Stop
11	0.001624	192.168.121.201	192.168.121.200	TCP	66	48665 > 1358 [FIN, ACK] S
12	0.001661	192.168.121.200	192.168.121.201	TCP	66	1358 > 48665 [FIN, ACK] S
13	0.001969	192.168.121.201	192.168.121.200	TCP	66	48665 > 1358 [ACK] Seq=87

Figure D.2: Wireshark packet list (displaying captured EVH2 data)

```
▷ Frame 8: 99 bytes on wire (792 bits), 99 bytes captured (792 bits)
▷ Ethernet II, Src: 18:a9:05:41:d2:38 (18:a9:05:41:d2:38), Dst: 00:23:7d:e8:8c:40 (00:23:7d:e8:8c:40)
▷ Internet Protocol Version 4, Src: 192.168.121.201 (192.168.121.201), Dst: 192.168.121.200 (192.168.121.200)
▷ Transmission Control Protocol, Src Port: 48665 (48665), Dst Port: 1358 (1358), Seq: 33, Ack: 21, Len: 33
▽ EVH2 Protocol
    Version: 3
  ▽ Flags: 0x00
      0... .... = F: Not set
      .0.. .... = M: Not set
      ..0. .... = T: Not set
      ...0 .... = Reserved: Not set
      .... 0... = Reserved: Not set
      .... .0.. = Reserved: Not set
      .... ..0. = Reserved: Not set
      .... ...0 = Reserved: Not set
    Reserved: 0x0000
    Length: 33
    Key: 303 (User-Account-Closed)
    Reserved: 0x0000
    Sequence: 0
    Timestamp: Jun 11, 2012 11:07:17.000000000 CEST
  ▽ Attribute: User-Name(301) l=13
      Key: 301 (User-Name)
      Length: 13
      Value: Anna
```

Figure D.3: Wireshark detail view (displaying captured EVH2 data)

Appendix E

Graphs

Figure E.1: The graph of `authd-aaa-requeset_rtt.csv`

Figure E.2: The graph of `grouped_rtt.csv`

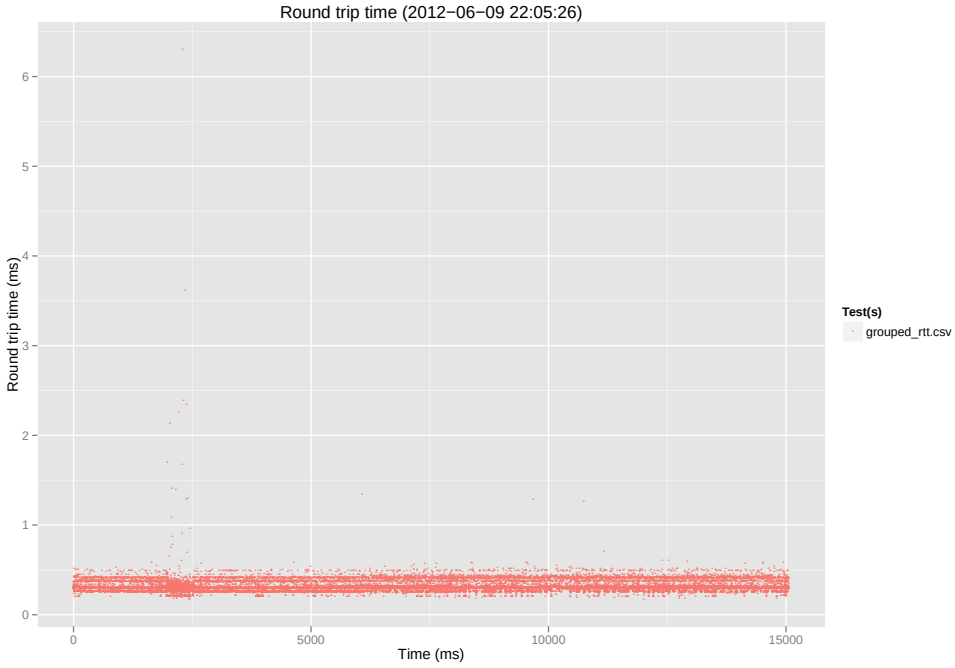

Figure E.3: The graph of `grouped_rtt.csv` with only dots

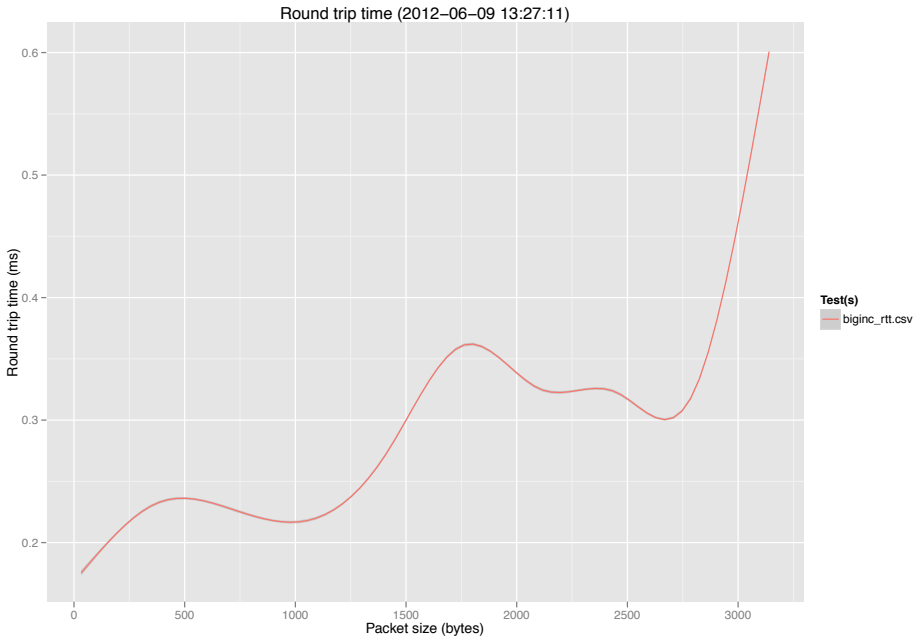

Figure E.4: RTT for events sent with increasing size

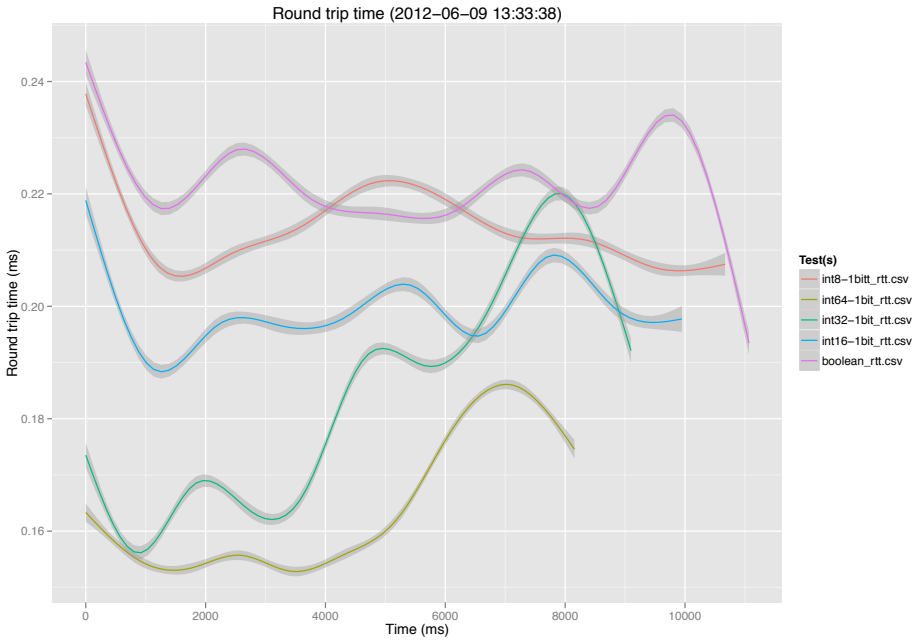

Figure E.5: RTT comparison of boolean versus integer with 1 bit set (boolean, int8, int16, int32, int64)

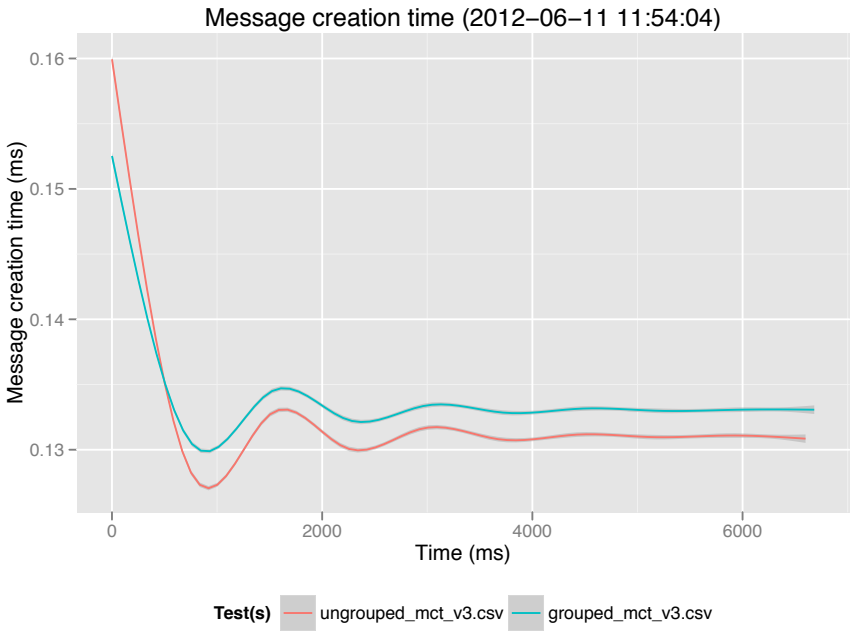

Figure E.6: The graph of grouped vs. ungrouped attributes

Appendix F

R-Script table outputs

	Test	Min (ms)	Max (ms)	Mean (ms)	Sum (ms)
1	ungrouped_mct_v3.csv	0.129	0.320	0.1319460	6597.300
2	grouped_mct_v3.csv	0.132	0.329	0.1336072	6680.361

Figure F.1: The summary of grouped vs. ungrouped attributes

F.1 Multiple independent runs

	Min (ms)	Max (ms)	Mean (ms)	Sum (ms)
int8_v6_rtt.csv	0.091	0.503	0.17	8511.818
int8_v5_rtt.csv	0.096	1.048	0.171	8553.229
int8_v4_rtt.csv	0.089	0.736	0.19	9516.111
int8_v3_rtt.csv	0.107	1.606	0.163	8165.13
int8_v2_rtt.csv	0.093	1.136	0.189	9458.082
int8_rtt.csv	0.092	2.46	0.213	10657.945
All tests	0.089	2.46	0.183	54862.315

Figure F.2: The summary of independent runs of int8

	Min (ms)	Max (ms)	Mean (ms)	Sum (ms)
int16_v6_rtt.csv	0.089	1.51	0.189	9474.041
int16_v5_rtt.csv	0.093	1.053	0.159	7939.764
int16_v4_rtt.csv	0.097	2.339	0.16	8020.979
int16_v3_rtt.csv	0.094	2.526	0.174	8684.863
int16_v2_rtt.csv	0.086	0.933	0.173	8646.585
int16_rtt.csv	0.103	1.003	0.202	10104.134
All tests	0.086	2.526	0.176	52870.366

Figure F.3: The summary of independent runs of int16

	Min (ms)	Max (ms)	Mean (ms)	Sum (ms)
int32_v6_rtt.csv	0.094	0.475	0.179	8954.298
int32_v5_rtt.csv	0.088	0.886	0.183	9127.588
int32_v4_rtt.csv	0.092	2.871	0.202	10119.961
int32_v3_rtt.csv	0.093	1.472	0.162	8102.843
int32_v2_rtt.csv	0.092	0.99	0.211	10529.958
int32_rtt.csv	0.084	1.08	0.189	9457.002
All tests	0.084	2.871	0.188	56291.65

Figure F.4: The summary of independent runs of int32

	Min (ms)	Max (ms)	Mean (ms)	Sum (ms)
int64_v6_rtt.csv	0.083	0.606	0.185	9265.098
int64_v5_rtt.csv	0.091	0.462	0.214	10677.987
int64_v4_rtt.csv	0.091	0.474	0.202	10114.76
int64_v3_rtt.csv	0.088	0.868	0.184	9218.64
int64_v2_rtt.csv	0.105	1.165	0.201	10037.852
int64_rtt.csv	0.088	0.476	0.186	9305.143
All tests	0.083	1.165	0.195	58619.48

Figure F.5: The summary of independent runs of int64

F.2 Multiple independent runs (1 bit)

	Min (ms)	Max (ms)	Mean (ms)	Sum (ms)
int8–1bitt_rtt.csv	0.081	0.786	0.213	10670.126
int8–1bit_v6_rtt.csv	0.116	0.486	0.204	10221.961
int8–1bit_v5_rtt.csv	0.091	1.207	0.207	10339.309
int8–1bit_v4_rtt.csv	0.091	1.288	0.204	10184.62
int8–1bit_v3_rtt.csv	0.083	2.553	0.192	9602.586
int8–1bit_v2_rtt.csv	0.089	0.847	0.2	10020.594
All tests	0.081	2.553	0.203	61039.196

Figure F.6: The summary of independent runs of int8 (1 bit)

	Min (ms)	Max (ms)	Mean (ms)	Sum (ms)
int16–1bit_v6_rtt.csv	0.09	1.123	0.188	9385.316
int16–1bit_v5_rtt.csv	0.085	2.308	0.21	10485.294
int16–1bit_v4_rtt.csv	0.1	1.348	0.212	10581.169
int16–1bit_v3_rtt.csv	0.113	0.868	0.174	8676.06
int16–1bit_v2_rtt.csv	0.091	1.391	0.2	9999.141
int16–1bit_rtt.csv	0.09	0.648	0.199	9949.628
All tests	0.085	2.308	0.197	59076.608

Figure F.7: The summary of independent runs of int16 (1 bit)

	Min (ms)	Max (ms)	Mean (ms)	Sum (ms)
int32–1bit_v6_rtt.csv	0.109	1.855	0.209	10465.294
int32–1bit_v5_rtt.csv	0.092	2.076	0.187	9345.883
int32–1bit_v4_rtt.csv	0.097	2.607	0.176	8782.475
int32–1bit_v3_rtt.csv	0.088	1.983	0.211	10536.042
int32–1bit_v2_rtt.csv	0.091	0.557	0.215	10764.54
int32–1bit_rtt.csv	0.114	3.137	0.182	9099.506
All tests	0.088	3.137	0.197	58993.74

Figure F.8: The summary of independent runs of int32 (1 bit)

	Min (ms)	Max (ms)	Mean (ms)	Sum (ms)
int64–1bit_v6_rtt.csv	0.112	0.662	0.19	9477.984
int64–1bit_v5_rtt.csv	0.084	3.546	0.184	9211.475
int64–1bit_v4_rtt.csv	0.09	1.172	0.214	10686.512
int64–1bit_v3_rtt.csv	0.092	0.688	0.211	10565.876
int64–1bit_v2_rtt.csv	0.087	0.517	0.182	9080.607
int64–1bit_rtt.csv	0.091	0.766	0.163	8156.027
All tests	0.084	3.546	0.191	57178.481

Figure F.9: The summary of independent runs of int64 (1 bit)

Appendix G

A complete comparison output

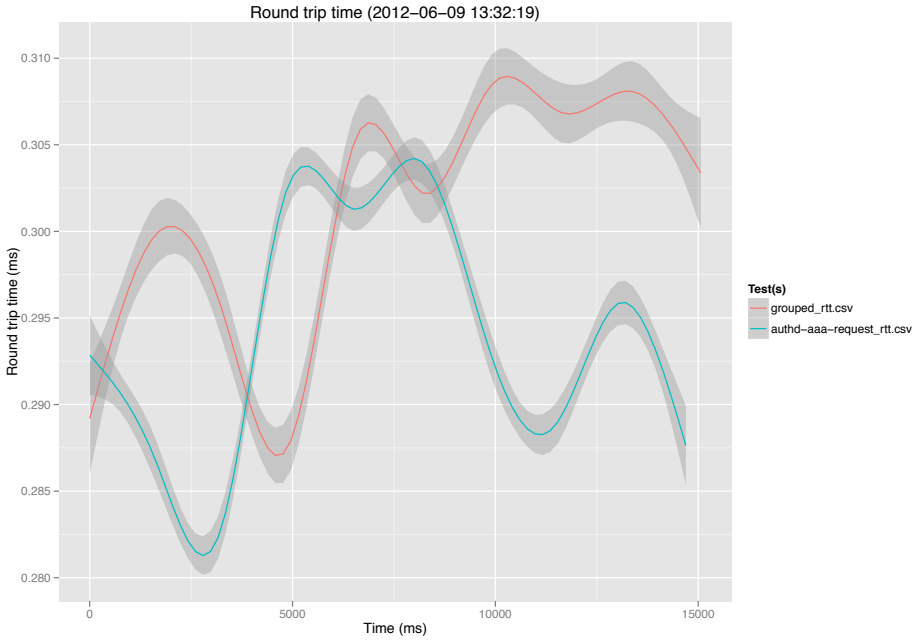

Figure G.1: The graph of comparison between `grouped_rtt.csv` and `authd-aaa-requeset_rtt.csv`

Case(s)	Messages	Min (ms)	Max (ms)	Mean (ms)	Sum (ms)
grouped	50000	0.174	6.299	0.3012872	15064.36

Figure G.2: The summary of `grouped_rtt.csv`

Case(s)	Messages	Min (ms)	Max (ms)	Mean (ms)	Sum (ms)
authd–aaa–request	50000	0.14	2.622	0.2939351	14696.75

Figure G.3: The summary of `authd-aaa-requeset_rtt.csv`

	Min (ms)	Max (ms)	Mean (ms)	Sum (ms)
grouped_rtt.csv	0.174	6.299	0.301	15064.361
authd–aaa–request_rtt.csv	0.14	2.622	0.294	14696.754
All tests	0.14	6.299	0.298	29761.115

Figure G.4: The summary of `grouped_rtt.csv` and `authd-aaa-requeset_rtt.csv`

TRITA-ICT-EX-2012:123

www.kth.se