MASARYK UNIVERSITY
FACULTY OF INFORMATICS

Generation of MSC Diagrams from Network Traffic

BACHELOR'S THESIS

Viktor Borza

Brno, Spring 2013

Declaration

Hereby I declare, that this paper is my original authorial work, which I have worked out by my own. All sources, references and literature used or excerpted during elaboration of this work are properly cited and listed in complete reference to the due source.

Advisor: Mgr. Matúš Madzin

Acknowledgement

I would like to thank my advisor Mgr. Matúš Madzin for valuable advice, comments and support during the writing of this thesis. I also want to express my gratitude to the members of the SCStudio project. Finally, I want to thank to my family for their support. This thesis was created within a joint project of Siemens Convergence Creators, s. r. o. and the research centre Institute for Theoretical Computer Science (ITI).

Abstract

Message Sequence Chart (MSC) diagrams were designed for modeling of message-based communication between entities in the network. They are used primarily by developing of a new network protocol as a description of design requirements. However, MSC can be as well used for visualization of real traffic captured from a network. One of the applications aimed at work with MSC formalism is Sequence Chart Studio (SCStudio), which is being developed in research centre Institute for Theoretical Computer Science at FI MUNI.

The thesis is focused on the problem of generating Message Sequence Chart (MSC) diagrams from PCAP (packet capture) file format. This functionality was integrated into SCStudio as a new feature for importing PCAP files. This paper describes the developing process and behaviour of implemented functionality.

Keywords

Message Sequence Chart, MSC, PCAP, Transformation, SCStudio, Traffic, Packet, Sniffer, Filter, Reassembling.

Contents

Chapter 1

Introduction

Nowadays computer is heavily used for communication and transmitting data over the Internet. The requirements on network protocols functionality and reliability grow up together with an improvement of network applications. However, to design the protocol properly, there are many aspects that have to be considered and various of scenarios which have to be described.

Very suitable method for modelling the network system behaviour is Message Sequence Chart (MSC) formalism [1]. It was initially developed by International Telecommunication Union (ITU) and includes textual and graphical representation. MSC formalism was primarily developed for requirement specification by designing of protocols. The MSC in form of diagram gives an overview of the message exchange among communicating entities, considering order of messages and time constraints. One of the tools that enables manipulation with MSC formalism is Sequence Chart Studio (SCStudio) [2]. SCStudio can be used for drawing MSC diagrams, import and export of MSC formats and also for execution of implemented verification algorithms on the Message Sequence Charts.

To verify the protocol's implementation, it is necessary to study its behaviour on real system. Firstly, we need to intercept and log traffic passing over the network which can be done by tool called sniffer. On the basis of the captured data it is possible to analyse network problems or gather traffic statistics. The primary format used for saving of the gained data is PCAP [3] that allows to analyse the captured data offline.

The MSC represents usually a communication scenario that is expected from protocol's behaviour. However in real network system there are a lot of problems like unreliable connections or overloaded communication channels. Therefore the interest has appeared, to present also the real traffic in form of MSC diagram. MSC provides a suitable way to get a better overview of the messages exchanging and their sequence in real communication.

This thesis was actually created as a part of the SCStudio. The main goal is to import the file in PCAP format to the SCStudio and represent it as MSC diagram. After the import, it is possible to run the verification algorithms that are available in SCStudio e.g. detection of deadlock or race conditions. Another feature is, that the transformation algorithm can be executed also from the SCStudio command line interface. The generated MSC is then in ITU-T Z.120 [1] textual representation.

For the purpose of explaining all issues related to the problematic of this thesis the paper was structured as follows. In Chapter 2 an MSC formalism is described and Sequence Chart Studio is characterized. The principles of traffic sniffing, filtering and capturing the packets are described in Chapter 3. There is also PCAP format mentioned and the popular libraries used for sniffing described. In the next Chapter 4 we explain the principles of parsing and rationalizing data from packets. The information gathered from traffic can be consequently used for generating an MSC diagram. This task is described in Chapter 5. If there are many packets in imported PCAP file, the created MSC can be quite big and unclear. For that reason we decided to use some kinds of abstraction, where more packets are displayed as one message in MSC. The first principle described in Chapter 6 concerns the reassembling feature. The other one, which allows a user to select specific type of aggregation is explained in Chapter 7. Regarding the amount of settings related to PCAP transformation, we have a graphical user interface created. It was added also a support for configuration files in SCStudio command line interface. Both of this topics are discussed in Chapter 8. The performance aspect of implemented PCAP import feature is discussed in following Chapter 9.

Chapter 2

Message sequence chart

Message Sequence Chart (MSC) formalism is primarily used for description of design requirements for concurrent systems such as networks or telecommunications software. Description expressed by MSC is given with formal notation and can be subjected to analysis of system behaviour. MSC depicts a potential exchange of messages among communicating entities in a distributed system and corresponds to a single (partial-order) execution of the system [4].

This formalism consists of two parts : Basic MSC (BMSC) and High-Level/Hierarchical MSC (HMSC). To represent the simple communication (scenario) Basic MSC is used. HMSC stands for a hierarchical structure of scenarios, where Basic MSCs can be combined to more complete system description. One HMSC can consist from many levels and each of them could contain nodes referencing to another HMSC or BMSC.

2.1 Basic message sequence chart

Basic Message Sequence Chart (BMSC) (Figure 2.1) consists of final set of system components communicating via messages. For each component exists a single *instance* that stands for a time axis. The *instance* contains *events* representing either receipt or sending of a *message*. The *events* can be situated in two kinds of areas: *strict order area* and *coregion area*. In *strict order area* the *events* are ordered totally, unlike the *coregion area*, where the *events* are unordered in default. The ordering is also denoted with *messages*. The *send event* of a *message* is ordered before *receive event* of the same message. *Message* that is not connected to any destination *instance* is represented with *lost message* and *message* without source is defined as *found message*. The *message's* description is situated above the *message* in *message label*.

For the purpose of describing *events'* time stamps and time restrictions there are components *absolute time* and *time interval* in the MSC structure. The *absolute time* represents *event's* timestamp related to start of communication. There are two types of *time intervals*. *Absolute interval* represents a time constraint between the start of a system communication and the *event*, while *relative interval* denotes the elapsed time with respect to the previous *events*.

There are two representations of MSC: graphical and textual. Textual standard is used for portability and compatibility between different software tools. In Figure 2.1 an example of BMSC diagram describing two devices communicating via HTTP protocol is to see. In this MSC, messages' timestamps are described via *absolute times* and duration of last message's transmission is presented in attached *relative interval*.

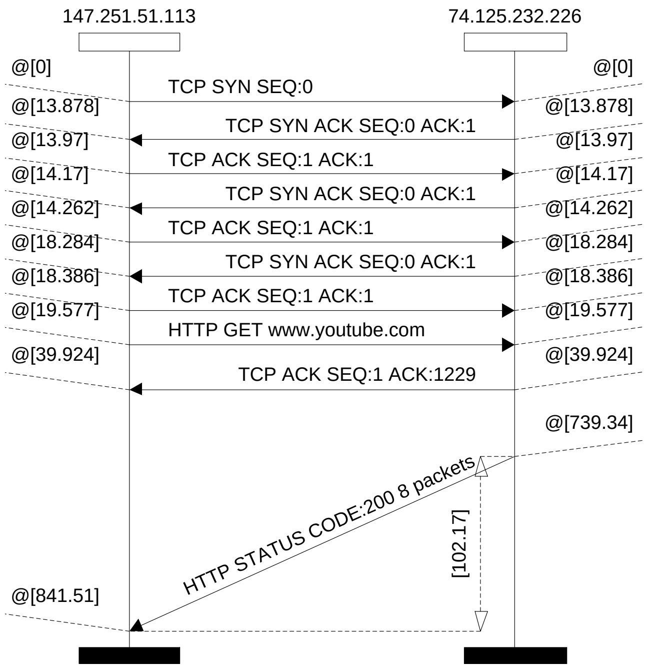


Figure 2.1: HTTP scenario

2.2 Sequence Chart Studio

The Sequence Chart Studio (SCStudio) is a user-friendly drawing and verification tool for Message Sequence Charts diagrams. The application was created as a joint project of Siemens Convergence Creators, s.r.o. and the research centre Institute for Theoretical Computer Science (ITI - Faculty of Informatics, Masaryk University) [2]. This software is freely available via SourceForge [2].

It provides several verification algorithms and offers an open interface for additional modules. The graphical front-end is currently implemented only as a Microsoft Visio add-on [2]. However, there is already a suggestion to integrate the SCStudio also with another application to enable its cross-platform usage.

To the basic SCStudio functionality belongs drawing MSC diagrams. On the drawn MSC diagram a graphical syntax verification can be executed. SCStudio provides also additional features like export and import of ITU-T Z.120 [1] textual format, export to LaTeXformat, executing of verification algorithm to detect deadlocks, cycles, race conditions etc.[1].

SCStudio has also a command line interface, which provides the functionality of import and export as well as verification features. In contrast with Microsoft Visio add-on, command line interface is cross platform. It allows to work with MSC diagrams in the textual format.

1. List of all available algorithms is published on SCStudio web page [2]

Chapter 3

Traffic capturing

Current computer networks are usually large and diverse systems communicating through a big range of protocols. For that reason arise the need for sophisticated software tools to monitor and troubleshoot network traffic called sniffers. Sniffer is programs that has the ability to intercept the networks traffic. It can be used to analyse network problems, monitor network usage, gather and report network statistics. The most important usage that make sniffers interesting for this thesis is debugging of client/server communication and network protocol implementation. One of the most popular sniffer applications is *Wireshark*. Functionality of all these tools is based on packet capturing, which is actually the action of collecting data as they travel over a network.

3.1 Packets capturing

In this section packet capturing over Ethernet-based networks is described. Every device in the network has its network card with MAC address. When the device receives an Ethernet frame, it checks the match of the destination address with its own. In case of conformity it generates an interrupt request and the network card driver handles the routine of frame processing. The driver timestamps received data and copies it from the card buffer to a block of memory in kernel space. There is only one difference by using a sniffer - the network driver also sends a copy of received or sent frame to a part of the kernel called packet filter, which makes packet capture possible [5].

3.2 Packets filtering

Every operating system has its own filtering mechanism. In fact, most of them are based on the same architecture - Berkeley Software Distribution Packet Filter (BPF) [6]. The BPF has two components : the network tap and the packet filter.

The network tap collects the copies of packets from the network device drivers and delivers them to applications. The filter decides of accepting the captured packet. A packet satisfying the filter is copied to the kernel buffer.

After packet arrives at a network interface the link level driver calls the listening BPF. If there is not any BPF interface, the packet is normally sent up only to system protocol stack. On the basis of the defined filter is the packet in BPF accepted and its data copied to the buffer associated with the filter. Since the time between the packets is usually only a few microseconds it is not possible to do a kernel call per packet and BPF collects the data from several packets and returns it the reading application as an unit. BPF encapsulates the captured data from each packet with a header that includes a timestamp, length, and offsets for data alignment [6]. This information could be saved to PCAP file format discussed in Section 3.3.

A packet filter is a simply boolean expression, which decides whether the captured packet is accepted or not. If the value is true, the packet is copied for the application by the kernel. Otherwise it is ignored. The packet filter is a powerful, low-level system primitive with high performance interface to limit the complexity of the interaction between user applications and the kernel [7].

3.3 Packets capture format

The problem of saving exchanging packet traces has been there for a long time. Unfortunately, no standard solutions exist for this task right now. One of the most accepted packet capture formats is the one defined by *Libpcap* (PCAP). The PCAP file format is the main capture file format used by Wireshark, TcpDump/WinDump, snort, Microsof Network Monitor and many other sniffers. As the *Libpcap* library became the "de facto" standard of network capturing on UN*X, it started to be considered as the common denominator for network capture files in the open source world [8]. The main requirement for PCAP format is portability. A a capture trace must contain all the information needed to read data independently from network, hardware and operating system of the machine that made the capture. The file extension for PCAP based files is ".pcap" or ".cap". PCAP file contains record of each packet captured from network by sniffing.

Important note: A captured packet in a PCAP capture file does not necessarily contain all the data in the packet as it appeared on the network. The capture file might contain at most the first N bytes of each packet, for some value of N. The value of N, in such a capture, is called the "snapshot length" or "snaplen" of the capture. N might be a value larger than the largest possible packet, to ensure that no packet in the capture is "sliced" short; a value of 65535 will typically be used in this case [8].

3.3.1 PCAP global header

The PCAP file has a global header containing general information about the capture. This header starts the file and it is followed by records holding information about every single captured packet. The global header contains fields *Byte-Order Magic*, *Major version*, *Minor version*, *Thiszone*, *Sigfigs*, *Snaplen* and *Network*.

Item *Byte-Order Magic* is used to detect the file format and the byte ordering. The application writes the hexadecimal number $0xa1b2c3d4$ with its native byte ordering into this field. The reading application can then distinguish the capture files saved on little-endian machines from the ones saved on big-endian machines.

Major version and *Minor version* attributes denote the version number of this file format. *Thiszone* stands for the time in seconds between GMT (UTC) and the local time zone of the following packet header timestamps.

Sigfigs was designated for accuracy of time stamps in the capture. *Snaplen* determines the *"snapshot length"* for the capture and *Network* identifies link-layer header type [3].

3.3.2 PCAP records of captured packets

Every record of captured packet consists from packet's header and carried data. The header contains items *Timestamp_sec*, *Timestamp_usec*, *Captured_len*, *Packet_len* and *Packet_data*.

Timestamp_sec is the date and time of packet capture. It contains value in seconds since January 1, 1970 00:00:00 GMT, also known as a UN*X time. *Timestamp_usec* are the microseconds as an offset to timestamp in seconds. *Captured_len* stands for the number of bytes of packet data actually captured and saved in the file. *Packet_len* is the length of the packet as it appeared on the network when it was captured. *Packet_data* represents the actual packet data as a data block without specific byte alignment [3].

3.4 Libpcap/WinPcap

Libpcap is an open source library that provides an API interface to network packet capture systems. It was created in 1994 by McCanne, Leres and Jacobson as a part of research project to investigate and improve TCP and Internet Gateway performance at University of California, Berkeley [5]. *Libpcap* is maintained by *Tcpdump group* nowadays.

The *Libpcap API* is designed to be used from `C` and `C++`. *Libpcap* was primarily developed for UN*X based system, however there is also a Windows version of this library - *WinPcap*. Both of them provide the same functionality and exported functions, but they use different linking mechanism and system libraries. *Libpcap* is in contrast with *WinPcap* still active developed and about twice a year a new version is released.

The main functionality of these PCAP libraries can be split into four basic features:

1. Obtaining the list of available network adapters and retrieving a various information about them, like the network interface and the list of network addresses
2. Sniff the packets online, using one of the network interface card drivers of the device
3. Save captured packets to disk in PCAP format or load them from PCAP file with interface similar to live capture
4. Create BPF packet filters using a high level language and apply them to the packets sniffing

SCStudio is being developed in `C++` on UN*X as well as on Windows system. Therefore we have used both libraries, i.e. *Libpcap* and *WinPcap*, in dependence on platform. Regarding the library versions, we decided not to use the up to date *Libpcap* version. To meet the same functionality on both systems we prefer the version, which *WinPcap* implementation is based on. The essential functionality meeting our requirements is loading packets from PCAP files and creating and applying the traffic filters.

The principle of reading packets from PCAP file is very intuitive. With library exported function *pcap_open_offline()* the pcap file is being opened for reading. The captured packets are being obtained with the callback-based function *pcap_loop(pcap_handler)*. The programmer's main task is to implement an own *pcap_handler()* function that deals with parsing information from packets. The information can then be used by the main algorithm of the program, e.g. transformation to MSC.

3.4.1 PCAP filter expressions

Applying a filter on traffic involves three steps. Setting a filter expression, compiling the expression into BPF program similar to assembly and finally applying the filter [5]. The great advantage of *Libpcap/WinPcap* libraries is high level language that enables to define the filter expression in much easier way.

There are some examples of filter expressions:

- *"src host 192.168.5.77"*: returns only the packets with source IP address 192.168.5.77
- *"dst port 80"*: returns packets whose TCP/UDP destination port is 80
- *"ip[8]==5"*: returns packets whose IP TTL (Time To Live) is 5
- *"icmp[icmptype] != icmp-echoreply"*: returns all ICMP packets that are not echo replies

The filters language offers the possibility of using logical and also bitwise operators. This way an expression E for filtering RTP (Real Time Protocol) traffic was defined as follows:

$$E = udp[1]\&1 \,!= 1 \;\&\&\; udp[3]\&1 \,!= 1 \;\&\&\; udp[8]\&0x80 \;==\; 0x80 \;\&\&\; length < 250$$

The expression is compiled into BPF program mentioned in Section 3.2 that the kernel can understand. For this purpose there is a function *pcap_compile()* in *Libpcap API*. Once we have a compiled BPF program we can insert it into the kernel calling function *pcap_setfilter()* and all the packets obtained by *pcap_loop()* callback must then satisfy the packet filter expression.

Chapter 4

Analysis of captured traffic

To generate MSC diagram it is essential to parse and analyse captured traffic, which is saved in PCAP file. Regarding the complexity and amount of information that user can be interested in, it is necessary to parse all important data and provide them in human-readable form.

As we have mentioned in Chapter 3, *Libpcap* can be used except traffic sniffing also for offline reading of PCAP files. However, this feature allows to obtain the captured data from packets only as clusters of bytes. The key task of parsing activity is to rationalize the packet's data in a way, that they can be consequently used for MSC creation.

We have intended to use beside *Libpcap* another `C++` library, which will provide the complex parsing functionality. That means it should support analysis of data carried by the most used network protocols. In spite of invested time and effort, we haven't found any that meets this requirement. Most of the libraries focus on the functionality of network communication and data transmission, not on separate parsing. For that reason the parsing feature is very poor in well-known network libraries like *Boost::Asio*.

There are tools suitable for this task, some of them even based on *Libpcap* library (*Pcap-DotNet*), but only in other programming languages. We have found only several small open-source `C++` projects (*libcrafter-0.1*) that are not intensively used by programmers community and their functionality is not verified as well. Therefore we have decided to implement an own parser that will meet the specific needs of the thesis.

4.1 Packets parsing

To analyse and parse data from packet it is necessary to get familiar with packets structure. An abstract form of packet's data encapsulation is to see in Figure 4.1. Currently, we support only parsing of traffic captured over *Ethernet*-based (*IEEE 802.3*) networks. Every Ethernet frame contains Ethernet header and payload. Payload consists usually from network layer header that is followed with transport layer protocol. Transport layer encapsulates an application data or transfers some control information.

Data link protocol header	Network protocol header	Transport protocol header	Application data

Figure 4.1: Packet structure

The task a developer has to deal with is to rationalize the data from captured frame according to ISO/OSI model [9]. Thus from the lowest to the upper level, where the lowest level represents the data link layer and the upper one stands for application layer. To achieve this goal it is essential to implement the support for parsing of most used network and transport protocols. Only this way the data encapsulated in them can be identified and analysed. The structure of individual protocols is precisely described in RFC (Request for comments) documents marked as STD (Internet standard). In the parsing process, the identity of network protocol is detected from Ethernet header. The overlaying transport protocol is determined by information from network protocol header. To identify the application protocol the port numbers from underlying transport protocol are used.

4.2 Binary protocols

Binary protocols are used widely in data link, network and transport layer. Their basic advantage is terseness and space efficiency, which reflects in transmission and interpretation speed. Protocol's header structure defines which information stand the header's fields for. The data carried in protocol's header are then interpreted according to defined header format. For example, in Figure 4.2 there is an TCP header format definition from RFC document [10] (one tick mark represents one bit position).

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|          Source Port          |       Destination Port        |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                        Sequence Number                        |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                    Acknowledgment Number                      |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|  Data |           |U|A|P|R|S|F|                               |
| Offset| Reserved  |R|C|S|S|Y|I|            Window             |
|       |           |G|K|H|T|N|N|                               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|           Checksum            |         Urgent Pointer        |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                    Options                    |    Padding    |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                             data                              |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Figure 4.2: TCP Header Format [10]

Regarding the size of separate fields in the header, the structure representing the protocol's header can be declared. In Figure 4.3, there is a C++ structure of TCP header to see. The packet's data could be consequently assigned to the structure and interpreted. That is actually the main principle how the information from binary protocols is parsed.

```
struct TcpHeader
{
      u_short sport; // Source port (16 bits)
      u_short dport; // Destination port (16 bits)
      u_int32 seq_num; // Sequence number (32 bits)
      u_int32 ack_num; // Acknowledgment (32 bits)

#if HOST_IS_BIG_ENDIAN
      u_char data_off:4; // Data offset
      u_char reserved:4; // Reserved
#else
      u_char reserved:4; // Reserved
      u_char data_off:4; // Data offset = size of header (in 32-bits words)
#endif
      /* Flags */
      u_char fin :1;  // Finish Flag
      u_char syn :1;  // Synchronise Flag
      u_char rst :1;  // Reset Flag
      u_char psh :1;  // Push Flag
      u_char ack :1;  // Acknowledgement Flag
      u_char urg :1;  // Urgent Flag
      u_char ecn :1;  // ECN-Echo Flag
      u_char cwr :1;  // Congestion Window Reduced Flag

      u_short window;  // Flowing window (16 bits)
      u_short crc;  // Checksum (16 bits)
      u_short urg_point; // Urgent pointer (16 bits)
};
```

Figure 4.3: TCP Header declaration

4.3 Text protocols

Text protocols are in contrast with previous mentioned protocols structured often as a plain ASCII text. The information in headers is identified by protocol's specific keywords. Therefore the header structure is not fixed and the fields format is more variable. For that reason the data can't be interpreted according to field's position but a text parser has to be used. From text protocols we support parsing of *HTTP* (hypertext transfer protocol) and *SIP* (session initialization protocol) in our implementation.

The functionality of *HTTP* packets parsing is provided via *HTTP Parser* library [11]. *SIP* packets are being parsed with functionality of *The GNU oSIP library* [12].

4.4 Implementation of parser

To interpret the header's fields properly, it is necessary to check the endianity of machine where the sniffing was performed. This information can be obtained from the PCAP file header via *Byte-Order-Magic* property. It is also important to determine the endianity of machine, where the parsing is executed.

Network byte-order is usually Big-Endian, but the host-order can be also Little-Endian.

In this case, the captured bytes have to be translated into host byte order. For that purpose, there are *ntohs* and *ntohl* system functions. The first one converts a *u_short* data type, the second one *u_long* type to host byte order. However, if the parameter is already in right order, the function will reverse it. For that reason, the endianity of the host-machine has to be checked before.

The most important information to obtain is the actual operating system (Windows, Linux, FreeBSD, OpenBSD, MacOSX etc.) and consequently, the information about the host byte order. The order-swapping functions are defined in platform-dependent system libraries. For example, in ⟨*Winsock2.h*⟩ for Windows, ⟨*arpa/inet.h*⟩ for some UNIX versions and ⟨*netinet/in.h*⟩ for Linux. The demonstration of parsing a packet, which carries HTTP data, is described in Function 4.4 $parsePacket$.

To parse a protocol, its structure and behaviour has to be studied. For that reason we have implemented only the protocols, which are used in network traffic more frequently. From data link layer *Ethernet 802.3*, from network layer *Ipv4*, *Ipv6*, *ARP* (Address resolution protocol), and from transport layer *TCP* (Transmission control protocol), *UDP* (User datagram protocol), *ICMP* (Internet control message protocol) and *DNS* (Domain name system protocol). If there is not support for specific protocol, the only information that can be interpreted is protocol's identification number or specific name and data size.

Note: Regarding the modular structure of our parser implementation, the support for another protocols parsing could be easily added. The extended parsing functionality must overload methods from class *Packet* and consequently it can be included in project structure.

4.5 Comparison with Wireshark

The are various packet analysers and sniffers. One of the most popular tools is *Wireshark*, which is being developed since 1997 and is also based on *Libpcap* functionality. It is an free and open-source application with a big base of developers and maintainers. *Wireshark* provides a very intuitive view on the captured communication, with support of parsing a great variety of protocols. The support for import of PCAP files is matter of course.

From those reasons we have used *Wireshark* to verify the correctness of parser's implementation. However, it was not our intention to achieve the functionality provided with *Wireshark*. Its user interface was designed exactly to represent the packet's data and related information. In contrast with MSC formalism, which is aimed more on the packet sequence and ordering. *Wireshark* provides also the feature of a simple flow graph creation. However, there is not possible to set the OSI layer, from which the information should be displayed. It doesn't enable to join more related packets into one message in graph, choose the time unit etc. However the most important advantage of importing PCAP files in SCStudio are the verification algorithms, which can be executed over generated MSC diagram and portability of MSC textual format.

Function 4.4 *parsePacket*(pcap_pkthdr* *packet_header*,u_char* *packet_data*)

```
#if HOST_IS_BIG_ENDIAN
#define TO_NTOHS(data) (data)
#define TO_NTOHL(data) (data)
#else
#define TO_NTOHS(data) (ntohs(data))
#define TO_NTOHL(data) (ntohl(data))
#endif
time_t timestamp ← packet_header →get_time()
// retrieve the position of IP header,Ethernet header size = 14
IpHeader* ip_header ←(IpHeader*)(packet_data + 14)
string source_address ← ip_header →get_source_address()
string destination_address ← ip_header →get_destination_address()
// retrieve the TCP header position
int ip_length ← ip_header →get_data_offset()
TcpHeader* tcp_header ←(TcpHeader*)((u_char*)ip_header + ip_length)
// get TCP ports
u_short source_port ← TO_NTOHS(tcp_header → sport)
u_short destination_port ← TO_NTOHS(tcp_header → dport)
// get Acknowledgment and Sequence number
u_int32 sequence_number ← TO_NTOHL(tcp_header → seq_num)
u_int32 acknowledgment_number ← TO_NTOHL(tcp_header → ack_num)
// retrieve HTTP header position
int tcp_length ← tcp_header → data_off * 4
u_char* http_data ←(u_char*)tcp_header + tcp_length
// use HTTP_Parser library exported functions to parse data
```

Chapter 5

Transformation to MSC

The transformation and visualization of captured traffic is intuitive. The communicating devices are represented with *instances* named by their network addresses. Packet is transformed in *messages* and its data described in *message labels*. The ordering of messages reflexes the traffic trace, which is approximated also with information from packets timestamps. The principle of transformation is described in Algorithm 5.1

5.1 Communicating devices

Every packet captured from Ethernet-based network contains MAC source and destination address. If the network layer protocol is *IPv4* or *IPv6*, then there are also IP addresses to parse. After the transformation, every device that communication was sniffed is represented through a single *instance* with specific name in MSC.

By default way of our implementation, the instances are named by devices' IP addresses. In case there is not IP header in captured packet, MAC address is used. This behaviour can be changed via settings (e.g. a user wants to see only information from data link layer, then MAC address is default name).

5.2 Packets

Packets from PCAP are in the MSC diagram displayed as *messages*. Regarding the packets sequence in PCAP file and their timestamps, the whole captured communication is strictly ordered. For that reason, there is only one *strict order area* on every *instance*, which contains the corresponding *events*. We have implemented two ways how the packets could be represented in MSC. Either one packet stands for one *message* or more packets are aggregated in a single *message*. The second approach is discussed in chapters 6 and 7.

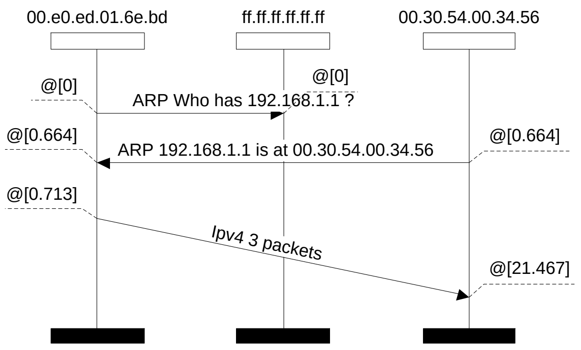
The carried data are described via *message label*. The information from which the label's text consists depends on the user-defined settings and encapsulated protocols. The protocol-specific information can be obtained only from protocols supported in our implementation. Otherwise the *message label* contains only identification of carried protocols (name or number) and packet size.

5.3 Timestamps

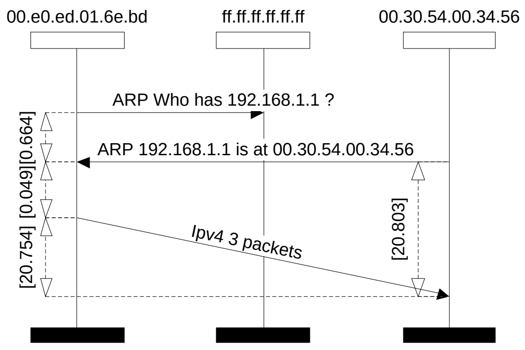
Timestamp stands for the information, when the single packet was sent or received by listening network driver. Because of the missing information, when the packet was sent or delivered to other device, the transmission duration is abstracted in our implementation. The time of first packet obtained from PCAP file is redefined as zero and the following timestamps are calculated according to its value. It is possible to define the time unit that will be used in MSC: second, millisecond or microsecond.

In the MSC diagram generated from PCAP file, timestamps could be represented via *absolute times* or *time intervals*. The meaning of *absolute times* (Figure 5.1a) is quite clear. Every *absolute time* is gained from packet as a timestamp. If there are not more packets aggregated in one *message*, the timestamps attached to its head and tail are equal. Otherwise the *absolute time* attached to *message's* tail represents the timestamp of first aggregated packet. The second *absolute time* connected to *message's* head presents the timestamp of last aggregated packet.

The MSC with *time intervals* (Figure 5.1b) is just a modification based on the *absolute times* implementation. Every *time interval* consist from one value calculated as a difference of the related *absolute times*. To every *message* a time constraint is attached, which denotes the time between consecutive *messages* in MSC diagram. The *time interval* is created between every consecutive *events* on the same instance. But this solution is not perfect and should be improved in the future, because there are some redundant *time intervals* added to MSC. For that reason the efficiency of algorithms checking the time consistency is degraded. However, in our implementation the MSC is generated concurrently with obtaining of packets. Therefore the interval's redundancy can't be checked in moment of its creation. The duration of transmission that includes more packets, which are joined together to one MSC *message*, is represented via *time interval* between *message's* events.



(a) Absolute times



(b) Time intervals

Figure 5.1: MSC time formats

Algorithm 5.1: Generating MSC from PCAP

```
input  : A $pcap$ file
output: A BMsc $bmsc$
1  load from settings: $enum\ m\_layer\_to\_parse$
2  load from settings: $enum\ m\_time\_format$
3  load from settings: $string\ m\_filter\_expression$
4  load from settings: $label\_settings$
5  $map\langle string, Instance\rangle\ instances$
6  $pcap\_t * adhandle \leftarrow pcap\_open\_offline(''file.pcap'')$
7  $pcap\_compile(adhandle, m\_filter\_expression)$
8  $pcap\_setfilter(adhandle)$
9  BMsc $bmsc \leftarrow$ new $Bmsc()$
10 $pcap\_loop(adhandle, handler)$
11 foreach $packet$ obtained from $pcap\_loop()$ as callback do
12     double $time \leftarrow packet$.get_timestamp()
13     char* $packet\_data \leftarrow packet$.data_part
14     Protocol $protocol \leftarrow$Ethernet
15     ParseLayer $current\_layer \leftarrow$ DataLinkLayer
16     while ($current\_layer$ != $m\_layer\_to\_parse$ && $packet$.size $\rangle 0$) do
17         ProtocolHeader $proto\_h \leftarrow$ new $ProtocolHeader(protocol, packet.data\_part)$
18         if ($current\_layer$ = NetworkLayer || DataLinkLayer) then
19             string $source\_address \leftarrow proto\_h$.get_source_address()
20             string $destination\_address \leftarrow proto\_h$.get_destin_address()
21         $protocol \leftarrow proto\_h$.get_next_protocol()
22         $packet$.data_part $\leftarrow packet$.data_part + $sizeOf(proto\_h)$
23         $packet$.size $\leftarrow packet$.size - $sizeOf(proto\_h)$
24         $current\_layer + +$
25     string $label \leftarrow get\_label(proto\_h, label\_settings)$
26     Instance $source\_instance \leftarrow instances$.find($source\_address$) OR
       $bmsc$.add_Instance($source\_address$)
27     Instance $destination\_instance \leftarrow instances$.find($destination\_address$) OR
       $bmsc$.add_Instance($destination\_address$)
28     Event $source\_event \leftarrow source\_instance$.get_strict_order_area().$add\_event()$
29     $source\_event \leftarrow add\_time(time, time\_format)$
30     Event $destination\_event \leftarrow destination\_instance$.get_strict_order_area().$add\_event()$
31     $destination\_event \leftarrow add\_time(time, time\_format)$
32     Message $message \leftarrow$ new $Message(label)$
33     $message$.glue_events($source\_event, destination\_event$)
```

Chapter 6

Packet reassembling

In data communications network the transmitted data are often split in more fragments. Either because of over ranging the maximum transmission unit supported by underlying protocol or on account of network unreliability. For that reason it is an appropriate measure to defragment the data into more single packets and so improve the probability of correct delivery to the target device. The segments are then reassembled according to protocol implementation, delivered and processed to the application.

The main purpose to implement reassembling feature in this thesis is to make the generated MSC more transparent and well-arranged.

6.1 TCP segments reassembling

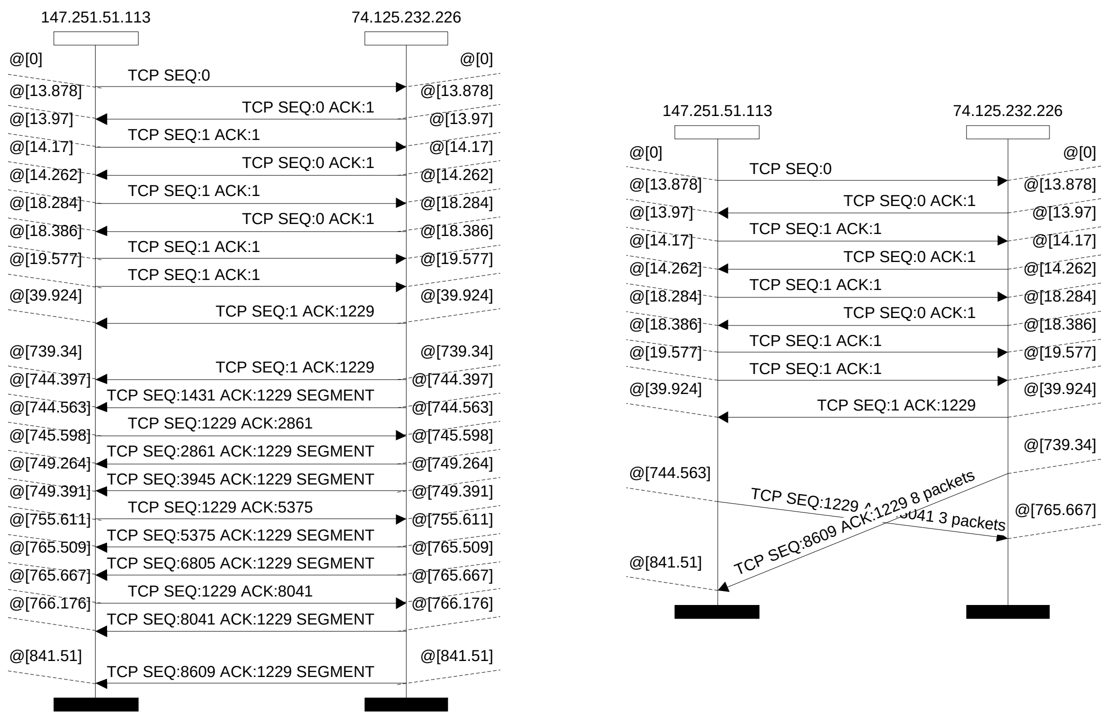


Figure 6.1: TCP reassembling

The functionality implemented by us is based on sequence (SEQ) and acknowledgment (ACK) numbers obtained from TCP headers. This numbers are designed for ARQ (Automatic Repeat-reQuest) and QoS (Quality of service) mechanisms. However, this approach is not used by every application communicating via TCP and it is quite hard to handle the wide-spectrum of special scenarios related to this issue. Therefore we have implemented the reassembling feature just for scenario, where the one side is transmitting and the receiving side is acknowledging the received TCP segments. The acknowledgment packets in this case must not transfer any data, only the control information.

The reassembled packets can by interpreted as a single *message* in MSC. There is also another functionality implemented - *message label* of TCP packet, which is a segment of previous data, contains text "SEGMENT". An example of TCP reassembling is to see in Figure 6.1.

6.2 HTTP fragments reassembling

HTTP works as request-response protocol in client-server model. The request and response messages are often split in more fragments. Data carried by single packet can then consist from header fields, or also from HTTP payload. Our HTTP reassembling feature allows to represent the split fragments with just one request or response message in MSC. It is also possible to provide information about message's sections transferred in single packets. If the packet carries only header fields, there is information "HEADERS_PART" or "HEADERS_COMPLETE" to see in the *message label*. In case that packet contains also payload, label contains text "DATA_PART" or "DATA_COMPLETE" (if all data were delivered). One packet can transfer also a complete HTTP message and this information is expressed with text "ALL_DATA". In Figure 6.2 the MSC diagrams generated with this functionality are displayed.

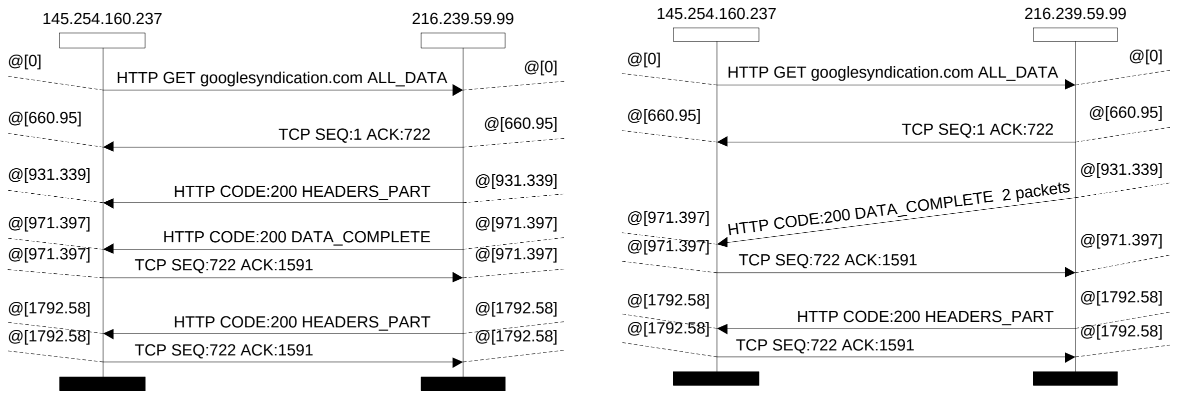

145.254.160.237
216.239.59.99
@[0]
HTTP GET googlesyndication.com ALL_DATA
@[660.95]
TCP SEQ:1 ACK:722
@[931.339]
HTTP CODE:200 HEADERS_PART
@[971.397]
HTTP CODE:200 DATA_COMPLETE
TCP SEQ:722 ACK:1591
@[1792.58]
HTTP CODE:200 DATA_COMPLETE 2 packets

Figure 6.2: HTTP reassembling

6.3 RTP/SIP reassembling

RTP (Real time transport protocol) is mostly used to transfer data stream between multimedia applications. For example for transmission of acoustic and visual streams in video conference sessions, carrying uncompressed video or MPEG formats. Because of the requirements on timeliness, RTP is running over UDP protocol.

SIP (Session initialization protocol) initializes the session between communicating endpoints. It employs design elements similar to HTTP request-response transaction model, but the messages transfer only control information. SIP is usually used to set up and control communication channels for data stream transmission via RTP protocol.

Our idea of RTP/SIP reassembling implementation is to aggregate the RTP packets sent in streams, which are initialized and handled with SIP protocol. The problem is, that the connection can be established via proxy server. Therefore we join also RTP packets, which were sent to another device, if they were captured between SIP messages. This feature's output is represented in Figure 6.3.

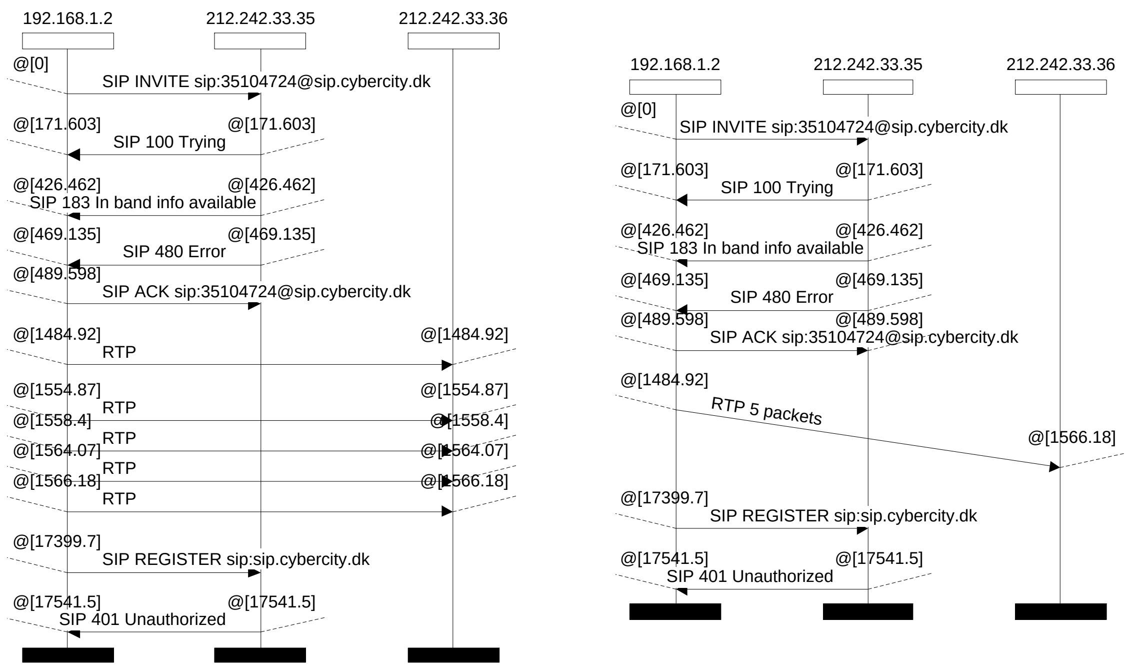


Figure 6.3: RTP/SIP reassembling

Chapter 7

Packet aggregation

The number of packets saved in PCAP file is not restricted. For example, on a 100-Mb/s Ethernet Link with minimum frame size 84 bytes, there can be 148 809 frames per one second captured. For that reason a requirement for some way of abstraction has arisen to keep the generated MSC readable, also in case that a bigger PCAP file is imported.

A suitable approach is to represent more packets with just one *message* in generated diagram. Our implementation is based on the idea, that we can use some kind of "aggregation" specified by user-defined parameters on the captured communication. This feature helps to provide a more simple and compact view on the entire communication. Such as which protocols are used, where is the specific scenario a user is interested in, how long the continuous communication between selected network devices takes etc. This view gives also more simple way to get some statistical information about the captured traffic (e.g. the count of packets transmitted between devices).

7.1 Packets identification

Before we join some packets together, they should be identified at first. The captured packet $\mathcal{P}$, according to carried data, is defined in Definition 1.

Definition 1. *Packet $\mathcal{P}$ is defined as a tuple $(\mathcal{SD}, \mathcal{N}, \mathcal{T}, \mathcal{A}, \mathcal{PS})$ where*

- *$\mathcal{SD}$ stands for source and destination address;*
- *$\mathcal{N}$ is network protocol (ARP, IP etc.);*
- *$\mathcal{T}$ represents transport protocol (TCP, UDP, ICMP etc.);*
- *$\mathcal{A}$ is application protocol (HTTP, SIP etc.);*
- *$\mathcal{PS}$ are source and destination ports identified for each address and protocol with 16 bit number (e.g. port 80 for HTTP)*

Note*: Not every captured packet specifies all items defined in tuple. For example an ICMP packet doesn't carry any application data (items $\mathcal{A}$ and $\mathcal{PS}$ are empty).*

Our aggregation feature is based on joining more packets in one MSC message. To preserve the identity of joined packets they have to be equivalent in characteristic attributes. For packets comparison the theorem from Definition 2 is used.

Definition 2. *Two* packets $\mathcal{P}_1$ *and* $\mathcal{P}_2$ *are equivalent* ($\mathcal{P}_1 \equiv \mathcal{P}_2$) *iff they match on the attributes from subset* $s \subseteq \{\mathcal{SD}, \mathcal{N}, \mathcal{T}, \mathcal{A}, \mathcal{PS}\}$. *The equivalency relation* $\equiv$ *on packets is transitive, reflexive and symmetric.*

The key attributes for packets comparing are defined as tuples in set $\mathcal{S}$ presented in Definition 3. Every tuple defines a specific level on which the packets are being compared. For example tuple $(\mathcal{SD})$ interprets the packets as equivalent if they have the same source and destination address. In this case, there are usually many packets that can be joined together. But in case that packets are characterized with tuple $(\mathcal{SD}, \mathcal{N}, \mathcal{PS})$ they must match also in network protocol and ports. The aggregation is then more refined and the user can focus on packets exchanged on different ports.

Definition 3. $(\mathcal{S}, \prec)$ *is a totally ordered set where* $\mathcal{S} = \{(\mathcal{SD}), (\mathcal{SD}, \mathcal{N}), (\mathcal{SD}, \mathcal{N}, \mathcal{T}), (\mathcal{SD}, \mathcal{N}, \mathcal{A}), (SD, N, PS)\}$ *and* $(\mathcal{SD}) \prec (\mathcal{SD}, \mathcal{N}) \prec (\mathcal{SD}, \mathcal{N}, \mathcal{T}) \prec (\mathcal{SD}, \mathcal{N}, \mathcal{A}) \prec (\mathcal{SD}, \mathcal{N}, \mathcal{PS})$. *Every tuple contains specific packet attributes from Defition 1. Ordering* $\prec$ *is related to strictness of every single tuple. That means if* $\nu \prec \mu$ *for* $\nu, \mu \in \mathcal{S}$ *and packets* $\mathcal{P}_1$, $\mathcal{P}_2$ *are equivalent in* μ, *then they are equivalent also in* ν.

If packet $\mathcal{P}$ doesn't contain all items from tuple μ, then $\mathcal{P}$ is characterized with less specific tuple $\nu \prec \mu$. These packets do not have to be displayed if a user is interested only in packets with selected subset of attributes $\mu \in \mathcal{S}$.

7.2 Communication model

The network communication is described with generated MSC. The MSC's Definition 4 was taken and edited from [13] and [14].

Definition 4. *A strictly ordered* MSC *is defined as a tuple* $(\mathcal{D}, \mathcal{C}, E, <, \tau, P, \mathcal{M})$ *where*

- $\mathcal{D}$ *is a finite set of* communicating devices;
- $\mathcal{C}$ *is a finite set of* communication channels, $\mathcal{C} = \{(x, y) \ : \ \{x, y\} \in \mathcal{D} \wedge x \neq y\}$;
- E *is a set of* events;
- $<$ *is a total ordering on* E *called* visual order;
- $\tau : E \to \{s, r\}$ *is a labeling function dividing events into* send, receive;
- $P : E \to \mathcal{D}$ *is a mapping that associates each event with a device;*
- $\mathcal{M} \subseteq (\tau^{-1}(s) \times \tau^{-1}(r))$ *is a bijective mapping relating every send with an unique receive;*

Visual order $<$ *is defined as the reflexive and transitive closure of* $\mathcal{M} \cup \bigcup_{d \in \mathcal{D}} <_d$ *where* $<_d$ *is a total order on* $P^{-1}(d)$.

In other words the devices are communicating via messages from $\mathcal{M}$. Every message $m \in \mathcal{M}$ contains send and receive event. Every event $e \in E$ is associated with different device $\mathcal{P}(e)$. The events are ordered from top to bottom on a device $d \in \mathcal{D}$ using $<_d$. Between devices the events are ordered in direction of message, i.e. send event is always before receive event.

7.3 Aggregation types

Aggregation $\mathcal{G}$ is characterized in Definition 5.

Definition 5. *Aggregation $\mathcal{G}$ is defined as surjective mapping $\mathcal{G}\colon \mathcal{P} \to m$ where*

- *$\mathcal{P}$ is a single captured packet*
- *m represents a MSC message : $m \in \mathcal{M}$.*

The ordering of events $s \in E$ and $r \in E$ is defined as follows:

- *If only one packet $\mathcal{P}_i$ is mapped to message m_i then send event s is followed immediately by receive event r.*
- *In case that packets $\mathcal{P}_i..\mathcal{P}_n$ are mapped to message m_i then there can be events ordered between s and r. For example in* Figure 7.3 *events of RTP message* 4 *are between send and receive events related to SIP message* 3.

We have implemented five aggregation types. A user can select one that meets his requirements and apply it on captured traffic. The samples of MSC diagrams in this chapter were generated from SIP and RTP communication characterized with tuple $(\mathcal{SD}, \mathcal{N}, \mathcal{A})$.

7.3.1 Successive packets in entire communication

This Aggregation defined as $\mathcal{G}_s$ is the simplest and most intuitive aggregation type, which makes the generated MSC more transparent with only little effect on original packet sequence. If successive packets are equivalent, then they are mapped to the same message. For example $packets$ 2 and 3 from Figure 7.1 are mapped to single message with number 2 in Figure 7.2. The aggregating function $\mathcal{G}_s$ is defined in Definition 6:

Definition 6. *$\forall$ packets $\mathcal{P}_k, \mathcal{P}_l\colon \mathcal{G}_s(\mathcal{P}_k) = \mathcal{G}_s(\mathcal{P}_l)$ iff*

- *$\mathcal{P}_k \equiv \mathcal{P}_l$*
- *$\nexists \mathcal{P}_i\colon \mathcal{P}_i \not\equiv \mathcal{P}_k$ and $\mathcal{P}_i$ was sent between packets $\mathcal{P}_k$ and $\mathcal{P}_l$*

7.3.2 One-way communication in channels

This type of aggregation denoted as $\mathcal{G}_o$ is focused on the communication running in the specific channels of network communication.

Note : In this case the communication is split in separate channels. Therefore the packet's source and destination address is defined by channel, in which the packet was sent. By packets comparing, item $\mathcal{SD}$ is ignored, i.e. two packets could be evaluated as equivalent also in case that they were sent in different channels. This approach is applied also in Section 7.3.3 .

Imagine that the network communication runs only between two devices $d \in D$ and $g \in \mathcal{D}$. That are actually the packets sent in communication channels $\{(d, g), (g, d)\} \in \mathcal{C}$. The successive equivalent packets sent in communication channel $c = (d, g)$ are mapped to the same message m. The aggregation is interrupted if an equivalent packet in channel $c^{-1} = (g, d)$ is captured. This procedure is applied individually to all communication channels. You can see, that $packets$ 4 and 9 from Figure 7.1 were mapped to message 3 in Figure 7.3. In this type of aggregation the RTP communication in other channel wasn't taken in account. The aggregation $\mathcal{G}_o$ is specified in Definition 7

Definition 7. *$\forall$ packets $\mathcal{P}_k, \mathcal{P}_l$: $\mathcal{G}_o(\mathcal{P}_k) = \mathcal{G}_o(\mathcal{P}_l)$ iff*

- $\mathcal{P}_k \equiv \mathcal{P}_l$
- *$\mathcal{P}_k$ as well as $\mathcal{P}_l$ were sent in c, $c \in \mathcal{C}$*
- *$\nexists \mathcal{P}_i$: $\mathcal{P}_i \equiv \mathcal{P}_k$ and $\mathcal{P}_i$ was sent in c^{-1} and $\mathcal{P}_i$ was captured between packets $\mathcal{P}_k$ and $\mathcal{P}_l$*

7.3.3 Continuous both-way communication in channels

This aggregation denoted as $\mathcal{G}_b$ is aimed at continuous exchange of equivalent packets between two communicating devices. While they are communicating to each other, without interaction with another device via equivalent packet, the exchanged packets are being aggregated together.

The continuous communication between devices $d, g \in \mathcal{D}$ is related to two channels: $c = (d, g)$ and $c^{-1} = (g, d)$. While the packets sent in c and c^{-1} are equivalent, they are being mapped to the same messages m_c and $m_{c^{-1}}$. The aggregation is interrupted if an equivalent packet between one of devices d, g and another device $x \in \mathcal{D}$ is captured. In Figure 7.4 the aggregation wasn't interrupted at all. The devices 192.168.1.2 and 212.242.33.35 were continuously communicating via SIP protocol. During this communication 192.168.1.2 started sending RTP packets to device 212.242.33.36. However, this packets weren't equivalent to SIP. For that reason the last two SIP packets from Figure 7.1 were also aggregated to messages 1 and 2 in Figure 7.4. The mapping function $\mathcal{G}_b$ is defined in Definition 8.

Definition 8. *$\forall$ packets $\mathcal{P}_k, \mathcal{P}_l$: $\mathcal{G}_b(\mathcal{P}_k) = \mathcal{G}_b(\mathcal{P}_l)$ iff*

- $\mathcal{P}_k \equiv \mathcal{P}_l$
- *$\mathcal{P}_k$ as well as $\mathcal{P}_l$ were sent in $c = (d, g)$, $c \in \mathcal{C}$*
- *$\nexists \mathcal{P}_i$: $\mathcal{P}_i \equiv \mathcal{P}_k$ and $\mathcal{P}_i$ was sent in one of channels $\{(d, x), (x, d), (g, x), (x, g)\}$ where $x \in \mathcal{D}$ and $x \neq d \land x \neq g$, $\mathcal{P}_i$ was captured between packets $\mathcal{P}_k$ and $\mathcal{P}_l$*

7.3.4 Communication along whole captured traffic

In this kind of aggregation $\mathcal{G}_w$ the information about message ordering is a bit degraded at the expense of statistical view on the captured communication. All equivalent packets from the whole captured traffic are mapped to the same message. Accidentally in this case the output from this aggregation is equal to MSC in Figure 7.4. The aggregation $\mathcal{G}_w$ is defined in Definition 9.

Definition 9. $\forall$ *packets* $\mathcal{P}_k, \mathcal{P}_l$: $\mathcal{G}_w(\mathcal{P}_k) = \mathcal{G}_w(\mathcal{P}_l)$ *iff*

- $\mathcal{P}_k \equiv \mathcal{P}_l$

7.3.5 Communication in time intervals

This type of aggregation $\mathcal{G}_t$ is a refinement of aggregation $\mathcal{G}_w$ (Definition 9). A user can define a specific time i (e.g. 100 ms) by which the network traffic is segmented in set of $intervals\ \mathcal{I} = \{[0,i),[i,2i),[2i,3i),..[t-ix,t)\}$ where t is duration of captured communication and $x = \lfloor t/i \rfloor$. The equivalent packets in the same interval are mapped to the same message. For example the $packets$ 7 and 8 from Figure 7.1 are mapped to message 7 in Figure 7.5 because they are equivalent and captured in time interval $[1.600, 1.700)$. The aggregation $\mathcal{G}_t$ is defined in Definition 10.

Definition 10. $\forall$ *packets* $\mathcal{P}_k, \mathcal{P}_l$: $\mathcal{G}_t(\mathcal{P}_k) = \mathcal{G}_t(\mathcal{P}_l)$ *iff*

- $\mathcal{P}_k \equiv \mathcal{P}_l$
- $\mathcal{P}_k$ *as well as* $\mathcal{P}_l$ *were sent in the same interval* $i \in \mathcal{I}$

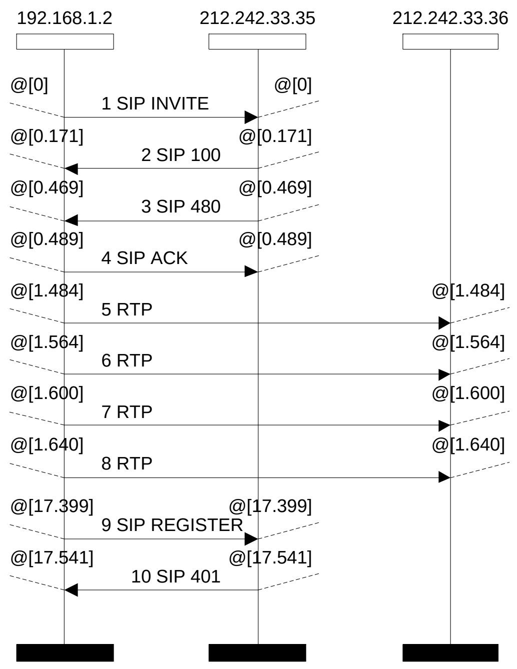


Figure 7.1: MSC without aggregations

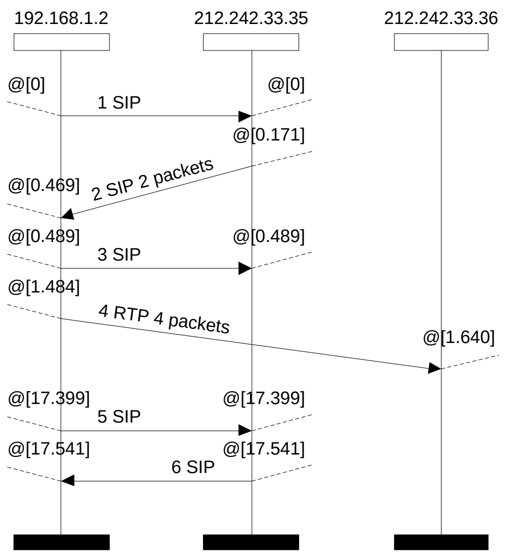


Figure 7.2: Succesive packets

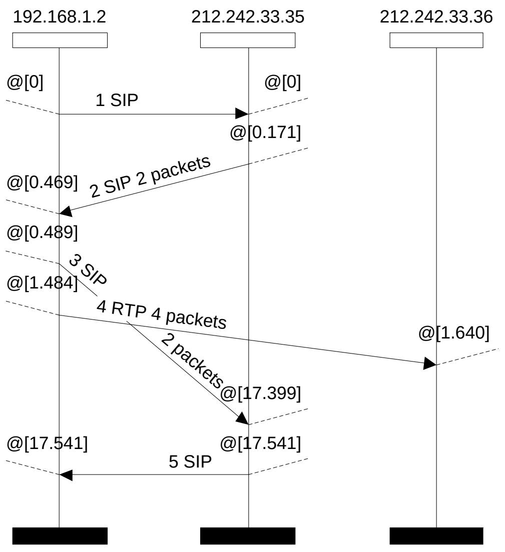
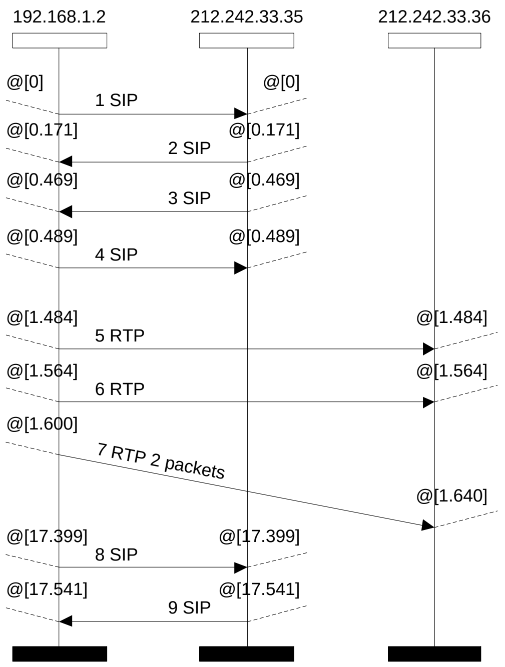


Figure 7.3: One-way communication in channels



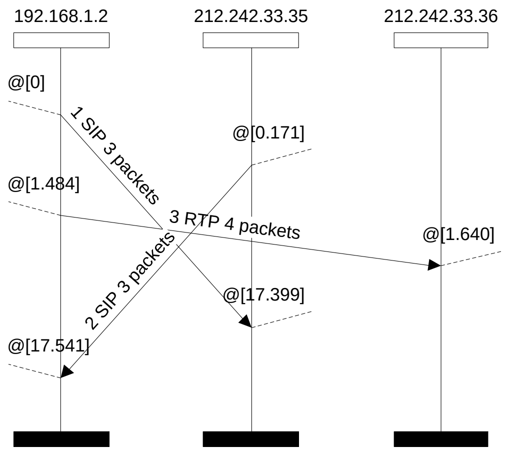


Figure 7.4: Continuous both-way communication in channels

Figure 7.5: Communication in time intervals 100 ms

Chapter 8

User interface

Regarding the amount of options related to implemented PCAP import functionality, it was important to provide a suitable interface for configuration of transformation parameters. This intention was achieved with comprehensible and user-friendly designed GUI, which allows to make use of all PCAP import related features. Therefore the design was carefully considered and consulted with SCStudio developers team. To help users understand the meaning of individual settings, the SCStudio *Help* section was extended with PCAP section.

8.1 Graphical user interface

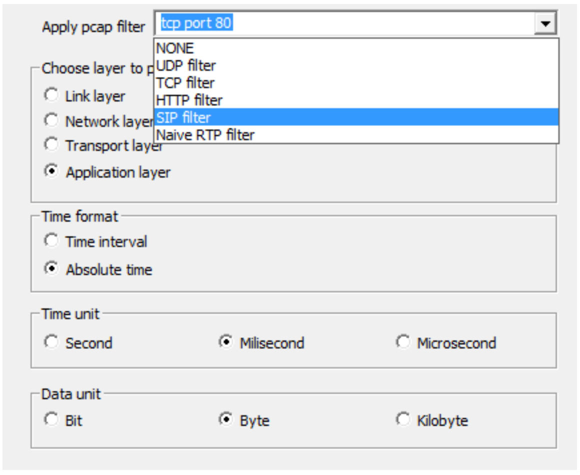
The graphical interface of SCStudio add-on sits above Windows Template Library (WTL) [15]. WTL is a set of `C++` templates, which focus purely on higher-level windowing functionality. The concept of dynamic data exchange allows the transfer of values in both directions between on-screen user interface controls and `C++` data members. WTL is based on ATL (Active Template Library) and Win32/Win64 API. Therefore it is small, ultra fast, non-intrusive and covers the needed UI concepts.

8.1.1 Selecting filter, Layer, Time format, Data units

The first dialog window is called *General Settings* (Figure 8.1). The combobox above allows to select a pre-defined filter or write another one, which will be applied on captured traffic. Radio buttons bellows provide selection of the layer in ISO/OSI model, from which the packet information should be presented in MSC *message labels*. There is also groupbox to select timestamp format - *absolute time* or *time interval*. Another options to set are the data units for time and data size.

8.1.2 Specific options related to MSC creation

The second dialog window called *View Settings* (Figure 8.2) contains specific options for MSC transformation. It allows the user to choose the amount of information presented in *message labels* according to spectrum of packet's layers, and also to view some additional information like packet number or size of carried data. The user can also choose if the packet, which doesn't contain required data (selected by ISO/OSI layer in *General Settings*), should be ignored or added to generated MSC.



Apply pcap filter: tcp port 80

NONE
UDP filter
TCP filter
HTTP filter
SIP filter
Naive RTP filter

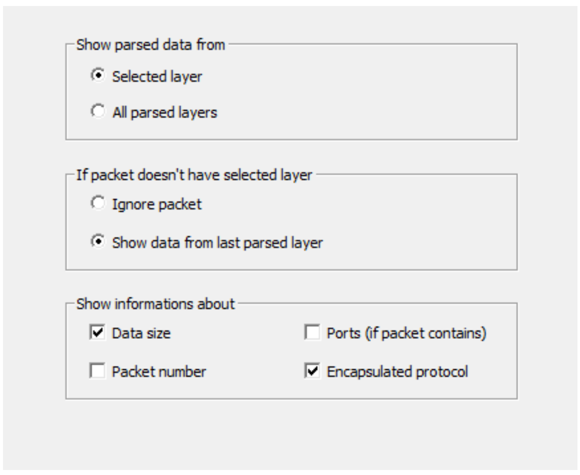
Choose layer to p
Link layer
Network layer
Transport layer
[✓ Application layer]

Time format
Time interval
[✓ Absolute time]

Time unit
Second [✓ Milisecond] Microsecond

Data unit
Bit [✓ Byte] Kilobyte

Figure 8.1: General settings

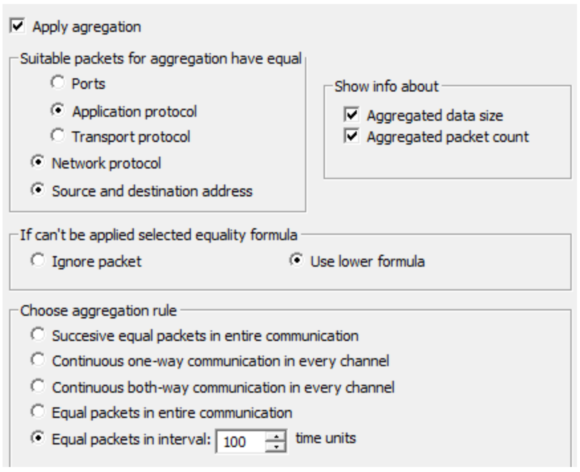


Show parsed data from
[✓ Selected layer]
All parsed layers

If packet doesn't have selected layer
Ignore packet
[✓ Show data from last parsed layer]

Show informations about
[✓ Data size] Ports (if packet contains)
Packet number [✓ Encapsulated protocol]

Figure 8.2: View settings



[✓ Apply agregation]

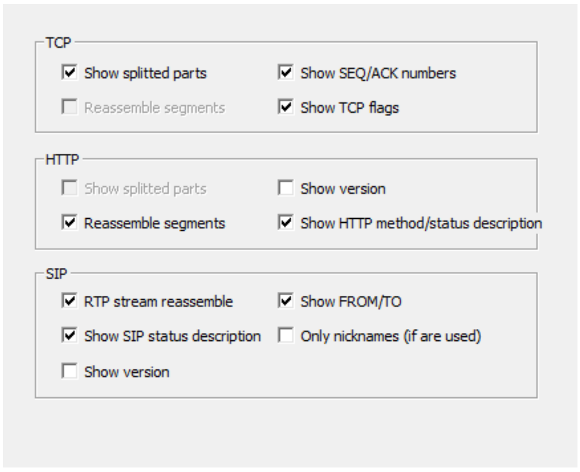
Suitable packets for aggregation have equal
Ports
[✓ Application protocol]
Transport protocol
[✓ Network protocol]
[✓ Source and destination address]

Show info about
[✓ Aggregated data size]
[✓ Aggregated packet count]

If can't be applied selected equality formula
Ignore packet [✓ Use lower formula]

Choose aggregation rule
Succesive equal packets in entire communication
Continuous one-way communication in every channel
Continuous both-way communication in every channel
Equal packets in entire communication
[✓ Equal packets in interval:] 100 time units

Figure 8.3: Aggregation settings



TCP
[✓ Show splitted parts] [✓ Show SEQ/ACK numbers]
Reassemble segments [✓ Show TCP flags]

HTTP
Show splitted parts Show version
[✓ Reassemble segments] [✓ Show HTTP method/status description]

SIP
[✓ RTP stream reassemble] [✓ Show FROM/TO]
[✓ Show SIP status description] Only nicknames (if are used)
Show version

Figure 8.4: Advanced settings

8.1.3 Aggregation Settings

The window *Aggregation Settings* (Figure 8.3) is related to aggregation feature mentioned in Chapter 7. The groupbox above allows to define a subset of packet's properties considered by their comparison. This dialog contains also the settings for specific case, when the selected subset can't be used. It allows also to select if the information about count and size of aggregated packets should be displayed in *message label*. In the bottom there are the radio buttons to select the aggregation type.

Note that many selected options in another PCAP dialog windows are ignored by aggregation because of abstraction related to this feature.

8.1.4 TCP, HTTP and SIP

The last dialog called *Advanced Settings* (Figure 8.4) contains some specific options related to the supported protocols. There are groupboxes for TCP, HTTP and SIP allowing to set the information that are included in *message labels*. For example, whether TCP message should contain also sequence and acknowledgment numbers or TCP flags. A user can also set if he wants to see only the type of HTTP packet (”GET”) or also its specification (”GET google.com”). Similar settings are there available also for SIP. This dialog allows to define if the protocol-specific reassembling features mentioned in Chapter 6 should be used.

8.2 Configuration files

Next to Microsoft Visio add-on SCStudio implementation it is possible to run the transformation from PCAP to MSC also from SCStudio command line interface. The information about user-defined configuration parameters selected in Visio add-on GUI are saved in Windows Registry. But the command line program can run except Windows also on UNIX and therefore an alternative solution for settings was chosen.

We have decided to use configuration files that enable to define the same transformation parameters as in GUI. The configuration files are written in *INI* [16] format and in contrast with Windows Registry are cross-platform.

The PCAP import program executed from command line takes as an argument beside the PCAP file also a related configuration file. The biggest advantage in contrast to GUI is the possibility of creating and using of more independent configuration files with different settings, according to user’s current needs.

Chapter 9

Performance of PCAP import feature

9.1 Analysis of problems

There are some problems with importing of bigger PCAP files in Microsoft Visio add-on.

The first performance problem was caused by functionality called *Beautify*, which allows a well-arranged layout of MSC diagrams to graphical editor. *Beautify* uses *linear solver* for arranging the *events* to the right places on *instances* according to their sequence. But it is not possible to get result for MSC with more than 400 *events*, i.e. 200 *messages* [17]. The MSC diagram generated from PCAP files is quite specific. The *events* are strictly ordered and to every event a time information is attached. Therefore another solution for MSC's graphical layout is possible, which uses another approach for *events* distribution along *instances* - position of *event* on *instance* is related to its time. This functionality was implemented by Tomáš Márton (*Time Relevant Layout* [18]) and it is obviously faster and more efficient than *Beautify* for PCAP import feature.

The second problem lies in Visio module and concerns slow graphical visualization of MSC diagrams. The table in Figure 9.1 provides information from measurements of importing PCAP file (two devices' communication) in Visio add-on. The analysis was executed on the computer with Intel(R) Core(TM) i3 CPU M350 and 4 GB of memory, running on 64bit Windows 7.

The PCAP import configuration was defined as follows : *Layer to parse:* Network layer, *Time format:* absolute time, *Pcap filter:* NONE, No active aggregations or reassembling.

Number of packets	Generating MSC [s]	Visualization [s]
50	0.029	5.738
100	0.030	15.445
150	0.032	30.483
200	0.036	51.153
300	0.038	112.557
500	0.047	334.562
600	0.055	529.711
700	0.057	796.295

Figure 9.1: Visio import measured data

Transformation from PCAP file to MSC Z.120 textual format works great from SCStudio command line interface. The measurements with the same PCAP configuration as for import to Visio are to see in Figure 9.2.

Number of packets	Processing time [s]
700	0.071
10 000	0.848
50 000	4.229
100 000	8.547
500 000	42.644
1000 000	85.189

Figure 9.2: Z.120 export measured data

9.2 Possible solutions

MSC diagram with more than 200 messages is quite difficult to read. For that reason it is appropriate to apply some filter or aggregation on bigger PCAP file. The communication is then more clear and the orientation in MSC diagram is more intuitive. The bigger MSCs should be used mainly for execution of verification algorithms implemented in SCStudio or other software tools. For that purpose Z.120 format is more suitable because of its simplicity and portability.

We have also regarded the solution to split the big MSC in more smaller diagrams and visualize them one per page in Microsoft Visio. However it is quite hard to estimate the efficiency of this approach and the implementation can be difficult in case that aggregations are used. The most suitable way is to reduce the performance problems directly in Visio drawing module, but until now we have not been able to find an appropriate solution.

Chapter 10

Conclusion

The functionality of importing PCAP files was successfully implemented in SCStudio. During this process we have met some problems with different platforms on which SCStudio is running. They were related mainly to compiling and linking of extern libraries and to using of specific system functions. We have also designed a new approach that derives the MSC diagrams from traffic via some kind of packets aggregation. This way we aspired to make the MSC diagram from a bigger PCAP file more compact and easier to read.

PCAP import feature fits well with an objective of SCStudio and provides a new approach for protocols analysis. The earlier supported methods how an MSC diagram could be created in SCStudio are drawing in graphic editor or import from Z.120 textual format. Now it is possible to generate an MSC diagram without any knowledge about this formalism, via automatic transformation from PCAP file. Our intention was to make the transformation from PCAP to MSC as adjustable as possible. Therefore it can be specified via settings in designed graphical user interface in Visio add-on or via configuration files in command line interface.

On the generated MSC verification algorithms implemented in SCStudio could be executed. One of the most interesting but not fully implemented is *find flow*. This feature allows to find the predefined MSC scenarios in bigger diagrams. In association with PCAP import functionality it will be possible to verify if the protocol's implementation meets its requirements defined by another MSC diagram. It could be a huge advantage for developers by designing the protocols and discovering bugs.

Other benefit of PCAP to MSC transformation is a possibility to gain a lot of specific MSC diagrams for testing through automatic and easy way. A developer can download a PCAP file with expected communication scenario from the internet or execute a sniffing on network via suitable tool (Wireshark). Our intention is, that in the future it should be possible to visualize the MSC diagram in real time, consequently with packet capturing in SCStudio. However, firstly the performance problems with diagrams rendering in Visio add-on have to be removed.

Bibliography

[1] ITU-T Telecommunication Standardization Sector of ITU - Study group 17. ITU recommendation Z.120, Message Sequence Charts (MSC), 2011.

[2] ANF DATA spol. s r.o. and Masaryk University the research center Institute for Theoretical Computer Science (ITI), Faculty of Informatics. The Sequence Chart Studio. [online], [cited April 15, 2012]. Available at: <`http://scstudio.sourceforge.net/`>.

[3] PCAP-DumpFileFormat2. [online], [cited March 31, 2013]. Available at: < `http://www.tcpdump.org/pcap/pcap.html` >.

[4] Rajeev Alur, Kousha Etessami, and Mihalis Yannakakis. Inference of message sequence charts. In *Software Concepts and Tools*, pages 304–313, 2003.

[5] Martin Garcia. Programming with libpcap. sniffing the network from our own application. *Hakin9 Magazine, English Edition*, 3(2):163–173, 2008.

[6] Steven Mccanne and Van Jacobson. The bsd packet filter: A new architecture for user-level packet capture. pages 259–269, 1992.

[7] Fulvio Risso and Loris Degioanni. An architecture for high performance network analysis. In *Proceedings of ISCC 2001, Hammamet*, pages 686–693, 2001.

[8] Wireshark-Development/LibpcapFileFormat:wiki. [online], [cited March 31, 2013]. Available at: < `http://wiki.wireshark.org/Development/LibpcapFileFormat` >.

[9] ISO/IEC 7498-1:1994. *Open System Interconnection-Basic Reference Model: The basic model*. ISO, Geneva, Switzerland, 1994. Available at: <`http://standards.iso.org/ittf/PubliclyAvailableStandards/index.html`>.

[10] University of Southern California Information Sciences Institute. Definitions of Managed Objects for the Ethernet-like. RFC 793, RFC Editor, September 1991.

[11] HTTP parser. [online], [April 1, 2013]. Available at: < `https://github.com/joyent/http-parser` >.

[12] The GNU oSip library. [online], [cited April 1, 2013]. Available at: < `http://www.gnu.org/software/osip` >.

[13] E. Elkind, B. Genest, and D. Peled. Detecting Races in Ensembles of Message Sequence Charts. In *TACAS'07*, volume 4424 of *LNCS*, pages 420–434. Springer, 2007.

[14] P. Slovák. Decidable Race Conditions in High-Level Message Sequence Charts. *Bachelor thesis*, Faculty of Informatics, Masaryk University, Brno, 2008.

[15] E. O'Tuathail. WTL developers guide. [pdf], [cited April 15, 2012]. Available at: <`http://www.assembla.com/code/cristianadam/subversion/nodes/WTL%20Developer's%20Guide`>.

[16] INI file format. [online], [cited May 8, 2013]. Available at: < `http://en.wikipedia.org/wiki/INI_file` >.

[17] M. Malota. Layout Configuration for Message Sequence Charts. Bachelor's thesis, Masaryk University, Faculty of Informatics, 2012.

[18] T.Marton. Time relevant layout of MSC. Bachelor's thesis, Masaryk University, Faculty of Informatics, 2013.

Appendix A

Contents of Attached DVD

The attached DVD contains the following items:

- a PDF and LaTeX version of the thesis;
- source code of implemented protocols (data_link_layer.h, data_link_layer.cpp, network_layer.h, network_layer.cpp, transport_layer.h, transport_layer.cpp, application_layer.h, application_layer.cpp, packet.h, packet.cpp, protocols.h);
- source code of transformation and aggregation (pcap_settings.h, pcap_settings.cpp, aggregation_formats.h, aggregation_formats.cpp, aggreg_handler.h, aggreg_handler.cpp, pcap_handler.h, pcap_handler.cpp, pcap_load.h, pcap_load.cpp, bytes.h);
- source code of dialog windows;
- executable installation file of SCStudio;
- source code of SCStudio
- template of **.ini* configuration file for transformation
  from PCAP to Z.120 textual format;