Design: Analyzer

DaCoPAn

Helsinki 31st May 2004

Software Engineering Project

UNIVERSITY OF HELSINKI
Department of Computer Science

UNIVERSITY OF PETROZAVODSK
Department of Computer Science

Course

581260 Software Engineering Project (6 cr)

Project Group

Carlos Arrastia Aparicio
Jari Aarniala
Alejandro Fernandez Rey
Vesa Vainio
Jarkko Laine
Jonathan Brown

Kirill Kulakov
Andrey Salo
Andrey Ananin
Mikhail Kryshen
Viktor Surikov

Customer

Markku Kojo

Project Masters

Juha Taina (Supervisor)
Yury Bogoyavlenskiy (Supervisor)

Turjo Tuohiniemi (Instructor)
Dmitry Korzun (Instructor)

Homepage

`http://www.cs.helsinki.fi/group/dacopan`

Change Log

Version	Date	Modifications
1.0	10.04.04	First version

Contents

1 Introduction

This document defines design for Analyzer subsystem of the DaCoPAn software according to [11]. The document can be considered as a model for sufficient implementation of the requirements, stated in the Requirement Specification document [10].

Decomposition of the Analyzer subsystem into principal modules is presented in section 3. Analyzer user interface is presented in section 4. Behavioral model describes details of module interface design, see section 7. This model is also needed for testing. Section 5 states all data structures used by the analyzer. Each module is designed in detail; this is presented in section 6. Configuration and installation issues are briefly described in section 8.

The document is intended mainly for the project development team. Experts from customer's side may analyze this document to be sure that the requirements are going to be implemented sufficiently and efficiently.

This specification may be changed during the implementation phase. All such changes must be shortly described and grounded in a separate document — The Implementation Document.

2 Glossary

Event (section 5.5) is a data structure that describes a unit of network protocol exchange; a unit can be sent, received or lost. For example "IP datagram was sent", "TCP segment was received", "HTTP message was lost" are `events`.

Events Sequence (ES) (section 5.5) is a data structure that stores a list of all protocol `events` in chronological order.

Header fields is a data structure that contains fixed packet fields for each protocol type.

Message Exchange List (MEL) (section 5.3) is a data structure that is used to store the information about links between sent/received units from different hosts.

Packet Trace Presentation (PTP) (section 5.1) is a data structure that presents one packet trace log as a list of PTUs in chronological order.

Packet Trace Unit (PTU) (section 5.1) is a data structure that stores information from Packet trace files, links between PTU for PTP construction (see section 5.2) and send/receive correspondence of units.

Packet Trace Units Sequence (PTUS) (section 5.4) is a data structure that stores common time sequence of sent/received units.

Protocol variables are end host characteristics of protocol implementation or characteristics of the network link between hosts. For example MTU, cwnd, TCP states, etc.

3 Architecture

The *Analyzer* subsystem architecture is modeled with two diagrams (see Fig. 1 and 2).

3.1 Decomposition into modules and their relations

Analyzer composition model is shown in Figure 1.

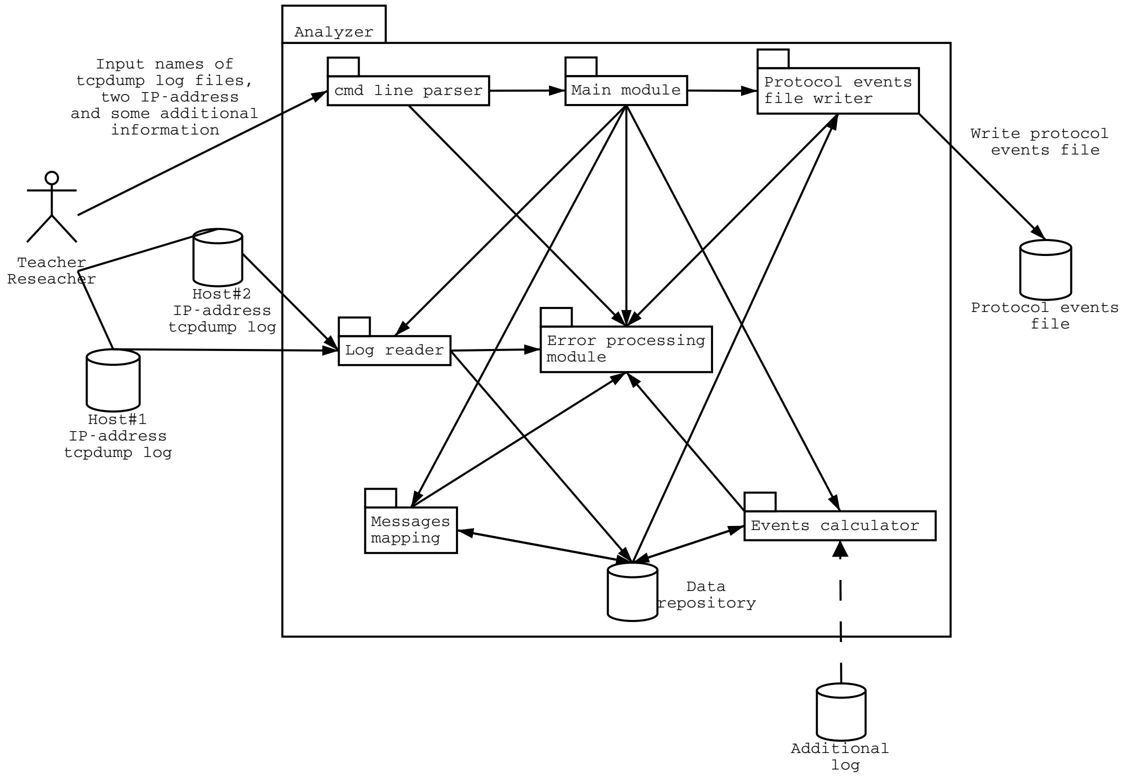

Figure 1: Composition model of the DaCoPAn Analyzer subsystem

The main module controls the analyzer subsystem (sect. 6.1). A user interacts with the Analyzer via command line user interface (sect. 4).

Packet trace files are primary input data; there are read by Log reader module (sect. 6.3). For the required processing the main module calls sequentially modules: Message mapper (sect. 6.4) and Events calculator (sect. 6.5), which transform the input data inside common data repository. The repository is designed as special data structures (sect. 5).

The result data are extracted from the repository and written out as an XML file (see [9]) by module Protocol events file writer (sect. 6.6).

Each module can course an error. This event is forwarded directly to the error processing module (sect. 6.7).

3.2 Data flows

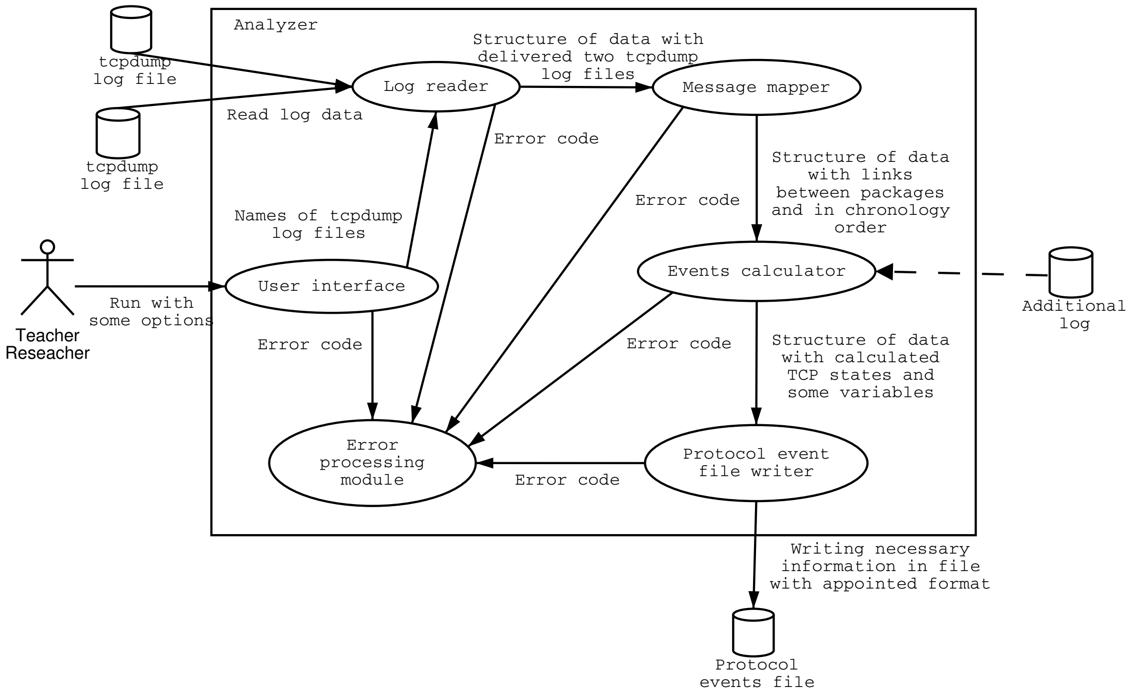

Figure 2: Data flow diagram of the DaCoPAn subsystem Analyzer

Figure 2 is a data flow diagram. It shows key data flows between the modules.

The **Analyzer** is intended for creation **Protocol events file** from packet trace files and additional input logs. **Log reader** gets header fields and timestamps from packet trace and fills Packet trace presentations (PTP, see sect. 5.2). **Message mapper** gets PTPs as input data and builds Message exchange list (MEL, see sect. 5.3) with links between corresponding packets. **Events calculator** splits Message exchange sequence into Events sequence (ES, see sect. 5.5) and calculates necessary protocol variables (gets necessary information for calculating from additional logs). After calculating **PEF writer** puts Events sequence in Protocol events file. Each module calls **Error processing module** whenever an error appears.

4 User interface

The Analyzer used a standard console user interface. The User interface consists of three modules: Command line parser (see section 6.2), Protocol events file writer (see section 6.6) and Error processing module (see section 6.7). Command line parser module gets a list of options from user, prints usage and help info and prints error messages that are processed by `getopt` library. Error processing module prints all errors. Results of Analyzer are printed by Protocol events file writer module.

5 Data structures

5.1 Packet trace unit

Packet trace unit (PTU) is the data structure, which is used by log reader and message mapper modules. It contains timestamp of the packet, network, transport, and application levels information, links, which are needed to prepare packet trace presentation (see section 5.2), and maps corresponding packet trace unit into the other packet trace presentation. The high-level structure of PTU is shown in Figure 3.

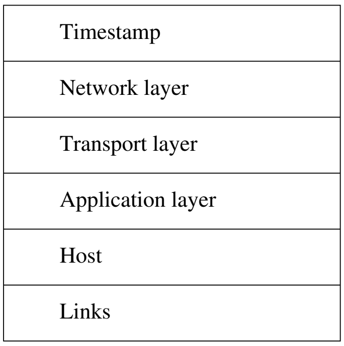

Figure 3: Packet trace unit (PTU)

5.1.1 Timestamp

Each packet in the packet trace file captured by tcpdump tool is prepended with the generic header, which is needed to get around the problem of different headers for different packet interfaces. This generic header is defined in `pcap.h` file as `pcap_pkthdr` structure.

- Data Type:

```
struct pcap_pkthdr {
        /* time stamp */
        struct timeval ts;
        /* length of portion present */
        bpf_u_int32 caplen;
        /* length this packet (off wire) */
        bpf_u_int32 len;
};
```

The `ts` member value of this structure is used as time stamp for PTU.

5.1.2 Network layer

The network layer of the PTU should contain all information which is relevant to network layer of the captured packet.

The basic case is IPv4 datagram. IPv4 datagram format is presented in Figure 4. Internet Protocol specification could be found in RFC 791 [5].

<table>
<tr><td colspan="4">0</td><td>31</td></tr>
<tr><td>Version</td><td>IHL</td><td>Type of Service</td><td colspan="2">Total Length</td></tr>
<tr><td colspan="3">Identification</td><td>Flags</td><td>Fragment Offset</td></tr>
<tr><td colspan="2">Time to Live</td><td>Protocol</td><td colspan="2">Header Checksum</td></tr>
<tr><td colspan="5">Source Address</td></tr>
<tr><td colspan="5">Destination Address</td></tr>
<tr><td colspan="5">Options (optional)</td></tr>
</table>

Figure 4: IPv4 datagram header structure

The secondary case is ARP packet. The ARP header structure is presented in Figure 5. The Ethernet Address Resolution Protocol is described in RFC 826 [2].

<table>
<tr><td colspan="2">0</td><td>31</td></tr>
<tr><td colspan="2">Hardware Address Type</td><td>Protocol Address Type</td></tr>
<tr><td>H/w Addr Len</td><td>Prot. Addr Len</td><td>Operation</td></tr>
<tr><td colspan="3">Source Hardware Address</td></tr>
<tr><td colspan="2">Source Hardware Address (cont.)</td><td>Source Protocol Address</td></tr>
<tr><td colspan="2">Source Protocol Address (cont.)</td><td>Target Hardware Address</td></tr>
<tr><td colspan="3">Target Hardware Address (cont.)</td></tr>
<tr><td colspan="3">Target Protocol Address</td></tr>
</table>

Figure 5: ARP header structure

5.1.3 Transport layer

The transport layer of the PTU should contain all information that is relevant to transport layer of the captured packet. The transport layer type is defined using protocol field of IPv4 header.

The first basic case is TCP segment. TCP segment structure is presented in Figure 6. Transmission Control Protocol is described in RFC 793 [6].

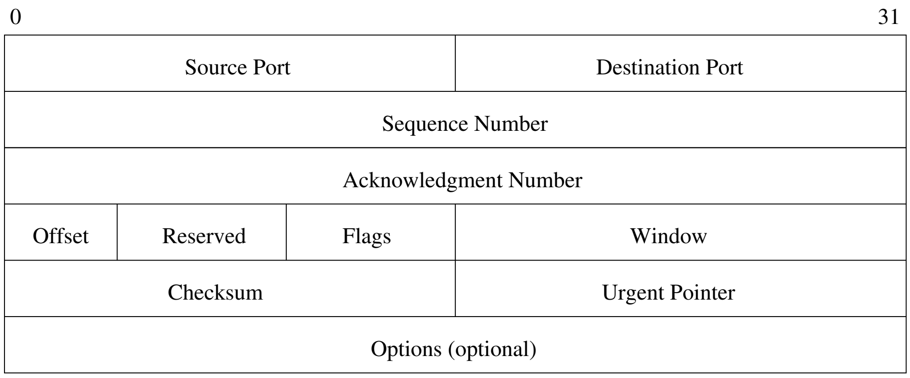

Figure 6: TCP segment header structure

The second basic case is UDP segment. UDP segment structure is presented in Figure 7. User Datagram Protocol is described in RFC 768 [4].

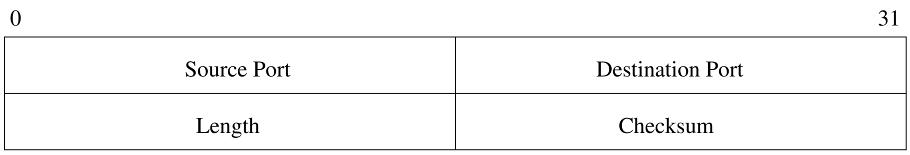

Figure 7: UDP segment header structure

5.1.4 Application layer

The application layer of the PTU should contain all information that is relevant to application layer of the captured packet. The application layer type is detected using source or destination port field of TCP or UDP header.

The first basic case is HTTP message. The general format of HTTP request and response messages is presented in Figures 8 and 9. Hypertext Transfer Protocol is described in RFC 1945 (Version 1.0) [1], RFC 2616 (Version 1.1) [3].

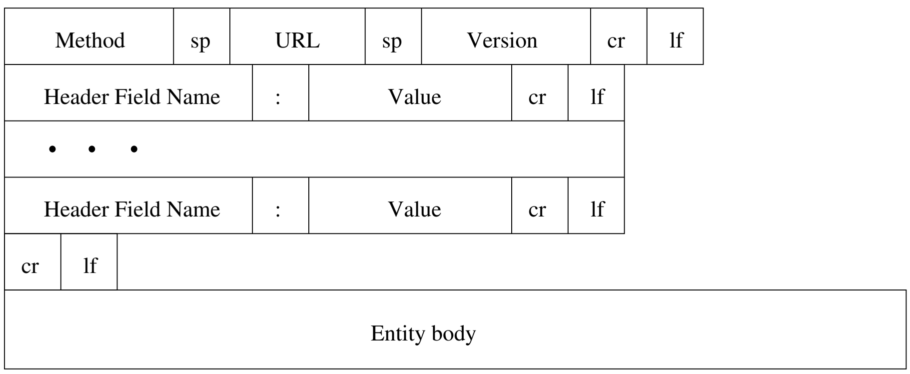

Figure 8: Format of HTTP request message

Version	sp	Status Code	sp	Phrase	cr	lf
Header Field Name	:	Value	cr	lf		
• • •						
Header Field Name	:	Value	cr	lf		
cr	lf					
Entity body						

Figure 9: Format of HTTP response message

The second basic case is DNS message. The DNS message format is presented in Figure 10. Domain Name Service is described in RFC 1034 [7], RFC 1035 [8].

0	31
Identification	Flags
Number of Questions	Number of Answer Resource Records
Number of Authority Resource Records	Number of Additional Resource Records
Questions (variable number of questions)	
Answers (variable number of resource records)	
Authority (variable number of resource records)	
Additional information (variable number of resource records)	

Figure 10: DNS message format

5.1.5 Host

Each packet trace unit belongs to some packet trace log. The host part of packet trace unit contains information on the host corresponding to this packet trace log. This is needed to identify the packet trace unit after producing packet trace unit sequence (section 5.4).

5.1.6 Links

PTU has two members, which are pointers to PTU structures. These pointers are used as links. The first link is used to form the packet trace presentation (see section 5.2). The second link is used by message mapper while mapping packet trace units between different packet trace presentations.

5.1.7 Packet trace unit structure

- Data Type:

```
struct ptu {
    /* Timestamp */
    struct timeval timestamp;

    /* Network layer */
    int net_type;
    void *net_hdr;

    /* Transport layer */
    int trans_type;
    void *trans_hdr;

    /* Application layer */
    int app_type;
    void *app_data;
    int app_len;

    /* Host */
    struct host *host;

    /* Links */
    struct ptu *link;
    struct ptu *next;

    /* Corresponding events.  */
    event *net_event;
};
```

This structure is the packet trace unit representation. All its members are filled by the Log reader module while analyzing packet trace file and preparing packet trace presentation (section 5.2), except the link, which is filled by the Message mapper module while preparing message exchange list (section 5.3).

Members:

- `struct timeval ts` is the time stamp of the captured packet trace unit.
- `int net_type` determines the type of the network layer protocol. It should be one of the macros related to network layer (listed below).
- `void *net_hdr` is a pointer to the corresponding network layer protocol header.

  - `int trans_type` determines the type of the transport layer protocol. It should be one of the macros related to transport layer (listed below).
  - `void *trans_hdr` is a pointer to the corresponding transport layer protocol header. The pointer should be a null pointer if there is no transport header, or a pointer to octet sequence representing the transport layer header otherwise.
  - `int app_type` determines the type of the application layer protocol. It should be one of the macros related to application layer (listed below).
  - `void *app_data` is a pointer to corresponding application layer related data. The pointer should be a null pointer if no application layer related data present, or a pointer to octet sequence representing the application layer data otherwise. If the pointer is non-null, the length of the data is defined using `app_len` value.
  - `int app_len` contains the length of application layer related data. The member should be equal to zero if the `app_data` pointer is null, or it should contain the length of the octet sequence representing the application layer data otherwise.
  - `struct host *h` points to structure describing the host (section 5.5) corresponding to packet trace log. The packet trace unit should belong to this log.
  - `struct ptu *link` is used to map to corresponding packet trace unit in the other packet trace presentation. It is filled by message mapper while preparing message exchange list (section 5.3).
  - `struct ptu *next` is used to prepare linked list, which is packet trace presentation (section 5.2).

- Macros:

  `PTU_IPV4` determines that network layer protocol is IPv4.

  `PTU_ARP` determines that network layer protocol is ARP.

  `PTU_TCP` determines that transport layer protocol is TCP.

  `PTU_UDP` determines that transport layer protocol is UDP.

  `PTU_DNS` determines that application layer protocol is DNS.

  `PTU_HTTP` determines that application layer protocol is HTTP.

  `PTU_NET_UNKNOWN` determines that network layer protocol is unknown.

  `PTU_TRANS_UNKNOWN` determines that transport layer protocol is unknown.

  `PTU_APP_UNKNOWN` determines that application layer protocol is unknown.

  All the macros must be expanded to an unique integer value. They should be used to determine protocol type for corresponding layer.

5.1.8 Expandability

The structure of packet trace unit provides expandability in the following ways:

1. Adding a new protocol.

   The PTU structure itself can be used without any changes, since the header pointers are of type void. It is needed to define new macro for new protocol type and corresponding structure for protocol header to use specific protocol header variables.

2. Using tools other than tcpdump.

   The PTU structure can be used without any changes. The only condition is that the tool should be able to provide information about time stamp of the captured packet (it is implied that the traffic capturing tool is able to grab the whole packet).

3. Using more than two packet trace logs.

   The PTU structure can be used without any changes.

5.2 Packet trace presentation

The packet trace presentation (PTP) is used to represent one packet trace log as a list of packet trace units in chronological order as they were captured by tcpdump tool (section 5.1). PTP is the normal output of the Log reader module. Several packet trace presentations are the input for the Message mapper module.

5.2.1 Packet trace presentation structure

The packet trace presentation is a linked list of packet trace units. The PTU is created by the Log reader module. The BEGIN is a pointer to the beginning of the PTP, i.e. the first PTU. When the Log reader gets next PTU from the packet trace file, it sets the `next` link of the previous PTU to this PTU. The `next` link of this PTU is set to null. Thus, `next` pointer equality to zero means end of the list. The structure of PTP is presented in Figure 11.

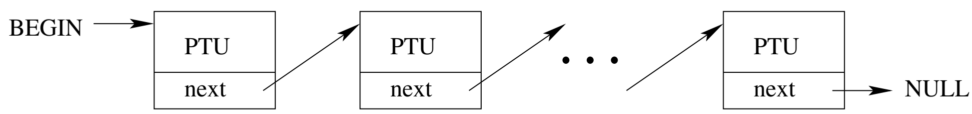

Figure 11: Packet trace presentation (PTP)

5.2.2 Expandability

The packet trace presentation provides expandability in the same ways as the packet trace unit (section 5.2).

5.3 Message exchange list

5.3.1 Description

This structure of the data is used to store the information about links of the packages sent from one host and received on the destination side (see Figure 12). Links are directed from the received to sent sides. It is represented with two (basic case) PTP lists, which have been created by Log reader (see section 6.3). If for one PTU in one Packet trace presentation corresponding PTU was found then connecting link is created.

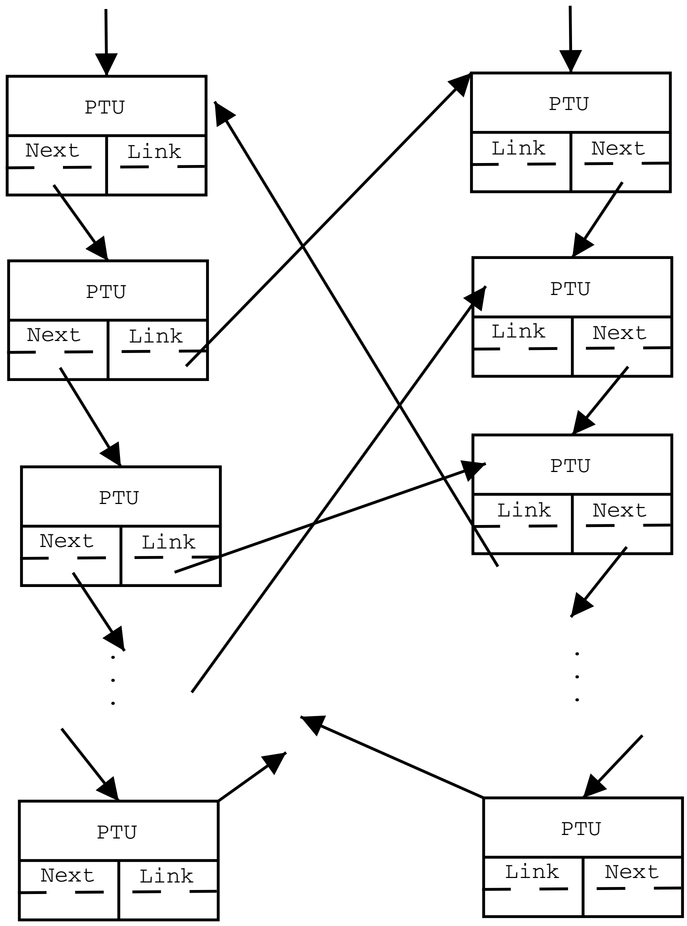

Figure 12: Message exchange list

5.3.2 Usage

Message exchange list is used by the Message mapper module (see section 6.4) as temporary structure in the process of building a PTU sequence.

5.3.3 Expandability

1. Using more than two packet trace logs.

The MEL structure can be used without any changes.

5.4 Packet Trace Units Sequence

5.4.1 Description

This structure is used for building the common time-ordered sequence of sent and received units for all given hosts. Structure of PTU sequence is shown in Figure 13.

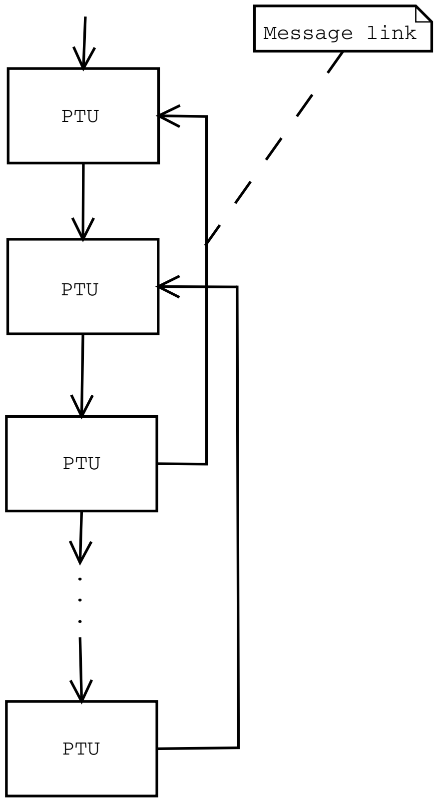

Figure 13: Protocol Trace Units Sequence

5.4.2 Usage

The PTU sequence is produced by Message mapper module (see section 6.4) and used as input data for the Events calculator (see section 6.5).

5.4.3 Expandability

1. Using more than two packet trace logs.

   The MEL structure can be used without any changes.

5.5 Events sequence and other structures

5.5.1 Description

Events sequence is a linked list of `event` elements of three possible types: protocol exchange unit sent, protocol exchange unit received and protocol exchange unit lost.

Each list element contains information about the event (`type`, `timestamp`), links to the unit `header` and data (`data`, `data_length`), protocol `variables`, link to the `next` and `prev` events (chronological order), link to the `children` event (link from the low-level packet to the next high-level layer packet), link to the first `parent` event, link to the last parent of the event `last_parent`, link to the next element in the linked list of parents (`next_parent`), and for the sent event link to the corresponding receive or lost event (`unit_next`). `index` link shows the first and the last sent or received fragment in the list of parents.

Each `event` element has links to one source and destination hosts in the linked list of hosts `host_next, host_prev`. Also each `event` element has link to one `flow` element.

All `host` elements are organized in a linked list of hosts and `flow` elements are organized in a linked list of flows. `Hostname` field in `Host` structure is reserved for the future versions of the Analyzer.

Structure `app_ports` is used for storing ports for application layer protocols.

The structure of Events sequence is shown in Figure 14 (some fields are not represented).

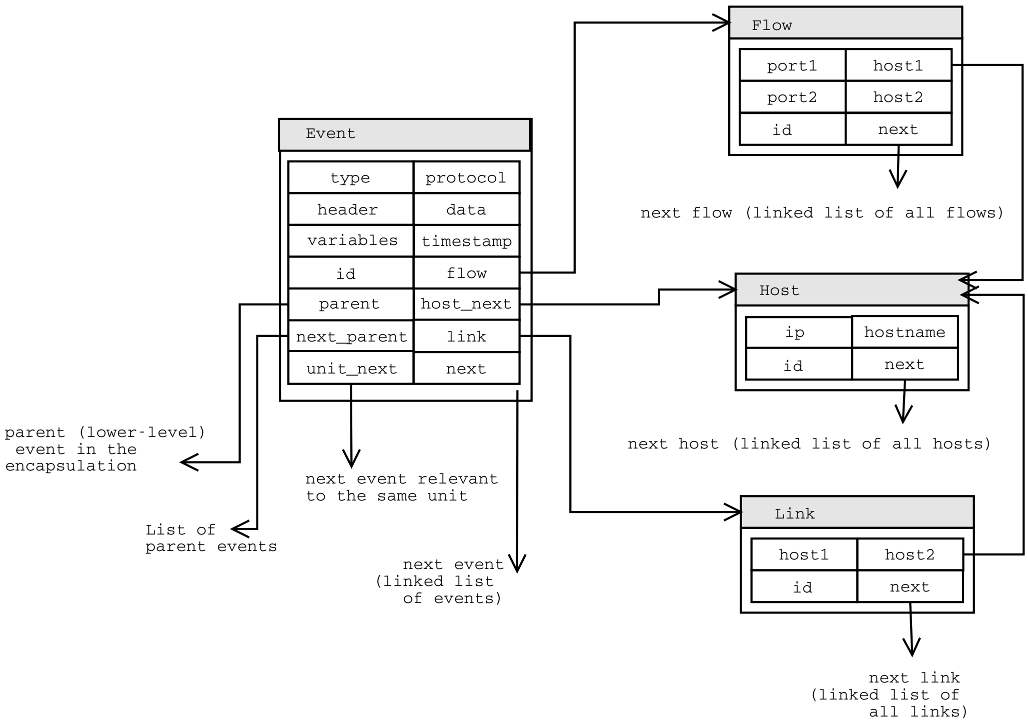

Figure 14: Events Sequence

5.5.2 Structure definition

```
struct host {
    struct in_addr ip;
    char *hostname;

    /* will be set by the PEF Writer.  */
    char *id;

    /* Packet trace file name.  */
    char *ptfname;

    /* Time difference.  */
    double timealign;

    /* the next element in a linked list.  */
    struct host *next;
};

struct flow {
    struct host *host1;
    int port1;

    struct host *host2;
    int port2;

    /* will be set by the PEF Writer */
    char *id;

    /* the next element in a linked list */
    struct flow *next;
};

struct link {
    struct host *host1;
    struct host *host2;

    /* will be set by the PEF Writer */
    char *id;

    /* the next element in a linked list */
    struct link *next;

    /* The MTU value */
    double MTU;

};

/* the events sequence is a linked list of elements of
   the structure defined below */
struct event {
    /* type of the event: UNIT_SENT, UNIT_RECEIVED, UNIT_LOST */
    int type;

    /* unit id, will be set by the PEF Writer */
    char *id;

    struct host *host;
    struct link *link;
```

```
    struct flow *flow;

    /* event time */
    struct timeval timestamp;

    /* protocol type */
    int protocol;

    /* protocol header - type of the referenced structure
       depends on the protocol field value */
    void *header;

    /* the unit data */
    void *data;

    /* length of the unit data */
    int data_length;

    /* protocol variables */
    struct variables vars;

    /* child  event (encapsulation) */
    struct event *child;

    /* parent event (encapsulation) */
    struct event *parent;

    /* last parent of the event */
    struct event *last_parent;

    /* used in case of fragmentation to
       create a linked list of parents */
    struct event *next_parent;

    /* Corresponding sent event.  */
    struct event *unit_prev;

    /* corresponding receive or lost event (non-NULL for UNIT_SENT) */
    struct event *unit_next;

    /* Source host (non-NULL for UNIT_RECEIVED) */
    struct host *host_prev;

    /* Destination host (non-NULL for UNIT_SENT) */
    struct host *host_next;

    /* the previous event in the sequence */
    struct event *next;

    /* the next event in the sequence */
    struct event *next;

    /* Used to determine the first and the last
       sent or received fragment in the list of parents.  */
    unsigned int index;
};

/* This structure is used to store ports
   for application layer protocols.  */
```

```
struct app_ports
{
    u_int16_t htpp[MAX_PORTS];  /* List of HTTP protocol ports.  */
    int http_count; /* HTTP protocol ports counter.  */

    u_int16_t dns[MAX_PORTS]; /* List of DNS protocol ports.  */
    int dns_count; /* DNS protocol ports protocol. */
};
```

5.5.3 Usage

The Events Sequence is produced by the Layer Splitter submodule of the Events Calculator (see section 6.5) and used in the Events Calculator and PEF Writer modules (see section 6.6). Linked lists of host and flows is used for outputing in the resulting protocol events file.

5.5.4 Expandability

1. support for more than two hosts.

   The ES structure can be used without any changes.

2. possibility to support new protocols.

   The ES structure can be used without any changes.

3. possibility to add new types of events.

   The ES structure can be used without any changes.

4. suitable for real-time processing of the communication data.

   The ES structure can be used without any changes.

6 Modules

6.1 Main module

Description

The main module contains all global variables and data structures. This module is responsible for control and direct of Analyzer modules. Also the main module deletes unused global data structures. The global data structure descriptions contains in separate header files. The Analyzer is a standard console application. The main module uses call-return control scheme for consecutive module calls. The general main module work scheme is the following.

```
int main(int argc, char **argv){
  /* call User interface module */
  /* call Log reader module */
  /* call Message mapper module */
  /* call Events calculator module */
  /* call PEF writer module */
  return 0;
}
```

Analyzer returns with a return code. If Analyzer was successfully exit then main module returns zero. Otherwise Error processing module or Command line parser module returns nonzero error code (see sect. 6.7, 6.2). Concrete code values will be defined on Implementation phase.

Functions

- Function:

  `int main ( int argc, char ** argv )`

  This function is used to consecutive execution other functions.

  Arguments:

  – `int argc` is a number of command line arguments,
  – `char **argv` is a array of command line arguments.

  Returns:

  The returns zero on success, otherwise — error code.

Possible types of errors

The main module does not directly calls Error processing module.

Implementation notes

- All global data structures contains in separate header files (`*.h`). The main module consists of two files `analyzer.h` and `analyzer.c`. The file `main.h` contains routines declarations. The file `analyzer.c` contains the routines code.

6.2 Command line parser

Description

This module is responsible for processing command line arguments of the Analyzer. It is a part of user interface, which is intended for parsing user input in command line, and notifying the user on command line errors. If the input is valid, the module should set all corresponding program variables to user specified values. The other command line parser destination is to show standard help and version information.

Command Line Arguments

The Analyzer command line options are designed according to POSIX conventions for command line arguments and the GNU coding standards.

The program has two mandatory non-optional arguments — the names of the packet trace files. If the number of arguments is not equal to 2, the program should exit with an error.

The command line options are available in two variants — short and long. The Analyzer has the following command line options:

- `-i, --ip-addresses` option is used to specify IP addresses corresponding to packet trace files. This option requires the argument — a list of two comma separated IP addresses in standard numbers-and-dots notation. There should be no white spaces inside the list. Alternatively, the option could be specified twice with single IP address as an argument. If the number of IP addresses is not equal to 2, the program should exit with an error.
- `-l, --var-log` option is used to specify the file, containing some host-specific variables, which can not be determined from packet trace logs.
- `-o, --output` option is used to specify output file name. This option requires the argument — the name of the output file. This option is not mandatory. The default output file name is used if the option is not specified.
- `-t, --time-align` option is used to specify the time difference value between the hosts. This option is not mandatory. The time difference is assumed to have a zero value if the option is not specified.
- `--http` option is used to specify list of HTTP ports in command line.
- `--dns` option is used to specify list of DNS ports in command line.
- `-h, --help` is a standard option specified in the GNU coding standards. This option should output brief documentation for how to invoke the program, on standard output, then exit successfully. Other options and arguments should be ignored once this is seen, and the program should not perform its normal function. The first line (or lines) should contain short program usage information. The next lines should contain list of all the program options and arguments with short descriptions. Near the end of the option's output there should be a line that says where to mail bug reports.
- `-v, --version` is another standard option specified in the GNU coding standards. This option should direct the program to print information about its name, version, origin and legal status, all on standard output, and then exit successfully. Other options and arguments should be ignored once this is seen, and the program should not perform its normal function. The first line is meant to be easy for a program to parse; the version number proper starts after the last space. In addition, it contains the canonical name for this program. The program's name should be a

constant string. The standard or canonical name for the program should be stated, not its file name. The following line, after the version number line or lines, should be a copyright notice.

The simplest way to invoke the program is:

```
$ analyzer -i IP1,IP2 FILE1 FILE2
```

or

```
$ analyzer -i IP1 -i IP2 FILE1 FILE2
```

The following table is the Analyzer command line options summary. It can also be used to prepare help information.

Option	Description
`-i, --ip-addresses=IP1,IP2`	Specify IP addresses
`-l, --var-log=FILE`	Read variables from `FILE`
`-o, --output=FILE`	Place the output into `FILE`
`-t, --time-align=VALUE`	Set time difference to `VALUE`
`    --http=PORT1[,...]`	Set HTTP ports to `PORT1,...`
`    --dns=PORT1[,...]`	Set DNS ports to `PORT1,...`
`-h, --help`	Show brief documentation and exit
`-v, --version`	Show version information and exit

Variables and Macros

- Variable:

  ```
  extern char *program_invocation_name;
  ```

  This string is the program name.

Functions

- Function:

  ```
  void parse_args (int argc, char **argv)
  ```

  This function is used to parse command line arguments and set global variables to user specified values. The arguments of the function are exactly `main` function arguments. If an error has been occurred while parsing command line arguments, the program is terminated and the corresponding error message is displayed.

Arguments:

- `int argc` argument value is the number of command line arguments.
- `char **argv` is a vector of strings; its elements are individual command line argument strings.

- Function:

  ```
  void anlz_version ()
  ```

  This function is used to print program version information.

- Function:

  ```
  void anlz_help (int tip)
  ```

  This function is used to print program help screen.

  Arguments:

  - `int tip` , if this value is set to 0, then program help screen will be shown, otherwise short tip will be shown.

Possible types of errors

There are following types of errors:

1. Command line syntax errors.

   These errors are produced by `getopt` function when command line has syntax errors. This errors prints to user without using Error processing module. After printing error message Command line parser module exit abnormally.

2. Wrong arguments.

   These errors are produced when user inputs wrong arguments or did not input all mandatory arguments. The descriptive error message string should be passed to error processing module. The command line parser itself should not output any error messages.

Implementation Notes

- The option `--var-log`, which is used to specify file with additional variables, should be recognized by the command line parser, but it would not be implemented in this version of the Analyzer.

- The command line parser consists of two files: `cmdlparse.h` and `cmdlparse.c`. The file `cmdlparse.h` contains the routines declaration. The file `cmdlparse.c` contains the variables and the routines code.

- `getopt_long` function could be used to parse command line arguments. The function `strtok` could be used to parse IP addresses list.
- `stderr` should be used to print error messages, `stdout` should be used to print help and version information.

References on user requirements

- 'Tcpdump logs', [10].
- 'Time difference', [10].

6.3 Log reader

Description

This module is responsible for reading binary tcpdump log files and producing packet trace presentation (PTP — section 5.2). It also uses the global variables containing IP addresses of the hosts to filter the packets related to either host. The module should return a PTP, which is a list of packet trace units (PTU — section 5.2) containing packet headers information on success.

Packet trace log processing

The log reader module processes the packet trace log using the following scheme:

1. Open packet trace log.
2. Set filter.
3. Determine link layer type and use corresponding callback function.
4. Get next packet and create new PTU for it.
5. Fill PTU by calling corresponding protocol processing functions as it is shown in Figure 15 and defined application port numbers.
6. Add the PTU to the end of PTP.
7. If there are more packets, go to 4.
8. Close packet trace log.
9. Return packet trace presentation.

The log reader module should call only the error processing module routines on errors, and shouldn't use any other ways for processing errors and showing messages.

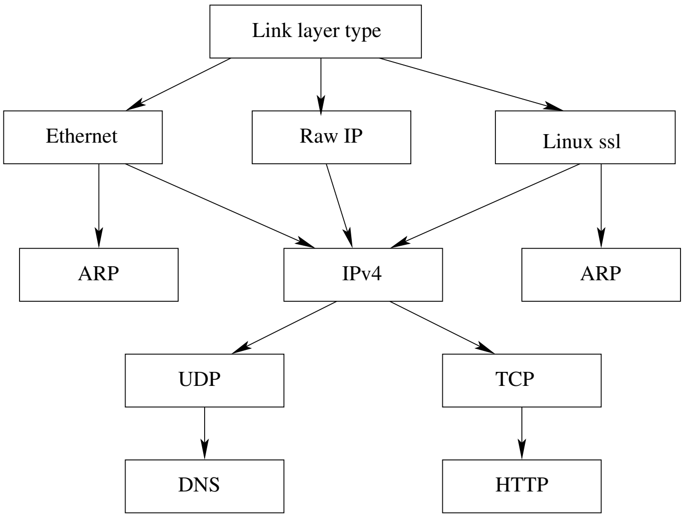

Figure 15: Packet processing scheme

IP fragmentation

The special case in packet trace log processing is fragmented IP packet. It is assumed, that the transport layer protocol header is completely placed in the first segment (with offset 0) as it is shown in Figure 16. Thus, the PTU, which represents packets with non-zero IP offset, should have only application layer data. The transport layer protocol type should be set to none and the transport layer protocol header pointer should be null.

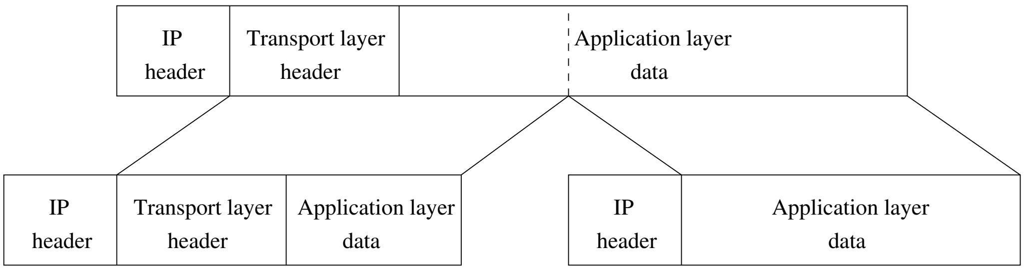

Figure 16: IP fragmentation

Variables and Macros

- Variable:

  `struct ptu *ptp_begin`

  This is a pointer to the beginning of the packet trace presentation. It is used by the callback functions.

- Variable:

  `struct ptu *ptp_tmp`

This is a pointer to the current packet trace unit. It is used by the callback functions.

- Variable:

  ```
  int data_link_type
  ```

  This variable is used by callback function to determine data link type.

Functions

- Function:

  ```
  void read_file (struct host *h, ptu **begin)
  ```

  This function is used to read the packet trace file, which is binary file captured by tcpdump tool. The function tries to open the file for reading, read the data, filter all packets related to either host, prepare a packet trace presentation (PTP), and close the file.

  Arguments:

  – `struct host *h` is the host structure which contains packet trace filename which is opened using `pcap_open_offline` function.
  – `struct ptu **begin` should be a pointer to the beginning of the packet trace presentation (linked list of packet trace units) on success.

- Function:

  ```
  void lr_setfilter (pcap_t *pcd)
  ```

  This function is used to set filter for the log reader. The filter passes a packet if it is a ARP packet or if its source or destination address is either of IP addresses specified by the user, and drops the packet otherwise. The filter is set using `pcap_compile` and `pcap_setfilter` functions.

  Arguments:

  – `pcap_t *pcd` is the packet capture descriptor.

- Functions:

  ```
  void lr_callback (u_char *unused,
                       const struct pcap_pkthdr *hdr,
                       const u_char *packet)
  ```

  These function is used to process data for all data link types of packet trace log. This function should create new PTU, fill it by calling corresponding protocol processing function as it is shown in Figure 15, and link the created PTU to the PTP.

Arguments:

- `u_char *unused` is unused.
- `const struct pcap_pkthdr *hdr` is the pcap header, which contains general information on packet.
- `const u_char *packet` is a pointer to the sequence of bytes representing the entire packet as captured by tcpdump tool.

- Functions:

```
void read_linux_ssl (u_char *packet, u_int len)

void read_ether (u_char *packet, u_int len)

void read_arp (u_char *packet, u_int len)

void read_ipv4 (u_char *packet, u_int len)

void read_tcp (u_char *packet, u_int len)

void read_udp (u_char *packet, u_int len)

void read_http (u_char *packet, u_int len)

void read_dns (u_char *packet, u_int len)

void read_app_unknown (u_char *packet, u_int len)
```

These function are used to process corresponding protocols data. Some of them may call others as it is shown in Figure 15.

Arguments:

- `u_char *data` is a pointer to the corresponding protocol related data (sequence of bytes).
- `u_int len` is the length of data.

Possible types of errors

The log reader errors can be divided into the following groups:

1. Memory allocation errors — the group of errors, which may occur while allocating memory for data structures. The descriptive error message string could be obtained using `strerror` function.

2. Libpcap errors — the group of errors, which may occur while using libpcap routines ("no suitable device found", "bad dump file format", , etc.). The descriptive error message string could be obtained using `pcap_geterr` function.

The descriptive error message string should be passed to error processing module. The log reader itself should not output any error messages.

Expandability

The log reader module provides expandability in the following ways:

1. Adding new link layer type.

   Each link layer type has its own callback function. Thus, the callback function should be created for processing new link layer type.

2. Adding new protocol.

   Each protocol should have own function, which processes the data related to this protocol. Thus, the function should be created for processing new protocol related data.

Implementation notes

- The log reader consists of two files: `logread.h` and `logread.c`. The file `logread.h` contains the routines declarations. The file `logread.c` contains the routines code.

6.4 Message mapper

Description

This module is responsible for linking and merging data from packet trace log files. The PTP structures (see section 5.2) which was read by Log reader module (see section6.3) are the input data for Message mapper module. First, the corresponded messages from both PTP structures are links. Second, both PTP structures are merged into the one PTUS list (see section 5.4).

For links messages Message mapper module uses IP identifier for IP protocol and ARP packet content for ARP protocol.

Algorithms

Step 1 establishes links between PTU structures for each PTP:

(a) takes the first PTP, and selects there the first PTU.

(b) If the current PTU doesn't represented as a sent package, then goes to step (e).

(c) searches in another PTP similar PTU.

(d) If such PTU was founded, then establishes link from the founded PTU to current PTU.

(e) If next PTU is exists, then selects this PTU and goes to step (b).

(f) If next PTP is exists, then selects this PTP, selects the first PTU in current PTP, and goes to step (b).

Step 2 Merges PTP structures to the one PTUS list ordered by timestamps:

(a) Selects first PTP and the first PTU in current PTP.

(b) If in next PTP first PTU has timestamp less than in selected PTU then selects current PTP and the first PTU in current PTP.

(c) If next PTP is exists then goes to step (b).

(d) Adds selected PTU to PTUS, sets the next after selected PTU PTU as first, removes selected PTU from selected PTP, goes to step (a).

Functions

- Function:
  `struct ptu *map_messages ( struct ptu **ptps )`
  This function sets links between messages of PTP structures and merges several PTP structures to PTU structure that sorts in chronology order.

  Arguments:
  - `struct ptu **ptps` is an array of PTP structures.

  Returns:
  This function returns PTUS structure.

Implementation Notes

- The *Message Mapper* consists of two files: `msgmap.h` and `msgmap.c`. The first file contains declarations of routines and second file contains variables and routines code.

Possible types of errors

The message mapper errors can be divided into following groups:

- Link errors — the group of errors, which may occur while mapping messages. For example “Not equal IP identifiers”.
- Merge errors — the group of errors, which may occur while merging messages. For example “Message contains bad timestamp”.

References on user requirements

- 'Supported protocols', [10].
- 'IP reordering', [10].
- 'Time difference', [10].
- 'Message delay', [10].
- 'IP version', [10].

6.5 Events calculator

Events calculator is responsible for calculating necessary protocol variables and TCP states. After message mapping there is a one **PTU sequence** which is a list of messages sorted by timestamps with links for corresponding messages.

First objective of the Events calculator is to split each **packet trace unit** on events (See **Layer splitter** subsection) by layers and build **Events sequence**, then the second is to go through this sequence and calculate TCP states and protocol variables (See **Calculator** subsection) and also select flows from PTU sequence .

Work of Events calculator and changing data structures are shown on Figure 17.

6.5.1 Layer splitter

Description

This submodule gets **PTU sequence** and splits it to **Events sequence**. Also it fills links, flows lists and add hosts in the hosts list. One of the main objectives of this submodule is to find and process IP fragmentation. Also it is important for Layer splitter to consider an encapsulation for packets. Common scheme of Layer splitter can be divided on two cases:

1. Basic case, when the packet trace unit does not contain IP fragmentation information. Each packet trace unit in PTU sequence contains protocol information which is separated on different layers, for example TCP, IP, UDP and other headers. The objective of Layer splitter is to split each packet trace unit on sequence of events, which contain information about protocols separately, See Figure 18.

2. Case, when the packet trace unit contain information about IP fragmentation. In that case the second objective of Layer splitter is to place fragmentation links after splitting. For example there are three packet trace units, which are received in different order (see Figure 19). Then Layer splitter divides these PTUs into events, but UDP packet will be placed as last event in corresponding sequence of events. Also Layer splitter adds fragmentation links.

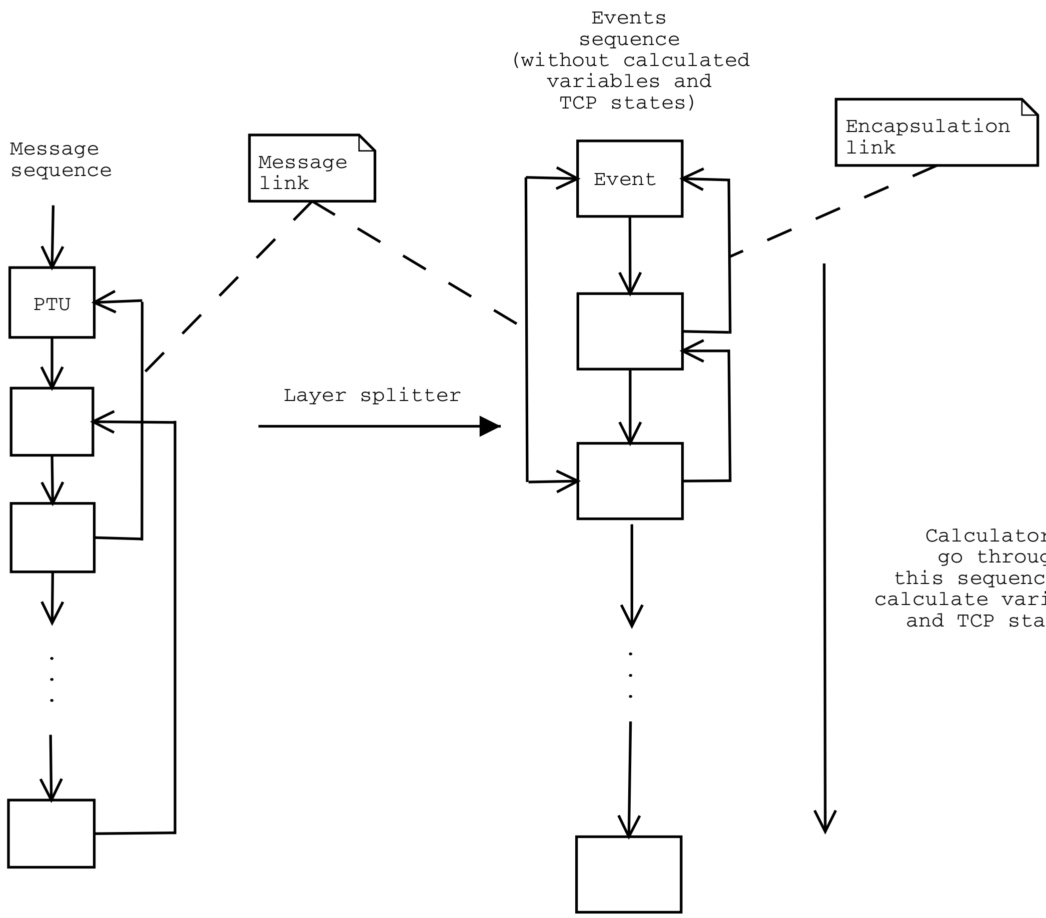

Figure 17: Principles of operation of Events calculator

Variables and Macros

These variables are used in description of the splitting algorithm and can be changed or defined more exactly for convenience of the implementing of the algorithm.

- Variable:

  `event *restore`

  The events sequence which will be filled by new events (NULL by default).

- Variable:

  `ptu *last`

  The last added fragment.

See **Algorithms** section for details on how these variables are used.

Functions

- Function:

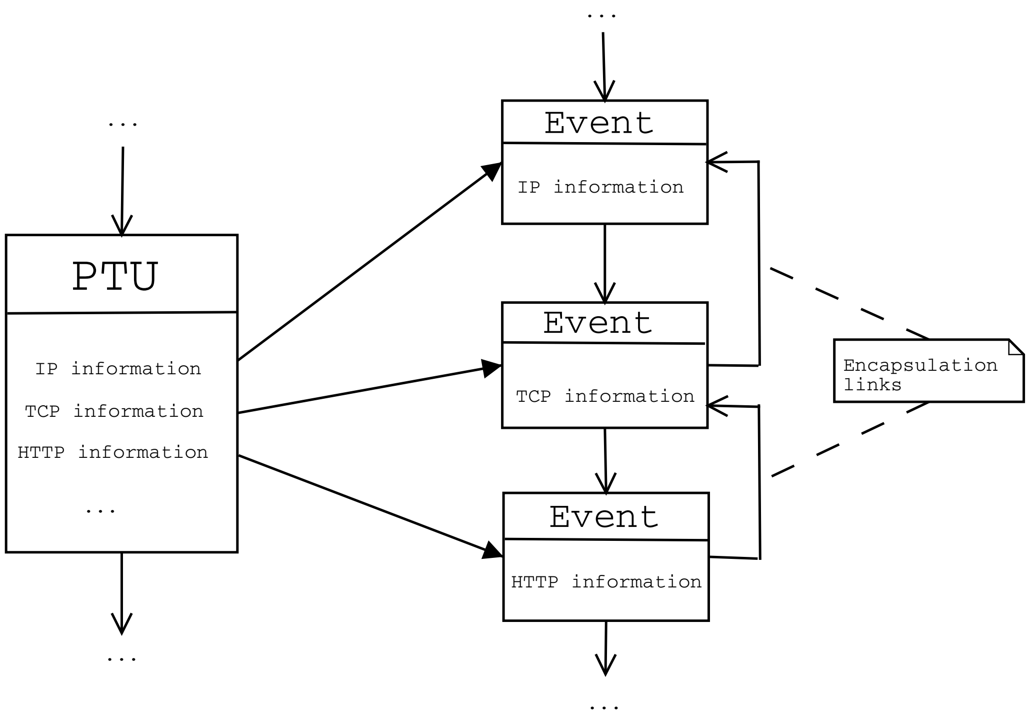

Figure 18: Packet trace unit splitting

```
void split (struct ptu *ptu_sequence,
            struct event **events_sequence
            struct host *hosts,
            struct link **links,
            struct flow **flows)
```

This function is used to split PTU sequence on events sequence and placing fragmentation packets and links, fills flows and links lists and also adds hosts in hosts list.

Arguments:

- `struct ptu *ptu_sequence` is the internal structure of the Analyzer. This is a list which contains information from the tcpdump logs and some additional fields. These structures must be processed with **Message mapper** module.
- `struct event **events_sequence` is the list of events.
- `struct host *hosts` is the list of links.
- `struct link **links` is the list of links.
- `struct flow **flows` is the list of flows.

Algorithms

This algorithm is used for splitting PTU sequence to Events sequence. See **Variables and macros** section above for variables explanations.
Notice: It is considered that all non-fragmented units consist of one fragment.

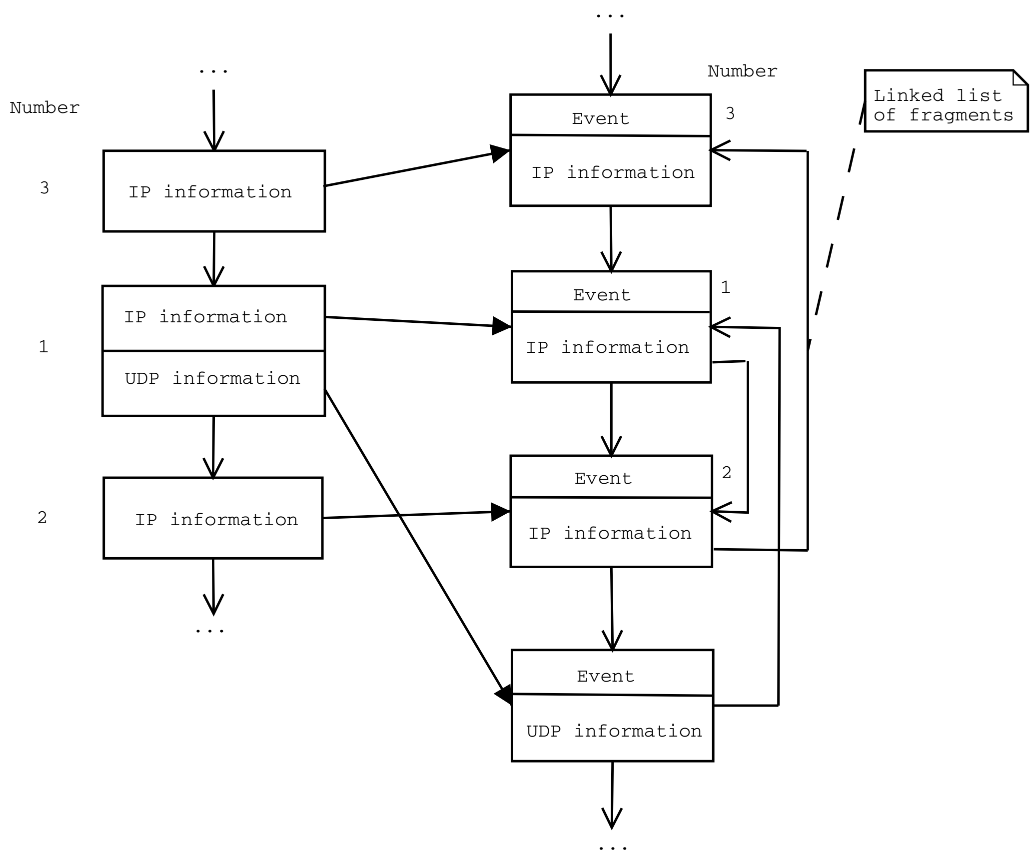

Figure 19: Fragmentation splitting

`FOR` each PTU in PTU sequence `DO`

1. `last` := new event, which is corresponded to lowest layer in PTU
2. Add `last` in Events sequence
3. `IF` PTU has headers of other layers `THEN` add corresponding events in `restore`
4. `IF last` is the last fragment `THEN`
   - Find element(`temp`) in `restore`, which has `last` as the last fragment
   - `temp`.timestamp := `last`.timestamp
   - `last := temp`
   - Remove last from `restore`
   - Add `last` in Events sequence
   - goto 4 (Check condition for new `last`)

Possible types of errors

- Layer splitter can't split packet trace unit on events
- Splitter can't place fragmentation links

6.5.2 Calculator

Description

This submodule is responsible for calculating some necessary variables and TCP states after splitting PTU sequence on events. Also it calculates and adds events with type UNIT_LOST ('dropped' events) in the events sequence. It gets Events sequence as input data and go through it (may be a few times) for calculating and filling variables structures.The first version of analyzer will support a minimal set of these calculations.

Variables and Macros

- Variables:

```
struct  link {
        host *host1   — First host
        host *host2   — Second host
        link *next    — Second host
        double MTU    — MTU value (IP, UDP protocols)
        char *id      — Id of the host
}
```

  — A list of physical links between hosts. It is used for storing variables, which are relevant to the physical link, for example MTU value.

```
struct  variables {
        int tcp_state       — current TCP state for event
                              ('TCP_STATE_UNKNOWN' by default,
                              TCP protocol)
        double cwnd         — Congestion window (TCP protocol)
        double ssthresh     — Slow start threshold size (TCP protocol)
        double dupack       — Duplicated ack threshold (TCP protocol)
        timeval tt          — Transition time (TCP protocol)
        double RTO          — Retransmission TimeOut (TCP protocol)
}
```

  — The internal structure for each event, which contains fields for storing protocol variables and TCP states.

- Macro:
  - `TCP_STATE_*`
    A symbolic TCP state name which describes current TCP state — `TCP_STATE_UNKNOWN, TCP_STATE_SYN_SENT, TCP_STATE_SYN_RCVD, TCP_STATE_ESTABLISHED, TCP_STATE_CLOSED`.

Functions

- Function:
```
void calculate (  struct ptu *ptu_sequence,
                  struct event **events_sequence,
                  struct link **links,
                  struct flow **flows,
                  char const *add_log_var)
```
  This function is used to calculate necessary protocol variables and TCP states, filling links and flows lists, adding 'dropped' events.

  Arguments:

  - `struct ptu *ptu_sequence` is the PTU sequence.
  - `struct event **events_sequence` is the Events sequence. These structures will be processed with **Layer splitter** module.
  - `struct links *links` is the list of links.
  - `struct flows *flows` is the list of flows (TCP or UDP).
    For calculating variables may be it will be needed to going through Events sequence a few times.

Algorithms

- TCP states algorithms (implementation of this feature in first version of the Analyzer is optional):
  - Check previous TCP state.
  - Check packet's flags.
  - IF flag of send packet == SYN THEN `TCPstate` = `TCP_STATE_SYN_SENT`
  - IF flag of received packet == SYN THEN `TCPstate` = `TCP_STATE_SYN_RCVD`
  - IF previous sequence of TCP states corresponds three-way handshake or segment will be send or received without SYN, ACK, FIN, RST flags THEN `TCPstate` = `TCP_STATE_ESTABLISHED`
- Exchange states algorithms (see **Events sequence** subsection in **Data structures** section):
  - IF PTU was sent THEN `type` = `UNIT_SENT`
  - IF PTU was received THEN `type` = `UNIT_RECEIVED`
  - IF reference on next PTU of unit is NULL THEN `type` = `UNIT_LOST`
- Calculating of protocol variables:

Network	**MTU(bytes)**
Hyperchannel	65535
16 Mbits/sec token ring (IBM)	17914
4 Mbits/sec token ring (IEEE 802.5)	4464
FDDI	4352
Ethernet	1500
IEEE 802.3/802.2	1492
X.25	576
Point-to-Point (low delay)	296

Table 1: Standard MTU values

- MTU (Maximal transmission unit) can not be calculated in this version of the Analyzer.
  Some typical MTU's are shown in Table 1.

- cwnd (Congestion window) can not be calculated in this version of th Analyzer
- ssthresh (Slow start threshold) can not be calculated in this version of the Analyzer.
- RTO (Retransmission TimeOut); an empirical value of this variable can be calculated as difference between times of sending segments with the same sequence numbers(without sending triple acknowledgments).

- dupack (Duplicated ack threshold) can not be calculated in this version of th Analyzer.
- tt (Transition time) can calculated as time difference between send event and receive event.
- droptime (Unit lost time) of packet `n` can be calculated as
  `(rec(n-1)-send(n-1))+(rec(n+1)-send(n+1))/4`

  In multiple dropping: find previous successful sent packet, and the next next previous successful sent packet. Take the average of the flight sizes of these packets. Then divide that by 2.

  There nothing but dropped packets:
  `droptime(n)=(send(n)+send(n+1))/2`
  except for the last packet:
  `droptime(n)=send(n)+(send(n)-send(n-1))/2`

  Timestamp of the 'dropped' event is a `droptime`.

The Events calculator should gets these variables from the additional logs (optional implementation priority). In the first version of the Analyzer protocol variables calculated on minimal level.

Possible types of errors

- Wrong calculating variables and TCP states
- Impossible calculations

Implementation Notes

- The Events calculator consists of four group of files: common group contains routines and macros, which are used by both Calculator and Layer splitter (`ec_common.c`, `ec_common.h`); groups of files, which describes routines and macros for each submodule (`eventcalc.c, eventcalc.h, split.c, split.h`); and the group of files, which are used by the Layer splitter and provides protocols specific functions (`ec_ip.c, ec_ip.h, ec_tcp.c, ec_tcp.h`, etc). The `calculate` routine run `split` routine.

References on user requirements

- 'Calculations', [10].
- 'IP reordering', [10].

6.6 Protocol Events File Writer

Description

This module writes the analyzed protocol communication data represented in the internal analyzer data structure into Protocol Events File.

Functions

- Function:

  ```
  void pef_write (  FILE *out,
                    struct event *events,
                    struct host *hosts,
                    struct link *links,
                    struct flow *flows
  ```

  This function writes the analyzed protocol communication data to the specified output stream using the Protocol Events Format.

  Arguments:

  – `FILE *out` specifies where to output the data.
  – `struct event *events` is the ready Events sequence.

- `struct host *host` is the filled list of hosts.
- `struct link *links` is the filled list of physical links.
- `struct flow *flows` is the filled list of TCP or UDP flows.

- Function:

```
void pef_writef (  char *filename,
                   struct event *events,
                   struct host *hosts,
                   struct link *links,
                   struct flow *flows)
```

  This function writes the analyzed protocol communication data to the specified file using the Protocol Events Format.

  Arguments:

  - `char *filename` specifies the name of the output file.
  - `struct event *events` is the ready Events sequence.
  - `struct host *host` is the filled list of hosts.
  - `struct link *links` is the filled list of physical links.
  - `struct flow *flows` is the filled list of TCP or UDP flows.

  Implementation notes:
  This function opens the specified file and calls `pef_write()` function to write the data.

Possible types of errors

The PEF Writer errors are following:

1. I/O Errors — the group of errors, which may occur while opening and writing output file. The descriptive error message string could be obtained using `strerror ()`. function.

2. Incorrect data — the group of errors, which may occur while writing PEF. The descriptive error message string could be obtained using `strerror ()`.

The descriptive error message string should be passed to error processing module (see section 6.7). The PEF Writer itself should not output any error messages.

Implementation notes

- The PEF Writer consists of four files: `pefwrite.h, pefwrite.c`, which contain the routines declarations and code for writing PEFs, and `pefwrite_http.c`, `pefwrite_http.h`, which contain routines for parsing and writing HTTP messages.

References on user requirements

- ’Supported protocols’, [10].
- ’IP reordering’, [10].
- ’Tcpdump logs’, [10].
- ’Calculations’, [10].

6.7 Error processing module

Description

This module is responsible for processing different types of errors occurred while running the Analyzer. It is a part of the user interface, which is intended for showing messages containing error descriptions. All the Analyzer modules should call only the error processing module routines on errors, and shouldn’t use any other ways for processing errors and showing messages. After processing the error and showing the error message the program should be correctly terminated. The module is designed to provide maximum extensibility during the implementation phase.

Error message format

If an error has been occurred while analyzing packet trace files, the user should be notified with a message containing the error description. The standard error message format is used:

```
<program_name>: <description>
```

or

```
<program_name>: <cause>: <description>
```

The `<program_name>` is the name that was used to invoke the program running in the current process, with directory names removed.

The `<cause>` is the name of the entity, which is the direct cause of the error (the file, which can’t be opened, the module, which can’t calculate something, etc.).

The `<description>` is a string containing the error description.

Variables and Macros

- Variable:

  ```
  const char *err_list[]
  ```

  The list of the Analyzer specific error message strings, for example “IP address not found”.

- Macro:

  ```
  ERR_*
  ```

  The symbolic error name for an error, for example `ERR_IP_NOT_FOUND` for the message string, which tells that IP address was not found. The macro should expand to an integer, which is corresponding error message string index in the `err_list[]`.

  ```
  WARN_*
  ```

  The symbolic error name for an warning, for example `WARN_NOT_NOT_SENT` for the message string, which tells that received unit was not sent. The macro should expand to an integer, which is corresponding error message string index in the `err_list[]`.

- Variable:

  ```
  const int err_num
  ```

  The number of strings in `err_list[]`. This variable is used by the function `anlz\_strerror` to detect invalid arguments.

See **Expandability** section for details on how to define and use the variables.

Functions

- Function:

  ```
  void error (int status, const char *cause,
              const char *descr)
  ```

  This function is used to display the error message according to the error message format and terminate the program with the exit status specified.

  Arguments:

  – `int status` is the exit status of the program. If its value is nonzero, the program will call `exit` with the `STATUS` value as its parameter. Using of predefined `EXIT_*` macros is preferred. If the value is `EXIT_SUCCESS` or 0, the program will not be terminated. It is useful when printing usage or help information after error reporting.

- `const char *cause` specifies the `<cause>` in the error message format. It may be a name of file, module, function, etc. If its value is NULL the error message will contain no `<cause>`.
- `const char *descr` is the `<description>` in the error message format. If the `descr` is NULL, "Unknown error" string will be displayed. The module, which calls this function, should not use any hand-made error description strings. The function `strerror` should be used for standard errors, `pcap_geterr` for libpcap errors, etc. The function `anlz_strerror` should be used only if there is no suitable descriptive message string for specific analyzer error. See **Expandability** for details on how to define new error and error message string for using with `anlz_strerror`.

```
void warning (const char *cause,
              const char *descr)
```

This function is used to display the warning message according to the error message format.

Arguments:

- `const char *cause` specifies the `<cause>` in the error message format. It may be a name of file, module, function, etc. If its value is NULL the error message will contain no `<cause>`.
- `const char *descr` is the `<description>` in the error message format. If the `descr` is NULL, "Unknown error" string is displayed.
  The module, which calls this function, should not use any hand-made error description strings. The function `strerror` should be used for standard errors, `pcap_geterr` for libpcap errors, etc. The function `anlz_strerror` should be used only if there is no suitable descriptive message string for specific analyzer error. See **Expandability** for details on how to define new error and error message string for using with `anlz_strerror`.

- Function:

```
consty char *anlz_strerror (int err)
```

This function maps the error code specified by its argument to a descriptive error message string. The error is supposed to be a specific Analyzer error.

Arguments:

- `int err` specifies the error code, which needs to be mapped to a string. The module, which calls this function, should use any of defined `ERR_*` macros to get appropriate error message string instead of specifying explicit integer value. If the value is invalid, `error` function should be called.

Returns:

The return value of the function is a pointer to a descriptive error message string. The string returned by the function should not be modified. If `err` is invalid, "Unknown error" string is returned.

Expandability

The error processing module provides opportunity to expand error list. Expanding error list means constructing new descriptive error message string and corresponding macro for specific analyzer error, which has no appropriate standard error message. The process of new error definition consists of the following steps:

1. Construct new error message string containing distinct description of the error. Note, that the whole error message including program name and cause should not be greater than one line.
2. Add the constructed string to the end of `err_list`.
3. Define new unique `ERR_NEW_ERROR` macros expanding to an integer, which is the constructed string index in `err_list[]`.
4. Increase `err_num` to 1. It should contain the correct number of `err_list[]` strings.

After that the `anlz_strerror` should be used to get the descriptive error message string. That extensible system should be used during the implementation phase.

Implementation Notes

- The error processing module consists of two files: `error.h` and `error.c`. The file `error.h` contains symbolic error names and the routines declarations. The file `error.c` contains the list of error message strings and the routines code.
- `extern char* program_invocation_short_name` variable could be used to access program name with directory names removed.
- `stderr` should be used for printing error messages.

7 Behavioral model

7.1 Produce a protocol events file

This is major action scenario for the Analyzer. A user can get Protocol events file from two packet trace files. “Produce a protocol events file” behavior is shown in Figure 20.

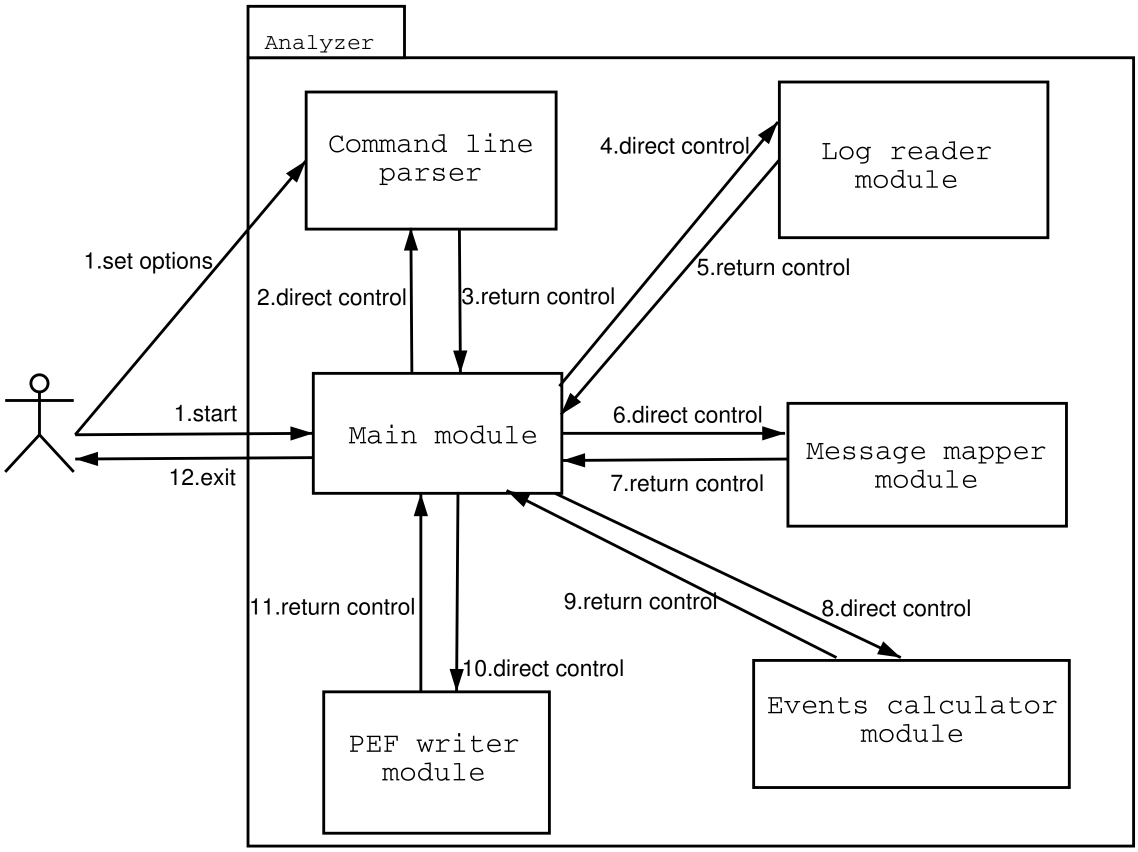

Figure 20: Produce a protocol events file Collaboration Diagram

1. User starts program with given parameters;
2. The Main module directs the control to the Command line parser module for parsing command line options;
3. The Command line parser module returns the control to the Main module;
4. The Main module directs the control to the Log reader module for reading packet trace files;
5. The Log reader module returns the control to the Main module;
6. The Main module directs the control to the Message mapper module for mapping messages;
7. The Message mapper module returns the control to the Main module;
8. The Main module directs the control to the Events calculator module for calculating events and protocol variables;
9. The Events calculator returns the control to the Main module;

10. The Main module directs the control to the Protocol events file writer module for writing protocol events file;

11. The Protocol events file writer module returns the control to the Main module;

12. The program exit successfully.

7.1.1 Get command line parameters

User inputs command line parameters. Analyzer parses this parameters and sets global variables to user specified values (see sect. 6.2). "Parse command line parameters" behavior is shown in Figure 21.

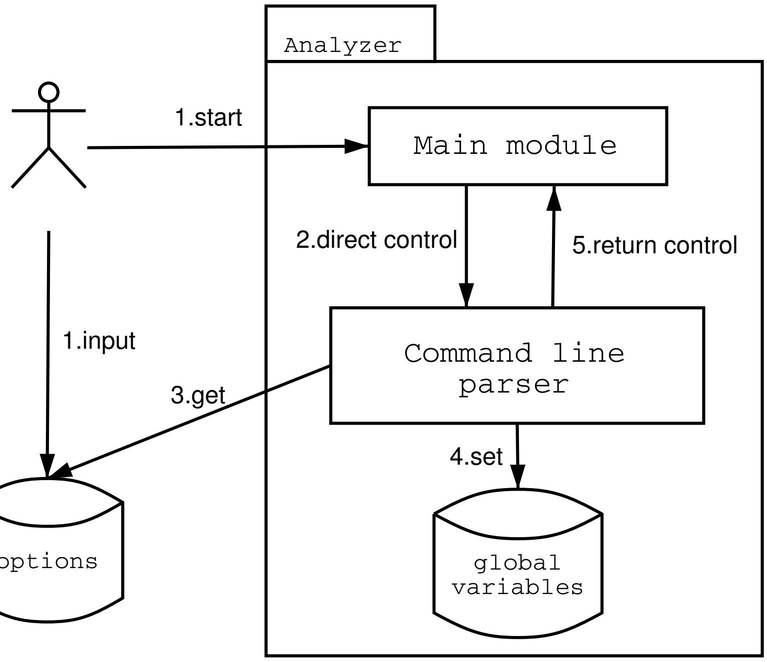

Figure 21: Get command line parameters Collaboration Diagram

1. User inputs command line parameters;

2. The Main module directs the control to the Command line parser module for parsing command line options;

3. The Command line parser module gets command line parameters;

4. The Command line parser module sets global variables;

5. The Command line parser module returns the control to the Main module.

7.1.2 Read packet trace files

User gets two packet trace files in binary format. The Analyzer reads these files and fills internal data structures (see sect. 6.3). “Read packet trace files” behavior is shown in Figure 22.

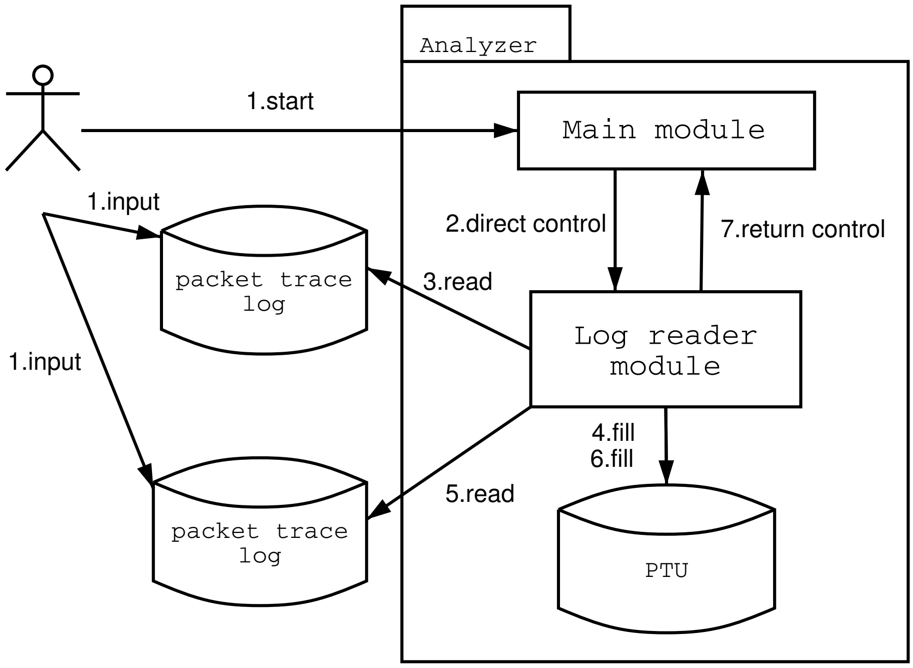

Figure 22: Read packet trace files Collaboration Diagram

1. User gets two packet trace files;
2. The Main module directs the control to the Log reader module for reading packet trace files;
3. The Log reader module reads first packet trace file;
4. The log reader module fills internal data structure;
5. The Log reader module reads second packet trace file;
6. The Log reader module fills internal data structure;
7. The Log reader module returns the control to the Main module.

7.1.3 Message mapping

After reading packet trace files the Analyzer adds links between packet trace units (see sect. 6.4). “Message mapping” behavior is shown in Figure 23.

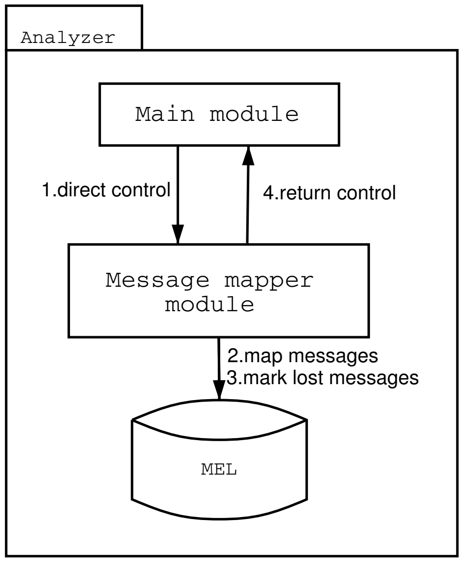

Figure 23: Message mapping Collaboration Diagram

1. The Main module directs the control to the Message mapper module for mapping messages;
2. The Message mapper module finds links between messages;
3. The message mapper module marks remain messages as lost message;
4. The Message mapper module returns the control to the Main module.

7.1.4 Calculate events

The Analyzer gets all data for all protocols and calculates protocol variables and events (see sect. 6.5). "Calculate events" behavior is shown in Figure 24.

1. The Main module directs the control to the Events calculator module for calculating events;
2. The Events calculator module directs the control to the Layer splitter module to splitting messages.
3. The Layer splitter module reads PTUS;
4. The Layer splitter module splits messages;
5. The Layer splitter module returns the control to the Events calculator module;
6. The Events calculator module calculates protocol variables and events;
7. The Events calculator module returns the control to the Main module.

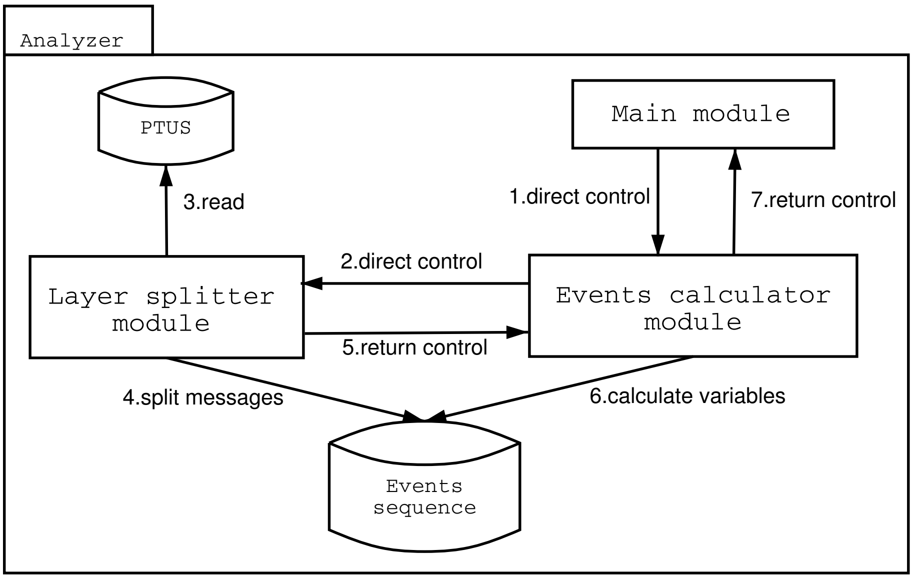

Figure 24: Calculate events Collaboration Diagram

7.1.5 Write PEF

After data processing the Analyzer writes all data to the Protocol events file (see sect. 6.6). "Write PEF" behavior is shown in Figure 25.

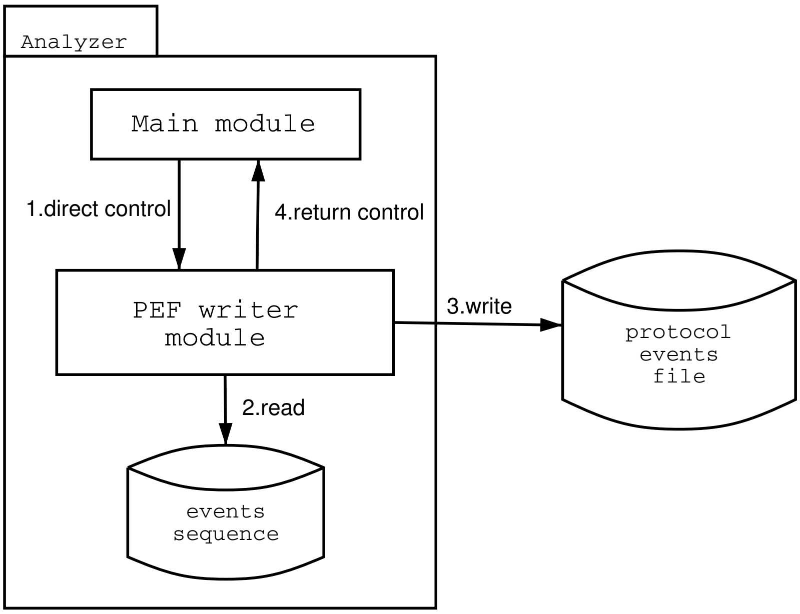

Figure 25: Write PEF Collaboration Diagram

1. The Main module directs the control to the PEF writer module for writing protocol events file;

2. The PEF writer module reads data from internal data structures;

3. The PEF writer module writes protocol events file;

4. The PEF writer returns the control to the Main module.

7.2 Get usage info

User can get brief documentation for how invoke the program (see sect. 6.2). “Get usage info” behavior is shown in Figure 26.

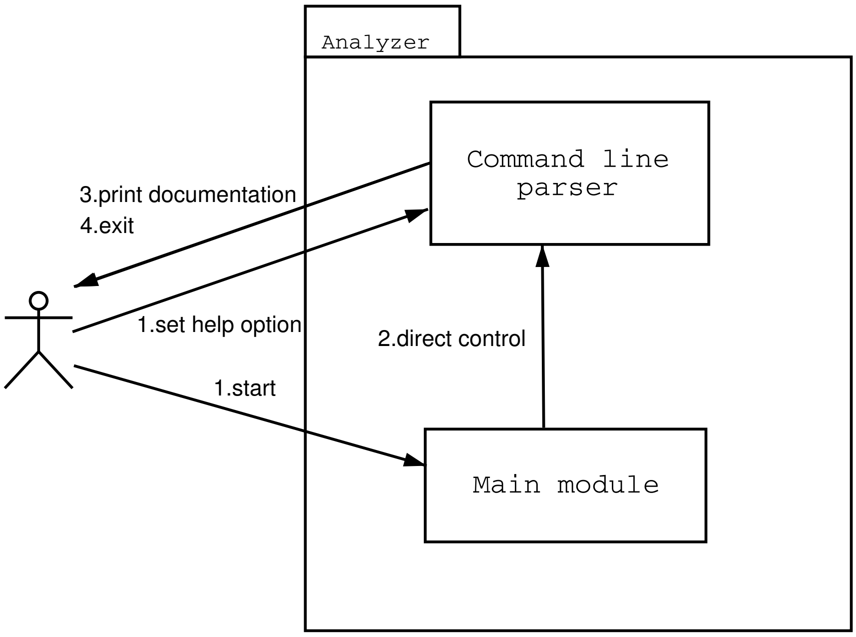

Figure 26: Get usage info Collaboration Diagram

1. User starts the program with option “`-h`” or “`--help`”;

2. The Main module directs control to the Command line parser module for parsing command line options;

3. The Command line parser module prints brief documentation to standard output;

4. Program exit successfully.

7.3 Get program info

User can get information about program name, version, origin and legal status (see sect. 6.2). “Get program info” behavior is shown in Figure 27.

1. User starts program with option “`-v`” or “`--version`”;

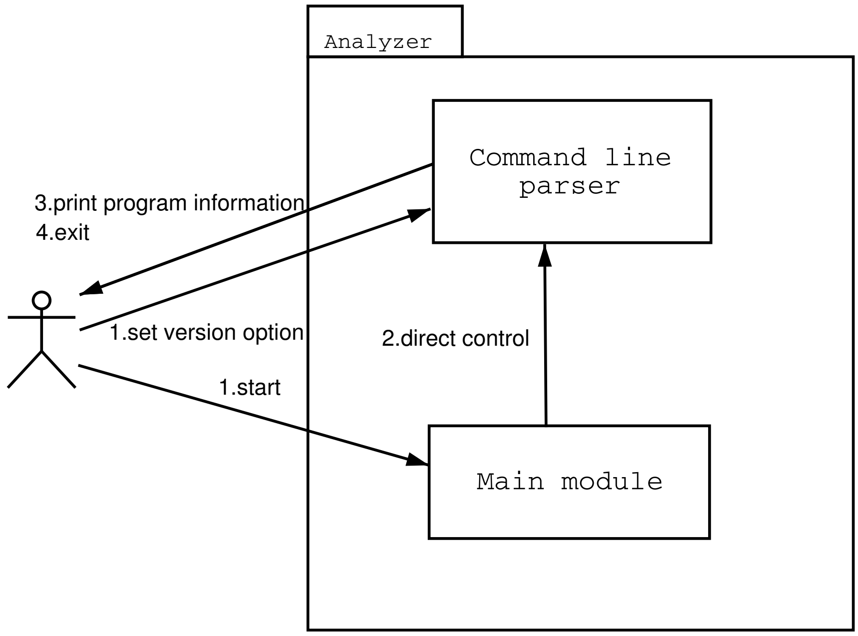

Figure 27: Get program information Collaboration Diagram

2. The Main module directs control to the Command line parser module for parsing command line options;

3. The Command line parser module prints information about its name, version, origin and legal status to standard output;

4. Program exit successfully.

7.4 Abnormal program termination

There are several scenarios for abnormal program termination. Each scenario represents modules work when some data was wrong.

7.4.1 Wrong command line options

User can input wrong command line options (see sect. 6.2). “Wrong command line options” behavior is shown in Figure 28.

1. User starts the program with wrong command line options;

2. The Main module directs the control to the Command line parser module for parsing command line parameters;

3. The Command line parser module finds wrong command line options and sends error message to the Error processing module;

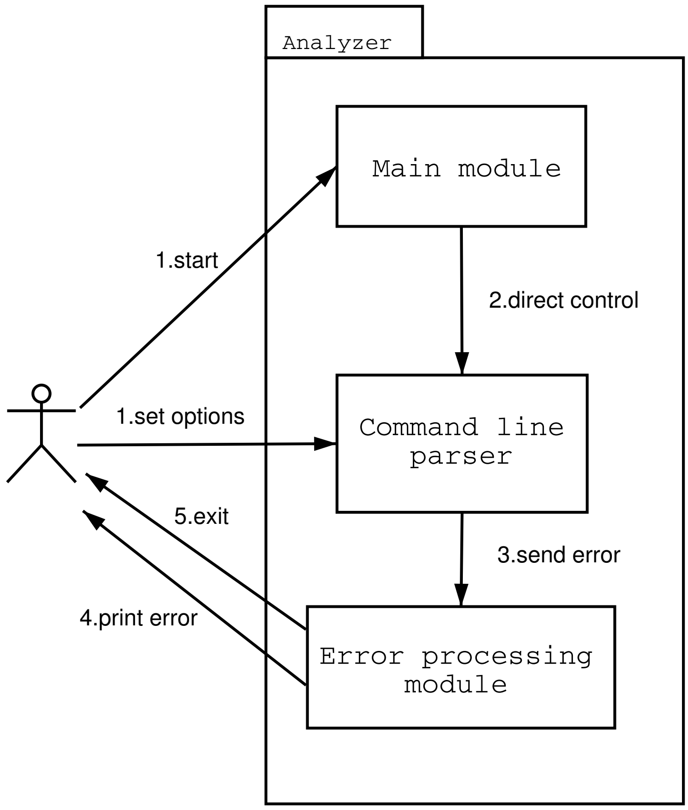

Figure 28: Wrong command line options Collaboration Diagram

4. The Error processing module prints error description;

5. The Program exit abnormally.

7.4.2 Wrong command line syntax

User can use wrong command line syntax for input command line options (see sect. 6.2). “Wrong command line syntax” behavior is shown in Figure 29.

1. User starts the program with wrong command line options;

2. The Main module directs the control to the Command line parser module for parsing command line parameters;

3. The Command line parser module finds wrong command line syntax;

4. The Command line parser module prints error description;

5. The Program exit abnormally.

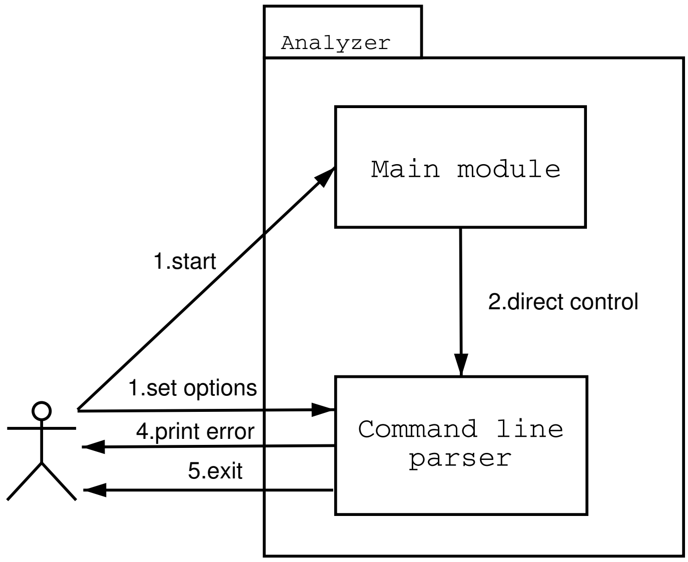

Figure 29: Wrong command line syntax Collaboration Diagram

7.4.3 Wrong reading packet trace files

User can get wrong packet trace files. For example this files are not exists or has wrong format (see sect. 6.3). “Wrong reading packet trace files” behavior is shown in Figure 30.

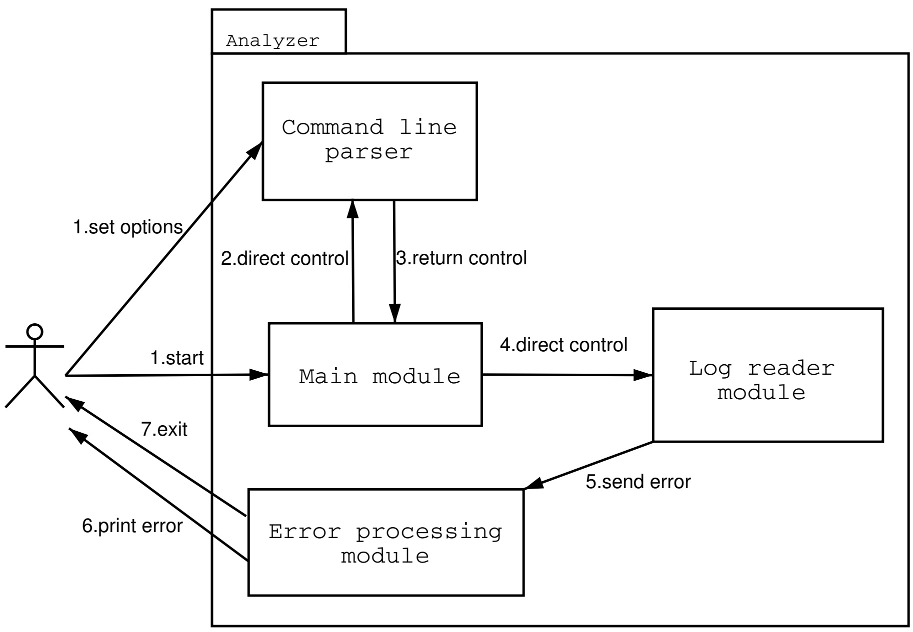

Figure 30: Wrong reading packet trace files Collaboration Diagram

1. User starts the program with corresponded parameters;

2. The Main module directs the control to the Command line parser module for parsing command line parameters;

3. The Command line parser module returns the control to the Main module;

4. The Main module directs the control to the Log reader module for reading packet trace files;

5. The Log reader module finds error in reading process and sends error message to the Error processing module;

6. The Error processing module prints error description;

7. The Program exit abnormally.

7.4.4 Wrong mapping messages

In some cases The Analyzer can not map messages. For example message contain wrong time stamp or wrong data (see sect. 6.4). “Wrong mapping messages” behavior is shown in Figure 31.

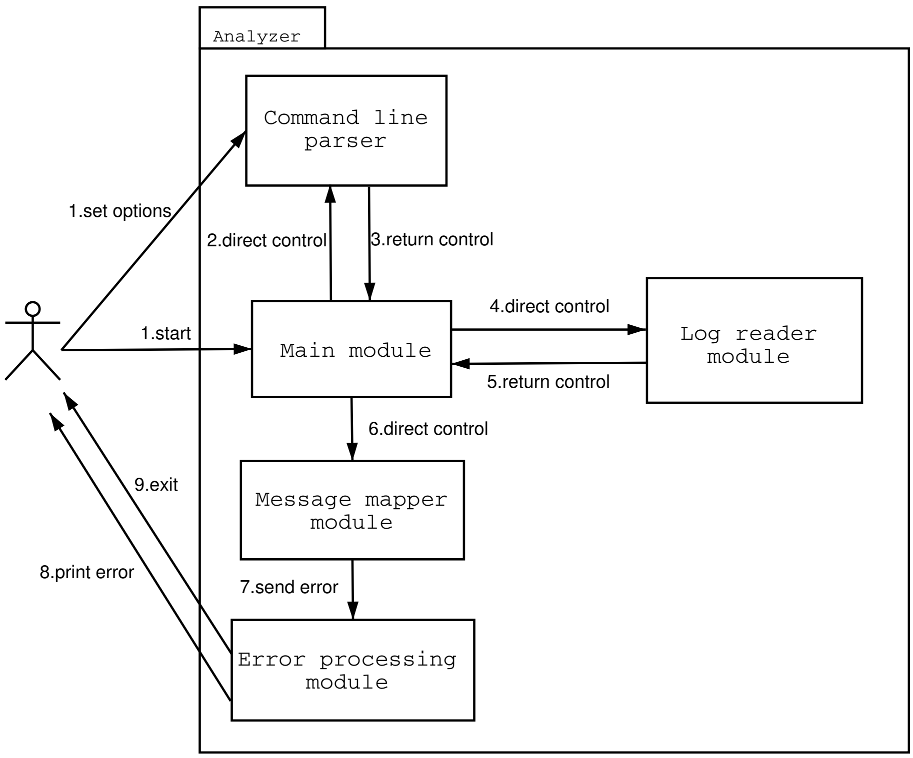

Figure 31: Wrong mapping messages Collaboration Diagram

1. User starts program with corresponded parameters;

2. The Main module directs the control to the Command line parser module for parsing command line parameters;
3. The Command line parser module returns the control to the Main module;
4. The Main module directs the control to the Log reader module for reading packet trace files;
5. The Log reader module returns the control to the Main module;
6. The Main module directs the control to the Message mapper module for mapping messages;
7. The Message mapper module finds error in mapping messages and sends error message to the Error processing module;
8. The Error processing module prints error description;
9. The Program exit abnormally.

7.4.5 Wrong calculating events

In some cases Analyzer can not calculate protocol variables and events. For example messages contains wrong data (see sect. 6.5). “Wrong calculating events” behavior is shown in Figure 32.

1. User starts program with corresponded parameters;
2. The Main module directs the control to the Command line parser module for parsing command line parameters;
3. The Command line parser module returns the control to the Main module;
4. The Main module directs the control to the Log reader module for reading packet trace files;
5. The Log reader module returns the control to the Main module;
6. The Main module directs the control to the Message mapper module for mapping messages;
7. The Message mapper module returns the control to the Main module;
8. The Main module directs the control to the Events calculator module for calculating events and protocol variables;
9. The Events calculator module finds an error and sends error message to the Error processing module;
10. The Error processing module prints error description;
11. The Program exit abnormally.

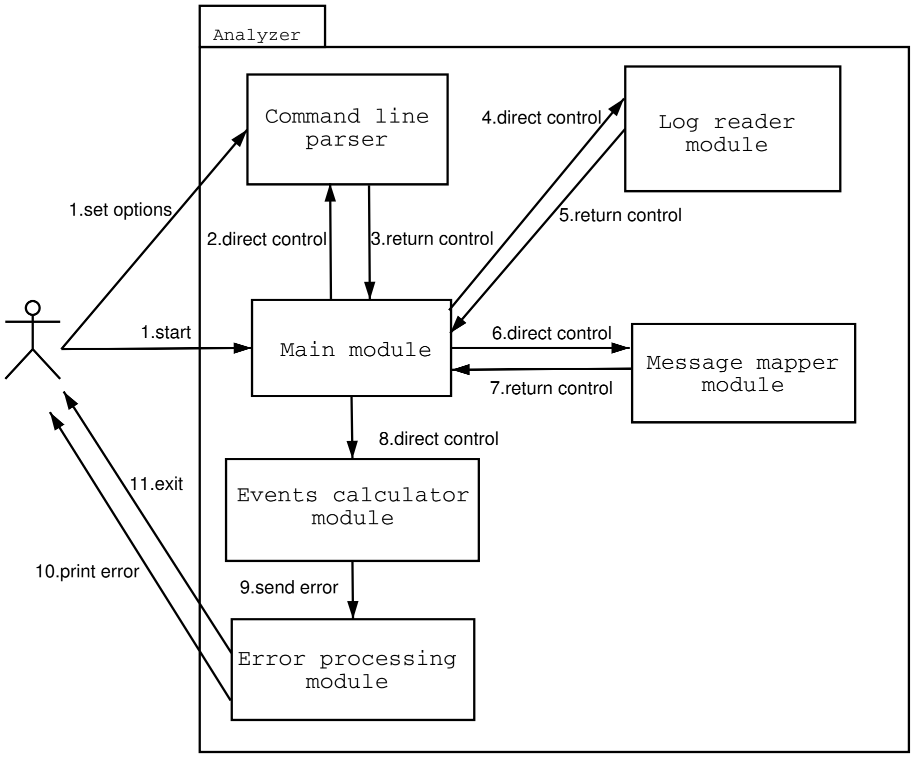

Figure 32: Wrong calculating events Collaboration Diagram

7.4.6 Wrong recording PEF file

In some cases Analyzer can not write Protocol events file. For example Analyzer don't have necessary permissions or don't have enough space in recording device (see sect. 6.6). "Wrong recording PEF" file behavior is shown in Figure 33.

1. User starts program with corresponded parameters;
2. The Main module directs the control to the Command line parser module for parsing command line parameters;
3. The Command line parser module returns the control to the Main module;
4. The Main module directs the control to the Log reader module for reading packet trace files;
5. The Log reader module returns the control to the Main module;
6. The Main module directs the control to the Message mapper module for mapping messages;
7. The Message mapper module returns the control to the Main module;
8. The Main module directs the control to the Events calculator module for calculating events and protocol variables;

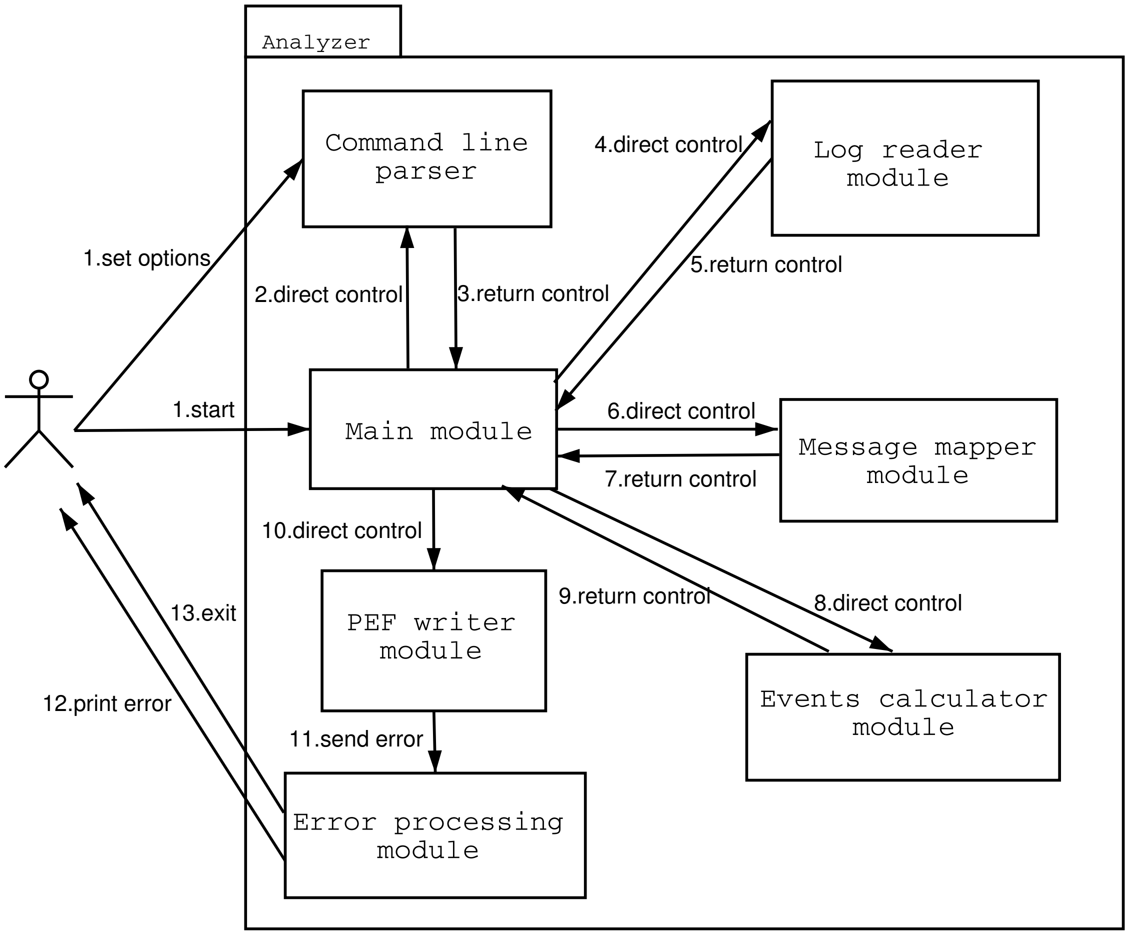

Figure 33: Wrong recording PEF file Collaboration Diagram

9. The Events calculator returns the control to the Main module;
10. The Main module directs the control to the Protocol events file writer module for writing protocol events file;
11. The Protocol events file finds error in recording process and sends error message to the Error processing module;
12. The Error processing module prints error description;
13. The program exit abnormally.

8 Configuration and installation

8.1 Configuration

The global Analyzer constants and defenitions includes in file `config.h`. This file contains:

- XML version;
- tcpdump binary format version;
- Analyzer version;
- Compiler directives.

8.2 Installation

The utilities "configure" and "make" used for installation process. "Configure" finds necessary software tools and libraries and sets correct values for system-dependent variables. As a result configure creates "Makefiles" for "make" utility. The "make" builds and installs analyzer software.

The "configure" has the following options:

- `--prefix=PREFIX` – install architecture-independent files in PREFIX directory. Default value is `/usr/local`;
- `--exec-prefix=EPREFIX` – install architecture-dependent files in EPREFIX directory. Default value is PREFIX.

The "make" has the following options:

- `all` — build Analyzer software;
- `install` — install Analyzer software;
- `uninstall` – uninstall Analyzer software;
- `dist` – create distributive for Analyzer software;
- `clean` — delete executing file and object files.

The standard installation process is following.

- Run "configure";
- Run "make";
- Run "make install".

The following typical software needs for build and install Analyzer:

- gcc compiler(with support ANSI and POSIX standards) version 3.2;
- Linux kernel version 2.4.21;
- Libpcap version 0.7.

References

1 Berners-Lee T., Fielding R., Frystyk H., *Hypertext Transfer Protocol – HTTP/1.0*. RFC 1945. Network Working Group, May 1996.

2 Plummer David C., *An Ethernet Address Resolution Protocol*. RFC 826. Network Working Group, November 1982.

3 Fielding R., Gettys J., Mogul J., Frystyk H., Masinter L., Leach P., Berners-Lee T., *Hypertext Transfer Protocol – HTTP/1.1*. RFC 2616. Network Working Group, June 1999.

4 Postel J., *User Datagram Protocol*. RFC 768. Information Sciences Institute, August 1980.

5 Information Processing Techniques Office, *Internet Protocol*. RFC 791. Information Sciences Institute, September 1981.

6 Information Processing Techniques Office, *Transmission Control Protocol*. RFC 793. Information Sciences Institute, September 1981.

7 Mockapetris P., *Domain Names - Concepts And Facilities*. RFC 1034. Network Working Group, November 1987.

8 Mockapetris P., *Domain Names - Implementation And Specification*. RFC 1035. Network Working Group, November 1987.

9 DaCoPAn Software Engineering project, *Design*. Release 1.0. Universities of Helsinki and Petrozavodsk, April 2004.

10 DaCoPAn Software Engineering project, *Requirements specification*. Release 1.0. Universities of Helsinki and Petrozavodsk, March 2004.

11 Taina J., Korzun D., Tuohiniemi T., Alanko T., Bogoyavlenskiy Y., *Software Engineering Project: Distributed Approach*. Release 1.0. Universities of Helsinki and Petrozavodsk, January 2004.