University of West Bohemia in Pilsen
Department of Computer Science and Engineering
Univerzitni 8
30614 Pilsen
Czech Republic

Log File Analysis

PhD Report

Jan Valdman

Technical Report No. DCSE/TR-2001-04
July, 2001

Distribution: public

Technical Report No. DCSE/TR-2001-04
July 2001

Log File Analysis

Jan Valdman

Abstract

The paper provides an overview of current state of technology in the field of log file analysis and stands for basics of ongoing PhD thesis.
The first part covers some fundamental theory and summarizes basic goals and techniques of log file analysis. It reveals that log file analysis is an omitted field of computer science. Available papers describe moreover specific log analyzers and only few contain some general methodology.
Second part contains three case studies to illustrate different application of log file analysis. The examples were selected to show quite different approach and goals of analysis and thus they set up different requirements.
The analysis of requirements then follows in the next part which discusses various criteria put on a general analysis tool and also proposes some design suggestions.
Finally, in the last part there is an outline of the design and implementation of an universal analyzer. Some features are presented in more detail while others are just intentions or suggestions.

Keywords: log file analysis, Netflow, software components, SOFA

Copies of this report are available on
http://www.kiv.zcu.cz/publications/
or by surface mail on request sent to the following address:

University of West Bohemia in Pilsen
Department of Computer Science and Engineering
Univerzitni 8
30614 Pilsen
Czech Republic

Contents

1 Introduction

1.1 About Log Files

Current software application often produce (or can be configured to produce) some auxiliary text files known as log files. Such files are used during various stages of software development, mainly for debugging and profiling purposes.

Use of log files helps testing by making debugging easier. It allows to follow the logic of the program, at high level, without having to run it in debug mode. [3]

Nowadays, log files are commonly used also at customers installations for the purpose of permanent software monitoring and/or fine-tuning. Log files became a standard part of large application and are essential in operating systems, computer networks and distributed systems.

Log files are often the only way how to identify and locate an error in software, because *log file analysis* is not affected by any time-based issues known as *probe effect*. This is an opposite to an analysis of a running program, when the analytical process can interfere with time–critical or resource–critical conditions within the analyzed program.

Log files are often *very* large and can have complex structure. Although the process of generating log files is quite simple and straightforward, log file analysis could be a tremendous task that requires enormous computational resources, long time and sophisticated procedures. This often leads to a common situation, when log files are continuously generated and occupy valuable space on storage devices, but nobody uses them and utilizes enclosed information.

1.2 Motivation

There are various applications (known as log file analyzers or log files visualization tools) that can digest a log file of specific vendor or structure and produce easily human readable summary reports. Such tools are undoubtedly useful, but their usage is limited only to log files of certain structure. Although such products have configuration options, they can answer only built-in questions and create built-in reports.

The initial motivation of this work was the lack of Cisco NetFlow [13, 14] analyzer that could be used to monitor and analyze large computer networks like the metropolitan area network of the University of West Bohemia (WEBNET) or the country-wide backbone of the Czech Academic Network (CESNET) using Cisco NetFlow data exports (see 3.1.2). Because the amount of log data (every packet is logged!), evolution of the NetFlow log format in time and wide spectrum

of monitoring goals/questions, it seems that introduction of an new, systematic, efficient and open approach to the log analysis is necessary.

There is also a belief that it is useful to research in the field of log files analysis and to design an open, very flexible modular tool, that would be capable to analyze almost any log file and answer any questions, including very complex ones. Such analyzer should be programmable, extendable, efficient (because of the volume of log files) and easy to use for end users. It should not be limited to analyze just log files of specific structure or type and also the type of question should not be restricted.

1.3 Problem Definition

The overall goal of this research is to invent and design a good model of generic processing of log files. The problem covers areas of formal languages and grammars, finite state machines, lexical and syntax analysis, data-driven programming techniques and data mining/warehousing techniques.

The following list sums up areas involved by this research.

- formal definition of a log file
- formal description of the structure and syntax of a log file (metadata)
- lexical and syntax analysis of log file and metadata information
- formal specification of a programming language for easy and efficient log analysis
- design of internal data types and structures (includes RDBMS)
- design of such programming language
- design of a basic library/API and functions or operators for easy handling of logs within the programming language
- deployment of data mining/warehousing techniques if applicable
- design of an user interface

The expected results are both theoretical both practical. The main theoretical result are (a) design of the framework and (b) the formal language for log analysis description. From practical point of view, the results of the research and design of an analyzer should be verified and evaluated by a prototype implementation of an experimental Cisco NetFlow analyzer.

1.4 Current State of Technology

In the past decades there was surprisingly low attention paid to problem of getting useful information from log files. It seems there are two main streams of research.

The first one concentrates on validating program runs by checking conformity of log files to a state machine. Records in log file are interpreted as transitions of given state machine. If some illegal transitions occur, then there is certainly a problem, either in the software under test or in the state machine specification or in the testing software itself.

The second branch of research is represented by articles that just describe various ways of production statistical output.

The following items summarize current possible usage of log files:

- generic program debugging and profiling
- tests whether program conforms to a given state machine
- various usage statistics, top tens, etc.
- security monitoring

According to available scientific papers it seems that the most evolving and developed area of log file analysis is the WWW industry. Log files of HTTP servers are nowadays used not only for system load statistic but they offer a very valuable and cheap source of feedback. Providers of web content were the first one who lack more detailed and sophisticated reports based on server logs. They require to detect behavioral *patterns*, paths, trends etc. Simple statistical methods do not satisfy these needs so an advanced approach must be used.

According to [12] there are over 30 commercially available applications for web log analysis and many more free available on the Internet. Regardless of their price, they are disliked by their user and considered too slow, inflexible and difficult to maintain.

Some log files, especially small and simple, can be also analyzed using common spreadsheet or database programs. In such case, the logs are imported into a worksheet or database and then analyzed using available functions and tools.

The remaining but probably the most promising way of log file processing represent data driven tools like AWK. In connection with regular expressions are such tools very efficient and flexible. On the other hand, they are too low–level, i.e. their usability is limited to text files, one-way, single-pass operation. For higher–level tasks they lack mainly advanced data structures.

1.5 Current Practice

Prior to a more formal definition, let us simply describe log files and their usage. Typically, log files are used by programs in the following way:

- The log file is an auxiliary output file, distinct from other outputs of the program. Almost all log files are plain text files.
- On startup of the program, the log file is either empty, or contains whatever was left from previous runs of the program.
- During program operation, lines (or groups of lines) are gradually appended to the log file, never deleting or changing any previously stored information.
- Each record (i.e. a line or a group of lines) in a log file is caused by a given event in the program, like user interaction, function call, input or output procedure etc.
- Records in log files are often parametrized, i.e. they show current values of variables, return values of function calls or any other state information
- The information reported in log files is the information that *programmers* consider important or useful for program monitoring and/or locating faults.

Syntax and format of log files can vary a lot. There are very brief log files that contain just sets of numbers and there are log files containing whole essays. Nevertheless, an average log file is a compromise of these two approaches; it contains minimum unnecessary information while it is still easily human-readable. Such files for example contain both variable names both variable values, comprehensive delimiters, smart text formating, hint keywords, comments etc.

Records in log files are often provided by time stamps. Although it is not a strict rule, log files without timestamps are rarely seen, because time-less events are of less valuable information.

2 Theoretical Fundamentals And Origins

This chapter introduces a theory of log files and several methods and problem domains that can be used as bases for further research. As it was mentioned before, there is no rounded-off theory of log files in computer science, so this chapter is more or less a heterogenous mixture of related topics. The source of the provided information are various papers that are referred in respective sections.

2.1 Foundations

This section provides a formal definition of a log file, i.e. it defines more precisely the abstract notation of report trace in contrast to the concrete notation of a log file as it is defined in [1, 2].

Definition Given a set R of *report elements* and a distinguished subset $K \subset R$ of *keywords*, we define a *report* as a finite sequence of report elements beginning with a keyword or with an ordered pair $[timestampelement; keyword]$.

There is an assumption that each report (often also referenced as *record*) starts with a keyword (or by a keyword preceded by a timestamp) because this is a common and reasonable pattern in log files. We write $\mathcal{R}_R$ for the set of reports arising from R.

Definition We define a *report trace* as a finite or infinite sequence of reports. For a report trace ρ of finite length, we define $|\rho|$ as the length of the trace.

It is necessary to consider also infinite report traces because of the possibility of continuously running processes like operating systems. Report traces are the mathematical notion, log files are real-word objects. It is useful to have at least simple definition of a log file corresponding to a finite report trace:

Definition A *portrayal function* is an injective function a from report elements to sequences of non-blank, printable ASCII characters, such that for a keyword k, $a(k)$ is a sequence of alphanumeric characters, numbers and underscores beginning with a letter.

We can extend a portrayal function to a function from reports to printable ASCII strings

Definition Function $a(e_1, e_2, ...e_n)$ is equal to $a(e_1) \oplus b \oplus a(e_2) \oplus b \oplus ... \oplus a(e_n)$, where $\oplus$ is a string concatenation operator and b is ASCII blank.

Definition For a report trace t, we call $a(t)$ the *log file corresponding to* t

The formal definitions outlined here are further used in [1, 2, 3, 4, 5] in the theory of software validation using finite state–machines.

2.2 Universal Logger Messages

The standardization efforts concerning log files resulted in (today expired) IETF draft [18] that proposes the *Universal Format for Logger Messages, ULM*.

The presented ULM format is a set of guidelines to improve semantic of log messages without exact formalization. It can be considered to be more a programming technique than a formal, theoretical description. On the other hand, the idea is valuable and can be utilized in further research.

In a ULM, each piece of data is marked with a tag to specify its meaning. For example in the message

```
gandalf is unavailable
```

even a human administrator could difficultly determine whether `gandalf` is a host, a user nickname, a network, or an application. But the

```
file_name=gandalf status=unavailable
```

notation disambiguates the meaning of each piece of data, and allows an automatic processing of the message.

ULM defines syntax of log files and some mandatory and optional fields. These fields, called *tags*, are linked to every log file record and they should eliminate any ambiguity.. Mandatory tags are listed in the following table:

tag	**meaning**	**value(s)**
LVL	importance	emergency, alert, error, warning, auth, security usage, system, important, debug
HOST	name	computer that issues the message
PROG	name	program that generated the log event
DATE	time stamp	YYYYMMDDhhmmss[.ms] [(+/-) hhmm]

Unlike mandatory tags those must be used, the optional tags merely supply additional useful information and should be used where applicable. The following table shows "basic" optional tags, if fact you may create your own tags if it is really necessary:

tag	meaning	value(s)
LANG	language	language used in textfields
DUR	duration	duration of the event in seconds
PS	process	process ID in multi-tasking environment
ID	system	any system ID/reference
SRC.IP	source	IP address in dotted decimals
SRC.FQDN	source	fully qualified domain name
SRC.NAME	source	generic name (host/user name)
SRC.PORT	source	TCP/UDP port number
SRC.USR	source	user name/ID
SRC.MAIL	source	related e-mail address
DST.*	destination	like SRC
REL.*	relay	proxy identification, like SRC
VOL.SENT	volume	bytes sent
VOL.RCVD	volume	bytes received
CNT.SENT	count	objects sent
CNT.RCVD	count	objects received
PROG.FILE	filename	program source file that issues the message
PROG.LINE	line	source line number
STAT	status	failure, success, start, end, sent etc.
DOC	name	identification of accessed file/document
PROT	protocol	used network protocol
CMD	command	issued command
MSG	text	arbitrary data that do not fit any category

The ULM proposal is one solution to the problem of chaotic or "encrypted" log files. It is a good solution for certain types of log files like operation system reports and similar. On the other hand, the ULM overhead is too big for applications like HTTP server or NetFlow and similar.

The main advantage of ULM is easy application in very heterogenous log files with many types of records because of the semantic information that is explicitly connected to every record. On the other hand, in homogeneous log files, the explicit semantic description stands just for useless overhead.

The ULM draft has expired without any successor. Some suggestions or even notices can be found on the Internet promising that ULM should be replaced by a system based on XML but no draft is available yet.

2.3 What Can We Get from Log Files

This paragraph summarizes application of log files in software development, testing and monitoring. The desired useful information that resides in log files can

be divided into several classes:

- generic statistics (peak and average values, median, modus, deviations...)
  Goal: finding and processing of set of report elements X, $X \subset R$, in report trace ρ
  Useful for: setting hardware requirements, accounting

- program/system warnings (power failure, low memory)
  Goal: finding all occurrences of reports $X \subset \mathcal{R}_K$ in report trace ρ, where K is a given set of keywords and $\forall x_i \in X \; \exists r_j \in x_i : r_j$ fulfills a given condition
  Useful for: system maintenance

- security related warnings
  Goal: finding all occurrences of of reports $x \in \mathcal{R}_K$ in report trace ρ, where K is a given set of keywords

- validation of program runs
  Goal: construction and examinqtion of state-machine $\mathcal{A} < Q, \Sigma, \delta, \rho_0, F >$; Q-set of states, $\Sigma \approx \rho$, $\delta : Q \times \Sigma$, $\rho_0 \in Q$, $F \subset Q$
  Useful for: software testing

- time related characteristics
  Goal: computing or guessing time periods between multiple occurences of report $x \in R$ or between corresponding occurences of reports x and y, $x, y \in R$ in report trace ρ
  Useful for: software profiling & benchmarking

- causality and trends
  Goal: finding a set of reports $X, X \subset \rho$ that often (or every time) preceed (or follow) within a given time period (or withing a given count of reports) a given report $y \in \rho$
  Useful for: data mining

- behavioral patterns
  Goal: using a set of reports $X \subseteq \Sigma \subset \rho$ as the set of transitions of a given Markov process Useful for: determinig performance and reliability

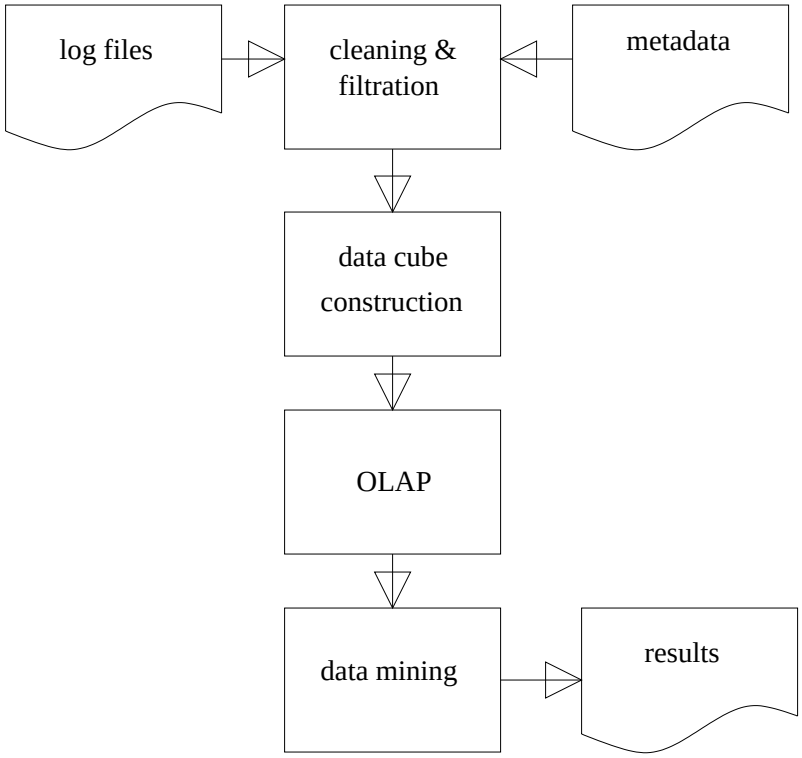
- ...
  Goal: ???
  Useful for: ???

The last item indicates that users may wish to get any other inforation for their specific reasons. Such wishes are expected to gradually emerge in the future; therefore any analytical tool should definitelly accept user-specified assignments.

2.4 Generic Log File Processing Using OLAP

It is obvious that current approach "different application — different log file format — different approach" is neither effective nor simple reusable. In addition, all log file analyzers must perform very similar tasks. This section presents one approach to generic log file analysis as it is described in [12].

The proposed generic log file analysis process consists of four steps that are ilustrated by the following figure that contains an outline of an engine of analysis:



1. **Data cleaning and filtration.** In the first stage, data are filtered to remove all irrelevant information. The remaining meaningful data are transformed into a relational database. The database is used as an effective repository for information extraction, for simple aggregation and data summarization based on simple attributes.

   This step requires the following programming techniques:

   - filtration (preferably based on regular expressions)
   - lexical analysis
   - syntax analysis/validation

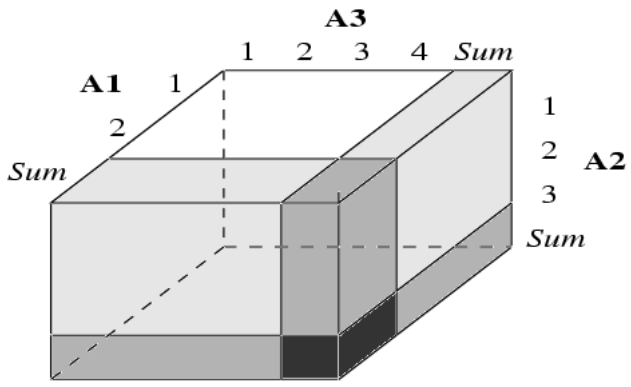
   The format and structure of the log file must be known to the analyzer. Let us call this information *metadata*. The metadata should not be definitely

hard-wired into the analyzer (what is unfortunately a common practice) but should be preferably explicitly defined in a separate data file. The meta-data information can also contain log-specific information used for filtration and transformation like mapping tables, translation dictionaries, synonym dictionaries etc.

2. **Data cube construction.** In the second stage, a data cube is created using all available dimensions.

   Definition An *n-dimensional data-cube* $C[A_1, ..., A_n]$ is a n-dimensional database where $A_1, ..A_n$ are n dimensions. Each dimension of the cube, A_i, represents an attribute and contains $|A_i| + 1$ rows where $|A_i|$ is the number of distinct values in the dimension A_i. The first $|A_i|$ rows are *data rows*, the last remaining row, the *sum row*, is used to store the summarization of corresponding columns in the above data rows. A data cell in the cube, $C[a_1, ..., a_n]$, stores the count of the corresponding tuple in the original relation. A sum cell, such as $C[sum, a_2, ..., a_n]$, (where sum is often represented by keyword "all" or "*") stores the sum of the counts of the tuples that share the same values except the first column.

   The following figure illustrates a 3D data cube with highlighted sum rows:



   The multi-dimensional structure of the data cube allows significant flexibility to manipulate the data and view them from different perspectives. The sum row allows quick summarization at different levels of hierarchies defined on the dimension attributes.

   A large number of data cell can be empty but there exist sparse matrix techniques to handle sparse cubes efficiently.

3. **On-line analytical processing (OLAP).** Building a data cube allows the application of on-line analytical processing (OLAP) techniques in this stage. The drill-down, roll-up, slice and dice methods are used to view and analyse the data from different angles and across many dimensions.

4. **Data mining.** In the final stage, data mining techniques are put to use with the data cube to dig out the desired interested information. The common data mining goals are the following: data characterization, class comparison, association, prediction, classification and time series analysis.

The above described four-step model of log file processing also implies a possible structure of an analytical engine. This model describes only processing of log files (i.e. the steps and their order) so it must be suplemented with a description of goals of the analysis. The model has also some limitations and thus it is not suitable for all analytical task and for all languages of analysis.

On the other hand, we can generalize this model and make it compatible with other language and system of analysis.

2.5 Text Processing Languages

There are several languages designed for easy text processing. Their opeation is based on regular expressions that makes their usage suprisingly efficient. The well-known representatives are AWK and Perl.

Log analysis (or at least simple log analysis) is in fact a text processing task, so the need to examine such languages is obvious. We will concentrate on AWK, because its application is only in text processing (unlike Perl). The rest of this paragraph is taken from AWK manual[25].

The basic function of awk is to search files for lines (or other units of text) that contain certain patterns. When a line matches one of the patterns, awk performs specified actions on that line. awk keeps processing input lines in this way until the end of the input files are reached.

Programs in awk are different from programs in most other languages, because awk programs are data-driven; that is, you describe the data you wish to work with, and then what to do when you find it. Most other languages are procedural; you have to describe, in great detail, every step the program is to take. When working with procedural languages, it is usually much harder to clearly describe the data your program will process. For this reason, awk programs are often refreshingly easy to both write and read.

When you run awk, you specify an awk program that tells awk what to do. The program consists of a series of rules. (It may also contain function definitions, an advanced feature which we will ignore for now. See section User-defined Functions.) Each rule specifies one pattern to search for, and one action to perform when that pattern is found.

Syntactically, a rule consists of a pattern followed by an action. The action

is enclosed in curly braces to separate it from the pattern. Rules are usually separated by newlines. Therefore, an awk program looks like this:

```
pattern { action }
pattern { action }
...
```

2.6 Data Mining and Warehousing Techniques

Some procudures during log file analysis can be a direct application of data warehousing and data mining techniques. For very large log files (hundreds of megabytes) text files become inefficient; the log information should be rather stored in a binary form in a database and manipulated efficiently using some DBMS. The usage of common database techniques is then straightforward and therefore also data data mining and warehousing techniques should be studied and evaluated in connection with log file analysis.

2.7 Event Ordering Problem

Many distributed systems suffer from so called event ordering problem that is caused by usage of multiple physical clocks. Log file analysis is no exception to the rule:

When merging log files from different nodes, the event have timestamps made by different clocks. It means that the events within each node are ordered (by timestamps of the same clock) but events originating at different places are not ordered because there is no chance how to identify which event has happened earlier.

The only solution is to use logical clocks but it is not often possible. Common workaround is a precise physical clock synchronization provided by advanced time protocols like NTP that can guarantee clock precision within a few milliseconds. Unfortunatelly, this is often not enough for very fast network or software engineering applications (for example gigabit switching, remote procedure calls, component-based middleware) when events can occur "almost" simultaneously. In addition, we can not verify (mainly in case of log analysis) whether the clocks are (or were) synchronized and how precisely.

The problem of clock synchronization and ordering of events is well explained in [6, 7] and [8].

2.8 Markov Chain Analysis

In some cases, depending on log records semantic and the corresponding analytical tasks, we can use the Markov model for analytical processing.

In more detail, we can threat various log entries as a transitions in a Markov chain. Corresponding Markov states can be specified for example using respective metadata information.

If there are no absorbtion states, then we can compute steady state probabilities, transition matrix and diagrams etc. and use them in further analysis (unless they are final results).

2.9 Logging Policies and Strategies

Logging policies and strategies are sets of precise rules that define what is written to a log file, under what conditions and in what explicit format. Logging strategies concern terms like the level of detail and abstraction etc.

Logging policies can be enforced by automatic instruments or simply by code review and inspection procedures.

The main point here is that log policies can under some conditions significantly affect results of analysis; therefore at least a rough knowledge of corresponding log policies and strategies is expected when writing a log analysis program for a given software.

Although logging policies play a fundamental role in the logging subsystem, they are slightly off-topic to this report and will not be more mentioned.

2.10 Conclusion

The brief theoretical inspection provided in this chapter revealed several fundamental ideas that are essential for further work. The most important conclusions are mentioned in the following list:

- some formal description of log files
- ULM as a way how to add semantic
- finite state machines can be used for program validation
- the set of expected user analytical assignments is infinite
- the idea of metadata

- the idea of four–step generic model
- the idea of data cleaning and filtration
- the idea of data cube
- an analytical engine can use OLAP
- data mining and warehousing techniques can be used
- there is a event ordering problem if merging distributed logs
- under some circumstances the Markov model can be used
- knowledge of log policies and strategies is recomended

3 Case Studies

This section concerns several problem domains where extensive usage and analysis of log files is critical or at least of a great benefit. There are presented examples of log files, mainly in the field of distributed systems, those can not be fully utilized using available analyzers. The purpose of this section is also to reveal the variety of log files and related application.

3.1 Cisco NetFlow

Cisco NetFlow [13, 14] is an advanced technology of fast IP switching embedded in Cisco boxes. Opposite to classic routing when next hop address is calculated separately for every packet, NetFlow enabled routers examine membership of packets to a *flow*. This is faster than examination of whole routing table and thus makes routing faster. As a side effect, routers can periodically export information about each flow.

3.1.1 Concepts

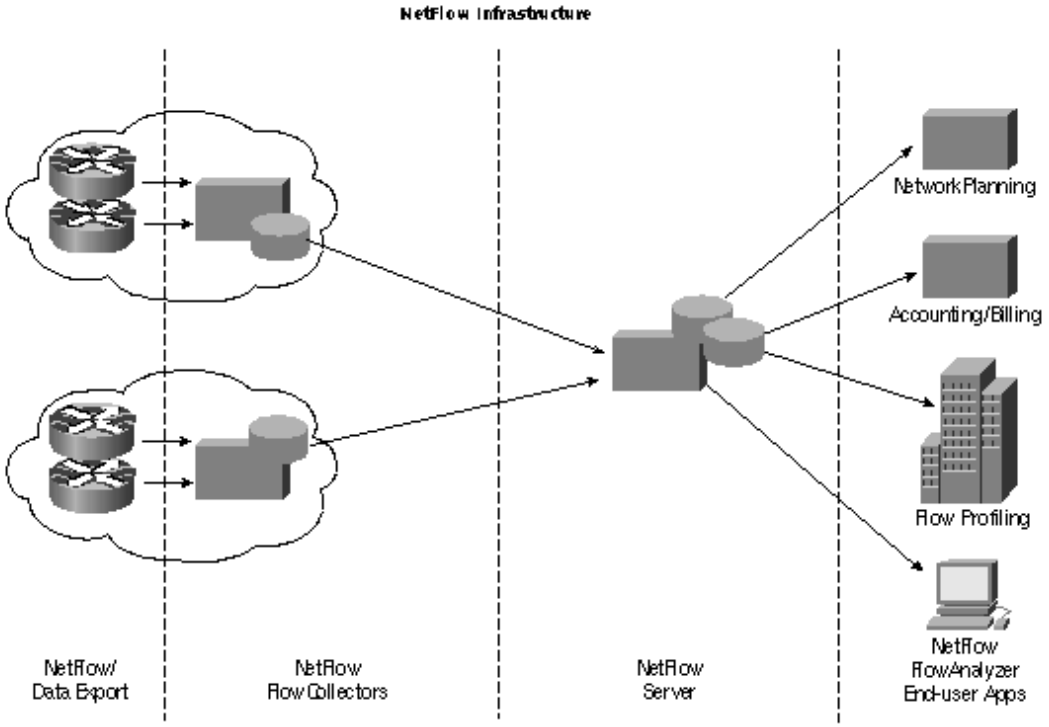
NetFlow services consist of high-performance IP switching features that capture a rich set of traffic statistics exported from routers and switches while they perform their switching functions. The exported NetFlow data consists of traffic *flows*, which are unidirectional sequences of packets between a particular source device and destination device that share the same protocol and transport layer information.

Routers and switches identify flows by looking for the following fields within IP packets:

- Source IP address
- Destination IP address
- Source port number
- Destination port number
- Protocol type
- Type of service (ToS)
- Input interface

NetFlow enables several key customer applications:

- Accounting/Billing
- Network Planning and Analysis
- Network Monitoring with near real-time network monitoring capabilities. Flow-based analytical techniques may be utilized to visualize traffic patterns associated to individual devices as well as on network-wide basis.
- Application Monitoring and Profiling
- User Monitoring and Profiling
- NetFlow Data Warehousing and Mining. This is especially useful for Internet Service Providers as NetFlow data give answers to the "who", "what", "when" and similar questions.



3.1.2 NetFlow Data Export

NetFlow data export makes NetFlow data statistics available for purposes of network planning, billing and so on. A NetFlow enabled device maintains a flow cache used to capture flow-based traffic statistics. Traffic statistics for each active

flow are maintained in the cache and are incremented when packets within the flow are switched. Periodically, summary traffic information for all expired flows are exported from the device via UDP datagrams, which are received and further processed by appropriate software. Flow cache entries expire and are flushed from the cache when one of the following conditions occurs:

- the transport protocol indicates that the connection is complete (TCP FIN) or reset
- traffic inactivity exceeds 15 seconds

For flows, that remain continuously active, flow cache entries expire every 30 minutes to ensure periodic reporting of active flows.

3.1.3 Data Formats

NetFlow routers export flow information in UDP datagrams in one of four formats: Version 1, Version 5, Version 7 and Version 8. V1 is the original version, V5 adds BGP autonomous system identification and flow sequence numbers, V7 is specific to Cisco Catalyst 5000 with NFFC and V8 adds router-based aggregation schemes. Omitting a discussion here, this work concerns only NetFlow version 5.

Each PDU (protocol data unit) encapsulated in a UDP datagram consist of a header and a variable number of flow reports. The header is of the following format for the version 5:

```
ushort version;      /* Current version=5 */
ushort count;        /* The number of records in PDU */
ulong SysUptime;     /* Current time in ms since router booted */
ulong unix_sec;      /* Current seconds since 000 UTC 1970 */
ulong unix_nsec;     /* Residual nanosecs since 0000 UTC 1970 */
ulong flow_sequence; /* Sequence number of total flows seen */
uchar engine_type;   /* Type of flow switching engine (RP, VIP) */
uchar engine_id;     /* Slot number of the flow switch */
```

Each flow report within a version 5 NetFlow export datagram has the following format:

```
ipaddrtype srcaddr; /* Source IP Address */
ipaddrtype dstaddr; /* Destination IP Address */
ipaddrtype nexthop; /* Next hop router's IP Address */
ushort input;       /* Input interface index */
```

```
ushort output;      /* Output interface index*/
ulong dPkts;        /* Packets sent in Duration (ms between */
                    /* first and last packet in this flow)*/
ulong dOctets;      /* Octets sent in Duration (ms between */
                    /* first and last packet in this flow)*/
ulong First;        /* SysUptime at start of the flow ... */
ulong Last;         /* ...and of last packet of the flow */
ushort srcport;     /* TCP/UDP source port number */
                    /* (e.g. FTP. Telnet, HTTP, etc.) */
ushort dstport;     /* TCP/UDP destination port number */
                    /* (e.g. FTP. Telnet, HTTP, etc.)*/
uchar pad;          /* pad to word boundary */
uchar tcp_flags;    /* Cumulative OR of TCP flags */
uchar prot;         /* IP protocol, e.g. 6=TCP, 17=UDP etc. */
uchar tos;          /* IP type of service */
ushort dst_as;      /* destination peer  Autonomous system */
ushort src_as;      /* source peer  Autonomous system */
uchar dst_mask;     /* destination route's mask bits */
uchar src_mask;     /* source route's mask bits*/
uchar pad;          /* pad to word boundary */
```

3.1.4 Event Ordering Problem

NetFlow is unfortunately also a good example of the events ordering problem explained in 2.7. The Cisco boxes use for timestamps their own hardware clocks that can be optionally synchronized using the NTP protocol. This is enough for various statistical analysis but not enough for tracing down individual packets (or small, repeated pieces of information aggregated in separate flows) as they travel through the network. For simple statistical analysis this is not fatal unless the clock are significantly apart.

3.1.5 Conclusion

In completely distributed systems like computer networks, knowledge may be distributed (or shared) between a group of nodes. For example, from the point of view of individual routers everything seems fine, but the composite information (over border routers) shows that some hosts suffer a DDOS attack (distributed denial of service) or that there are significant load disbalance.

Although NetFlow Data Export uses a binary format, it is still a log file. When taking into account also the volume of logged data (i.e. hundreds of megabytes daily) then plain text files seem not to be a good choice anyway.

Network protocols constantly evolve; a network analysis tool must keep up with the changes. Network administrators may also ask questions or examine phenomena that are not known at design time. A good analyzer must face all these challenges. The questions are numerous but not limited. This implies a modular (plug-in based) design of any analytical tool—one plug-in module for one question/problem.

There is also the problem of event ordering. Cisco routers can use NTP protocol for clock synchronization but it is not precise enough and thus we can encounter problems when examining causality.

3.2 Distributed Software Components

This section discuses challenges of log file analysis in the field of software components. The problem is explained on a new component architecture called *SOFA* (that means Software Appliances) that is developed at the Charles University, Prague [20] and at the University of West Bohemia in Pilsen.[21]

3.2.1 SOFA Concepts

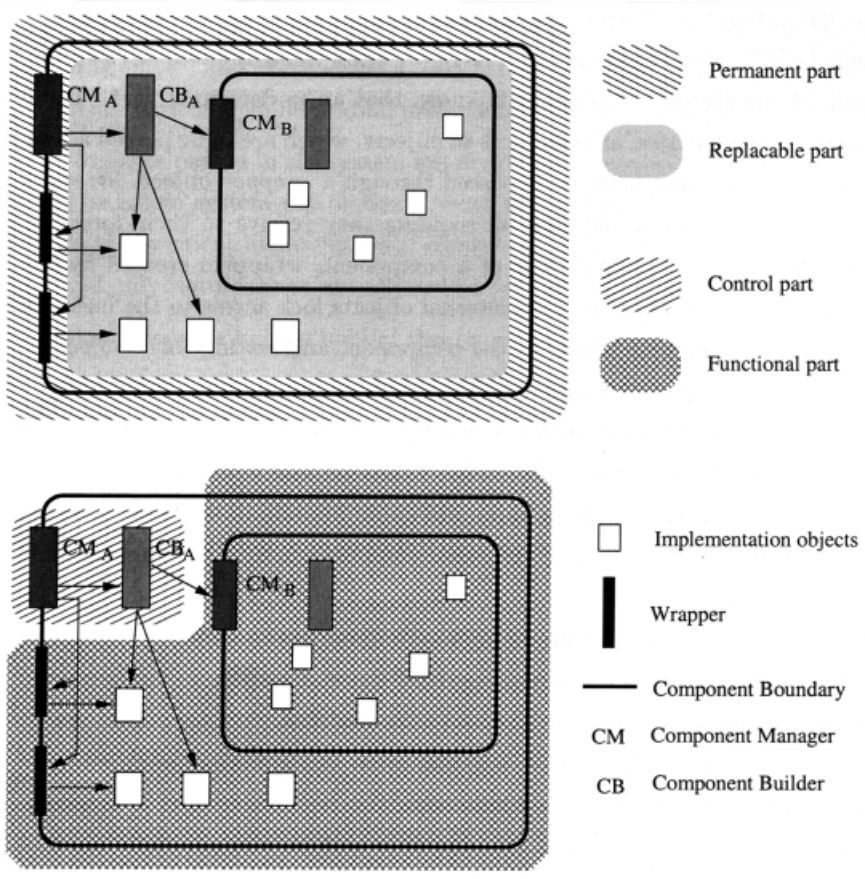
A SOFA application is a hierarchy of mutually interconnected software components. An application can be created just by a composition of bought components perhaps with great reuse of already owned components. The *DCUP* (Dynamic Updating) feature allows to upgrade or exchange components at application runtime.

SOFA components are described in *Component Description Language*. CDL is a high level structured language capable to describe interfaces, frames, architecture, bindings and protocols. CDL is used both at compile and run time.

A SOFA component is a black box of specified type with precisely defined *interfaces*. Each component is an instance of some component type. At runtime a component consists of a permanent and a replaceable part. The permanent part creates a "border" of the component. It contains the *Component Manager(CM)* and wrapper objects that intercept calls on component's interfaces. The replaceable part is formed by the *Component Builder (CB)*, internal implementation objects and subcomponents. In other words, CM and CB form the control part of a component while the rest is a functional part.

An important feature of SOFA architecture is support for electronic market of components. That is why SOFA introduces *SOFAnet* and *SOFAnode* concepts that reflect various roles that companies play on a market—they are producers, retailers, end-users or a combination of these. SOFAnet is a homogeneous network of SOFA nodes. In contrast to other component architectures, SOFA covers not

only the computer oriented aspects (objects, call schemes, operating systems) but it also faces real-word problems like manufacturing, trading, distribution and upgrading of components. From this point of view SOFA is a complex system and not only a component-based middleware.



3.2.2 Debugging SOFA Applications

Debugging distributed application is in general a difficult task. SOFA concepts make this task a bit easier because of the following reasons:

- SOFA utilizes an idea of interface wrappers. Every incoming or outgoing function/method call is processed by the wrapper. Interface wrappers can perform various value-added useful operations but they are also a nice place where to create run trace information.

  Programmers can easily use interface wrappers to put down component calls just by adding few lines of source code. The logging function of inter-

face wrappers is built-in in SOFA and can be simply turned on and off by SOFAnode administrator on a per-node or per-application basis.

- SOFA also introduces the idea of *behavioral protocols*. [22] Protocols are a part of CDL that can be used for validation of method call semantics. For example, a database must be first open, then (repeatedly) used and finally closed. Any different call order is wrong and implies a bug in the application.

  The validation can be performed either during application run-time or ex-post using log files.

  Let us look at a simple protocol that defines behavior of a sample transaction interface. Semicolon means sequence operation, plus sign means exclusive OR and an asterisk means iteration. The example — a real CDL fragment— declares interface `ITransaction` with methods `Begin`,`Commit`,`Abort` and specifies how to use them:

```
interface ITransaction{
    void Begin();
    void Commit();
    void Abort();
 protocol:
    (Begin; (Commit + Abort))*
}
```

  Protocols can span single interface or they can also span over several components. The following example shows a simple database component `Database`, that offers three methods `Insert`, `Delete`, `Query` via interface `dbSrv` and translates them into an iteration of calls over interfaces `dbAcc` and `dbLog`. The protocol distinguishes incoming (?) and outgoing (!) method calls.

```
frame Database{
  provides:
    IDBServer dbSrv;
  requires:
    IDatabaseAccess dbAcc;
    ILogging dbLog;
  protocol:
    !dbAcc.open;
    (  ?dbSrv.Insert{(!dbAcc.Insert; !dblog.LogEvent)*}+
       ?dbSrv.Delete{(!dbAcc.Delete; !dblog.LogEvent)*}+
       ?dbSrv.Query{!dbAcc.Query*}+
```

```
    )*;
    !dbAcc.Close;
}
```

Behavioral protocols are one way of formal description of program behavior. State machines are more straightforward and easier for real validation of program runs.

3.2.3 Conclusion

It is obvious that SOFA (and of course other component architectures) allow easy tracking of method calls and thus programmers would utilize a flexible tool for log analysis. The log information does not rise at a single central point, therefore the proper merge of multiple run-traces is important. That means either a precise clock synchronization (see 2.7) and precise timestamps in log files, or the analyzer must be able to work with partly ordered sets (posets) of run traces.

Again, the important information can be distributed (or shared) "in the system" while it is not obvious in any single component.

3.3 HTTP Server Logs

HTTP server and cache/proxy logs are the area of the most frequent and intensive analytical efforts. These logs contain records about all HTTP requests (called hits) made by web browsers. These log files can be source of valuable (and sensitive) information such as system load, megabytes transmitted, number of visitors etc. Besides common system statistics that are interesting for server operators, the content providers are interested mainly in *behavior* of the visitors, *usage trends* and causality issues. Many of such questions can not be answered by simple statistic tools and require more sophisticated techniques. In addition, popular web sites can see their log files growing by hundreds of megabytes every day that makes the analysis time-consuming and thus puts also strong performance requirements on the analyzer.

3.3.1 Adaptive Sites

The vision of today's web technology are so called *adaptive sites* — sites that improve themselves by learning from user access patterns. For instance, a WWW server can offer the most sought-after items to the beginning of lists, remove unattractive links etc. It is straightforward that such sites can rely on a powerful log analyzer. (Today, it is hard to imagine that HTTP servers could perform such calculations on-line.)

3.3.2 Sample Questions

Although web server logs are not the primary objective of this thesis, this section provides a brief overview of typical questions concerning web usage.

- *Who is visiting your site.* The fundamental point is who are the readers of your server, who are they like, from what countries, institutions etc. Many servers identify returning visitors and offers personal or personalized pages for frequent visitors.
- *The path visitors take through your pages.* Where do visitors start reading your server, where they come from, how easily they navigate, do they often fail to find the right hyperlink and thus often return... In brief, the objectives are *trends, paths,* and *patterns*.
- *How much time visitors spend on each page.* From the answer you can deduce what pages are interesting and what are very confusing or dull.
- *Where visitors leave your site.* The last page a visitor viewed can be a logical place where end the visit or it can be a place where visitor bailed out.
- *The success of users experiences at your site.* Purchases transacted, downloads completed, information viewed, tasks accomplished. Such indicators can show whether the site it well made, organized and maintained, whether the offered products are well presented.

3.3.3 Conclusion

HTTP log analysis and visualization is an intensively studied field. The point of the biggest importance here is to avoid all possible mistakes reported in available papers. The most frequent complaints are:

- analyzers are not enough flexible and therefore their usability is limited
- analyzers are too slow
- analyzers are difficult to use

In addition, HTTP log analyzers often offer some value-added functions like DNS translation. This idea can be generalized - an analyzer should be distributed along with a library (or a set of plug-ins or an API) of handy functions for such purposes.

4 Analysis of Requirements

This section present a summary of requirements that are put on an analytical tool for log file analysis. Some details were also discussed in the previous section, so the paragraphs bellow concern more general topics like usability, performance and user wishes. The term *tool* is used within this chapter when referencing a general log analysis system according the subsequent definition.

Definition A log analysis *tool* is a piece of software that allows analytical processing of log files according to a user-specified instructions. Further, the following prerequisites are expected:

1. there exist one or more log files in ULM-like format or in format specified in metadata
2. there exist metadata information if necessary
3. there exist one or more assignments (analytical tasks or goals) specified using in the given language of analysis
4. there exist an analyzer (either on-line or off-line) that can execute the assignments and report obtained results

4.1 Flexibility

The tool must be usable for various analytical tasks on any log file. Flexibility concerns log file format, syntax, semantics and types of queries (questions, statistics and other analytical tasks) that can be performed (see page 10).

The main point here is that exact types of queries are not known during the design time of a tool, so the tool must accept new queries at run time. Therefore the tool should offer a way (a programmable interface) to allow users to specify their own assessments of analysis and write corresponding programs or scripts.

The tool must cope well with either small or huge log files, either simple or very complex log files, either statistical or analytical queries.

4.2 Open Design, Extensibility

The tool must be open for later third-parties extensions. There must exist a way (may be an API) how to access internal data structures. This would probably results in a multi–layer design of the tool, with a core layer and many plug-in modules in higher layers.

Open design also means proper selection of involved technologies, i.e. open, standardized formats like XML, ULM, HTML etc. are preferred instead of any proprietary or ad-hoc solutions.

4.3 Modularity

The tool should be modular. A modular design allows easy extension of tool capabilities via additional packages for specific tasks. For example a package for log analysis of concrete piece of software can contain dozens of modules for various log cleaning and filtration, log transformation, queries processing and visualization. The user shall activate or deactivate appropriate modules in the analysis process according his or her wishes.

The plug-in modules can be divided into disjunct classes according their function:

- **Input filters** that perform data cleaning, filtering and transformation into internal data structures (or DBMS)
- **Data analyzers** that process queries, one type/kind of query per one module
- **Reporters** that present results of log file analysis in a form of text reports, lists, tables or in some visual way

4.4 Easy Application

The tool must be easy to use. It means a well-designed user interface and a good language for data and query description. For example, spreadsheets or AWK[24] are powerful enough even for non-trivial analytical tasks but they are not easy to use mainly because of lack of structured data types, abstraction, variable handling and overall approach.

The tool must work at high level of abstraction (to be universal) but on the other hand, users must be able to express their ideas in a simple and short way. For instance, the tool can utilize regular expressions for easy log cleaning, filtration and transformation.

The user-written modules should access log data via a structured way. An object model (like in dynamic HTML) seems to satisfy well this requirement. The object model allows both native and synthetic (computed) attributes and easy data manipulation.

The tool should support automatic recognition of data types (number, string, date, time, IP address) and allow multiple formats where applicable (i.e. the

time). The tool should also offer handy operators for DNS translation, time calculation, statistical tasks, precedence, causality and similar.

4.5 Speed

The speed issue is very important in case of working with large log files. The duration of log analysis depends on log size, CPU speed, free memory, number and complexity of queries and possibly on network/hard drive speed.

Some log files can grow at amazing speed. If the tool must store hundreds of new log entries per minute, the consumption of system resources at "idle" must be also considered.

If the tool should be used for continuous, near real-time monitoring/analysis, the speed becomes the most critical criterion.

4.6 Robustness

The tool is expected to work with log files of hundreds of megabytes. Some log entries can be damaged of missing due to errors in software, hardware errors, network down time and so on. Metadata information may not match log file and there can be also errors in user-written modules.

All these aspects can cause fatal miss-function or crash of the tool. Therefore the following issues should be incorporated into the design.

- all log entries that do not conform supplied metadata information must be filtered out
- the metadata information must be validated for correct syntax
- the user-written programs must be "executed" in a safe environment. The tool must handle or monitor allocation of memory, I/O operations and perhaps somehow limit the time of module execution (in case of infinite loops etc.)
- all user-supplied information must be checked for correctness
- tool design should avoid any inefficiency and unnecessary complexity

These are only the basic ideas; robustness is the art of proper software design and careful implementation.

4.7 Language of Analysis

A language for description of the analytical task (i.e. the language of user-written modules) is a subject of this research. It should be definitely a programming language to be enough powerful and to allow all necessary flexibility.

The language should work at high level of abstraction; it must certainly offer some descriptive and declarative features. Concerning the programming style, a data–driven or imperative approach seems to best choices. If possible, the language should allow usage in connection with both compilers and interpreters.

Further, the language should be simple (to allow easy implementation) but enough flexible and powerful, probably with object oriented data model with weak type checking. Easy mapping (translation) into lower–level commands of the corresponding engine (for example OLAP techniques or SQL commands) is also an important feature.

4.8 The Analytical Engine

This paragraph concerns the key operation of the tool, but because the engine is more or less hidden from the user, there are no special requirements or user expectations. The only user-visible part of the analytical engine is the task specification, i.e. the language of analysis.

There are three possible operational modes of an engine:

- data–driven, single–pass approach (like AWK[24]); only a portion (possibly one record) of the analyzed log must be in the memory at the same time
- multiple–pass (or single–pass with partial rewinds allowed); a portion of the analyzed log must be in the memory at the same time but not the whole log
- database–oriented approach, whole log is stored in a database; memory usage depends on the DBMS used

From a different point of view, the tool can process batch–jobs or perform continuous operation. In the second case, the tool accepts a stream of incoming log entries and updates the database. The tool can also regularly expire oldest log entries; this way of operation is suitable for system monitoring, when only recent data (one day, week, month) are relevant for analysis.

4.9 Visualization and Data Exports

Presentation of results of analysis in the form of tables, histograms, charts or other easily comprehensive way is the goal of all effort. On the other way, the requirements put on visualization can considerably vary with different analytical task. In addition, including visualization engine directly into the engine would make it too complicated.

Thus the suggestion here is to separate visualization and results reports into separate, user-written plug-in modules and provide necessary API. It could contain primitive functions to print desired data in a list, a table, a histogram, a pie chart etc. that would be processed by some generic visualization module.

4.10 User Interface

User interface should be possibly platform independent and should allow the following tasks:

1. selection of log source (local files or network socket)
2. selection of metadata information if applicable (from a text file)
3. selection/activation of all modules that should participate on the analysis
4. display results of analysis or handle output files

Opposite to the analytical engine, the user interface requires only few system resources. A Java application or web–based interface seems to be a good choice.

4.11 Hardware Requirements

Overall system performance should correspond to the size of log files and complexity of analytical tasks.

The minimum requirements for small logs and simple analysis should be easily fulfilled by present desktop office systems, i.e. the computer must be able to run simultaneously the analytical engine, the user interface, possibly a DBMS and allocate enough memory for log data.

For very large (hundreds of megabytes or gigabytes) log files and complex tasks (or many simple tasks in parallel) the minimum hardware requirements must change respectively: a dedicated multi-CPU machine with 1GB of operation memory and fast hard discs may be necessary.

5 Design and Implementation

This section describes in more detail some design issues and explains decisions that have been made. Although this paper is written during early stages of design and implementation but it is believed to reflect the key design ideas.

5.1 Language of Metadata

The first stage of log file analysis is the process of understanding log content. This is done by lexical scanning of log entries and parsing them into internal data structures according to a given syntax and semantics. The semantics can be either explicitly specified in the log file or it must be provided by a human (preferably by the designer of the involved software product) in another way, i.e. by a description in corresponding metadata file.

5.1.1 Universal Logger Messages and ODBC Logging

First, let us assume that the log is in ULM format (or in ULM modification or in Extended Markup Language, XML), i.e. the syntax is clear and semantic information in included within each record. In such case, the provided information is sufficient and the need of metadata is reduced to description of automatic recognition of used log format.

Similarly, if the log data are already stored in database (several products can store log via ODBC interface instead text files), then the log entries are already parsed and can be referenced using column titles (i.e. attribute names) of the corresponding table. The semantics is then also related to columns.

5.1.2 Log Description

Unfortunately, the common practice is that log files do not contain any explicit semantic information. In such case the user has to supply missing information, i.e. to create the metadata in a separate file that contains log file description.

The format of metadata information—the language of metadata—is hard-wired into the analyzer. The exact notation is not determined yet, but there is a suggestion in the next paragraph.

Generally the format of a log file is given and we can not change internal file structure directly by adding other information. Instead, we must provide some external information. For example we can simply list all possible records in the log file and explain their meaning or write a grammar that would correspond to a given log format.

A complete list of all possibilities or a list of all grammar rules (transcriptions) is exhausting, therefore we should seek a more efficient notation. One possible solution to this problem is based on matching log records against a set of patterns and searching for the best fit. This idea is illustrated by an example in the next paragraph.

5.1.3 Example

For illustration, let us assume a fake log file that contains a mixture of several types of events with different parameters and different delimiters like this:

- structured reports from various programs like
  ```
  12:52:11 alpha kernel:  Loaded 113 symbols in 8 modules,
  12:52:11 alpha inet:  inetd startup succeeded.
  12:52:11 alpha sshd:  Connection from 192.168.1.10 port 64187
  12:52:11 beta login:  root login on tty1
  ```
- periodic reports of a temperature
  ```
  16/05/01 temp=21
  17/05/01 temp=23
  18/05/01 temp=22
  ```
- reports about switching on and off a heater
  ```
  heater on
  heater off
  ```

Then we can construct a description of the log structure somehow in the following way: There are four rules that identify parts of data separated by delimiters that are listed in parenthesis. Each description of a data element consist of a name (or an asterisk if the name is not further used) and of an optional type description. There are macros that recognize common built-in types of information like time, date, IP address etc. in several notations.

```
time:$TIME( )host( )program(:  )message
time:$TIME( )*( sshd:  )*( )address:$IP( port )port
:$DATE( temp=)temperature
(heater )status
```

The first line fits for any record that begins by a timestamp (macro $TIME should fit for various notations of time) followed by a space, a host name, a program name followed by a colon, one more space and a message. The second line fits for record of sshd program from any host; this rule distinguishes also corresponding

IP address and port number. The third line fits for dated temperature records and finally the forth fits for heater status.

Users would later use the names defined herein (i.e. “host”, “program”, “address” etc.) directly in their programs as identifiers of variables.

Note This is just an example how the metafile can look like. Exact specification is subject of further research in the PhD thesis.

5.2 Language of Analysis

The language of analysis is a cardinal element of the framework. It determines fundamental features of any analyzer and therefore requires careful design. Let us repeat its main purpose:

> The language of analysis is a notation for efficient finding or description of various data relation within log files, or more generally, finding unknown relations in heterogenous tables.

This is the language of user-written filter and analytical plug-ins. A well-designed language should be powerful enough to express easily all required relations using an abstract notation. Simultaneously, it must allow feasible implementation in available programming languages.

The following paragraphs present a draft of such language. They sequentially describe the principle of operation, basic features and structures. A complete formal description will be developed during further work.

5.2.1 Principle of Operation And Basic Features

The language of analysis is of a data-driven programming style that eliminates usage of frequent program loops. Basic operation is roughly similar to AWK as it is further explained.

Programs are sets of *conditions* and *actions* that are applied on each record of given log file. Conditions are expressions of boolean functions and can use abstract data types. Actions are imperative procedures without direct input/output but with ability to control flow of data or execution.

Unlike AWK, there is no single-pass processing of the stream of log records. The analyzer takes conditions one by one and transforms them into SQL commands that select neccessary data into a temporary table. The table is then processed record–by–record: each time, data from one record are used as variable values int he respective action and the action is executed. The type system is weak, variable type is determined by assigned values.

The language of analysis defines four different sections:

1. An optional FILTER section that contains a set of conditions that allow selective filtration of log content. FILTER section are stored separately in filter plug-in. Filter plug-ins use different syntax to analytical plug-ins. More detailed description is available later in this chapter.

2. An optional DEFINE section, where output variables and user-defined functions are declared or defined. The log content is already accessible in a pseudo-array using identifiers declared in metafile.

3. An optional BEGIN section that is executed once before 'main' program execution. BEGIN section can be used for initialization of some variables or for analysis of 'command line' arguments passed to the plug-in.

4. The 'MAIN' analytic program that consists of ordered pairs [condition; action]. Rules are written in form of boolean expressions, actions are written in form of imperative procedures. All log entries are step-by-step tested against all conditions, if the condition is satisfied then the action is executed.

   In contrary to AWK, the conditions and actions manipulate data at high abstraction level, i.e. they use variables and not lexical elements like AWK does. In addition, both conditions and actions can use a variety of useful operators and functions, including but not limited to string and boolean functions like in AWK.

   Actions would be written in pseudo–C imperative language.

5. An optional END part that is executed once after the 'main' program execution. Here can a plug-in produce summary information or perform final, close-up computation.

DECLARE, BEGIN, 'MAIN' and END sections make together an analytical plug-in module.

5.2.2 Data Filtering and Cleaning

Data filtering and cleaning is simple transformation of a log file to reduce its size, the reduced information is of the same type and structure. During this phase, an analyzer has to determine what log entries should be filtered out and what information should be passed for further processing if the examined record successfully passes filtration.

The filtration is performed by testing each log record against a set of conditions. A set of conditions must be declared to have one of the following policies (in all cases there are two types of conditions, 'negative' and 'positive'):

- **order pass, fail** that means that all *pass* conditions are evaluated before *fail* conditions; initial state is FAIL, all conditions are evaluated, there is no shortcut, the order of conditions is not significant
- **order fail, pass** that means that all *fail* conditions are evaluated before *pass* conditions; initial state is PASS, all conditions are evaluated, there is no shortcut, the order of conditions is not significant
- **order require, satisfy** that means that a log record must pass all *require* conditions before but including the first *satisfy* condition (if there is any); the record fails filtration when it by the first unsatisfied condition; the order of conditions is significant

The conditions are written by common boolean expressions and use identifiers from metadata. They can also use functions and operators from the API library. Here are some examples of fragments of filter plug-ins (they assume the sample log file described on page 33):

```
#Example 1: messages from alba not older 5 days
order pass,fail
pass host=="alba"
pass (time-NOW)[days]<5


#Example 2: problems with hackers and secure shell
order fail, pass
fail all
pass (host=="beta")&&(program=="sshd")
pass (program=="sshd")&&(address.dnslookup IN "badboys.com")


#Example 3: heater test
order require,satisfy
satisfy temperature>30    # if temperature is too high -> critical
require temperature>25    # if temperature is high and ...
require status=="on"      # heater is on -> heater malfunction ???
```

There must be also a way how to specify what information should be excluded from further processing. This can be done by 'removal' of selected variables from

internal data structures. For example, the following command indicates that variables "time" and "host" are not further used:

```
omit time,host
```

5.2.3 Analytical Tasks

This paragraph further describes the language of analysis, especially features and expressions used in analytical task. Some features are illustrated in subsequent examples.

A program consists of an unordered list of *conditions* and *actions*.

Conditions are boolean expressions in C-like notation and they are evaluated from left to right. Conditions can be preceded by a label—is such case they act like procedures and can be referenced in actions. Conditions use the same syntax like actions, see bellow for more details.

Actions are imperative procedures that are executed if the respective condition is true. Actions have no file or console input and output. All identifiers are case sensitive. Here is a list of basic features:

- There are common control structures like sequence command, conditions `if-then-else`, three program loops `for, while, do-while`, function invocation etc. There are also special control commands like `abort` or `rewind`.
- Expression are a common composition of identifiers, constants, operators and parentheses. Priority of operators is defined but can be overridden by parenthesis. There are unary and binary operators with prefix, infix and postfix notation. Unary postfix operators are mainly type-converters.
- Variables should be declared prior they usage. At beginning they are empty—they have no initial value. Variable type is determined with first assignment but it can change with next assignments. There are the following basic types: `number, string, array`. Strings are enclosed in quotes, arrays use brackets for index.

  Variables are referred directly by their names. If a variable is used inside string, the name is preceded by a $ sign.

  Variable scope and visibility spans over whole program.
- Arrays are hybrid (also referenced as *hash* arrays); they are associative arrays with possible indexed access. Internally, they are represented by a list of pairs key–value. Multidimensional arrays are allowed; in fact they are array(s) of arrays. Fields are accessed via index or using internal pointer.

Index can be of type number or string. Array can be heterogenous. New field in array can be created by assignment without index or by assignment with not-yet-existing key.

They are available common array operators like `first, last, next` etc.

- Structures (record type) are internally treated as arrays; therefore there is no special type. Sample 'structures' are date, time, IP address and similar.
- Procedures and functions are written as named pairs [condition;action]. Conditions are optional. Arguments are referred within procedure a by their order, `#1` is the first one. Procedure invocation is done by its identifier followed by parentheses optionally with passed arguments. Recursion is not allowed.

There is also a set of internal variables that reflect internal state of the analyzer: current log file, current line number, current log record etc.

Like in the C language, there is a language design and there are useful functions separated in a library. There are 'system' and 'user' library. Some system functions are 'primitive' that means that they can not be re-written using this language.

Bellow there are several simple examples that illustrate basic features of the suggested language of analysis. The examples are believed to be self-explaining and they also use the same sample log file from page 33.

```
#Example 1: counting and statistics
DEFINE           {average, count}
BEGIN            {count=0; sum=0}
(temperature<>0) {count++; sum+=temperature}
END              {average=sum/count}


#Example 2: security report - ssh sessions
DEFINE           {output_tbl}
BEGIN            {user=""; msg=[]}
(program=="sshd"){msg=FINDNEXT(30,5m,(program==login));
                  user=msg[message].first;
                  output_tbl[address.dnslookup][user]++
                 }
```

```
#Example 3: heater validation by a state machine
DEFINE           {result}
BEGIN            {state="off"; tempr=0; min=argv[1]; max=argv[2]}
(temperature<>0) {tempr=temperature}
(status=="on")   {if (state=="off" && tempr<min) state="on"
                  else {result="Error at $time"; abort}
                 }
(status=="off")  {if (state=="on" && tempr>max) state="off"
                  else {output[0]="Error at $time"; abort}
                 }
END              {result="OK"}



#Example 4: finding relations
DEFINE           {same_addr,same_port}
(program=="sshd"){foo(address); bar(port)}
foo:(address==$1){same_adr[]=message}
bar:(port=$1)    {same_port[]=message}



#Example 5: finding causes
DEFINE           {causes}
BEGIN            {msg=[]}
(program=="sshd"){msg=FINDMSG(100,10m,3);
                  sort(msg);
                  for (i=0;i<msg.length; i++)
                    causes[]="$i: $msg[i][time] $msg[i][program]"
                 }
```

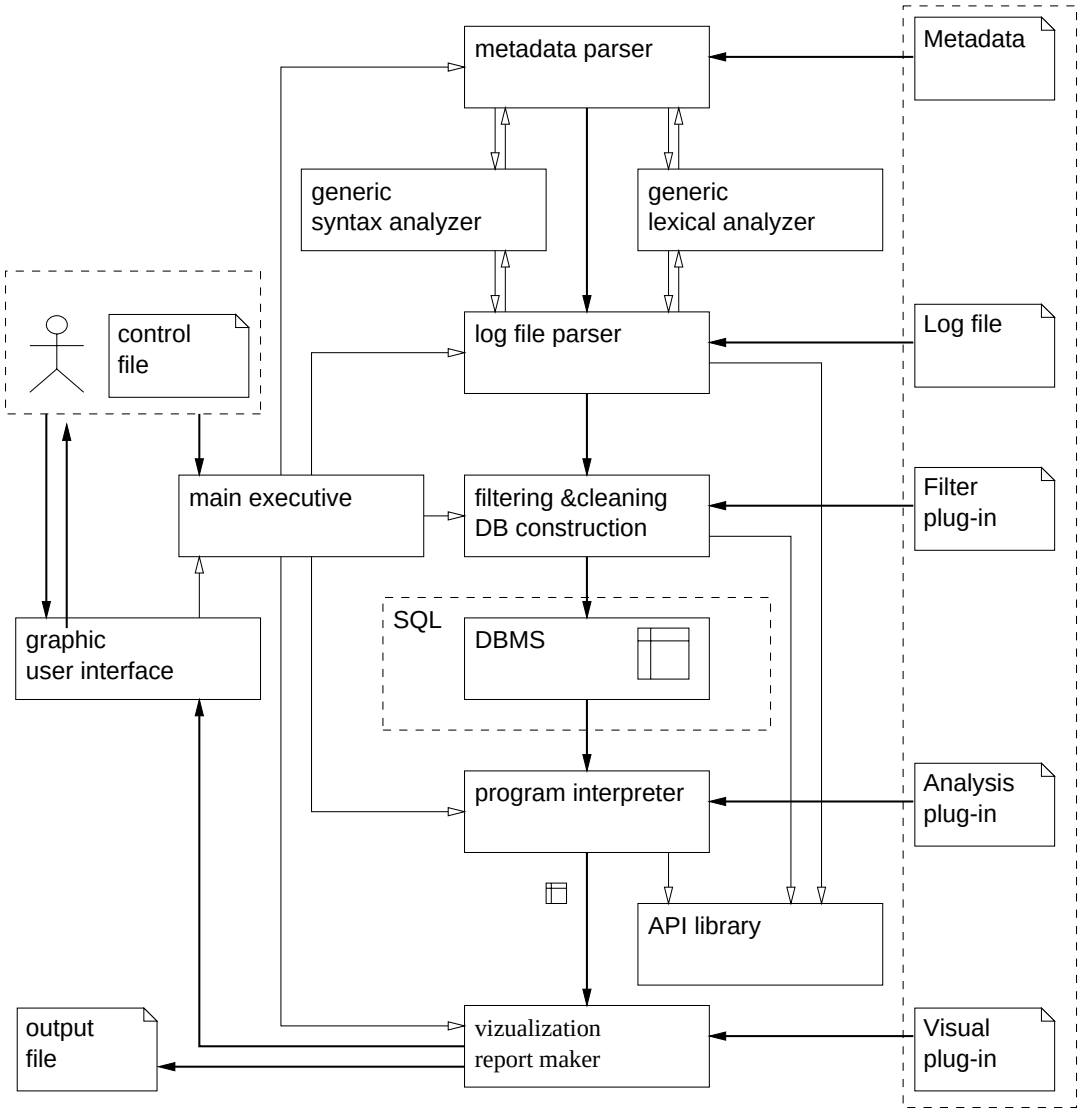
Again, the examples above just ilustrate how things can work. Formal specification is subject of further work.

5.2.4 Discussion

The proposed system of operation is believed to take advantage of both AWK and imperative languages while avoiding the limitations of AWK. In fact, AWK enables to write incredibly short and efficient programs for text processing but it is near useless when it should work with high-level data types and terms like variable values and semantics.

5.3 Analyzer Layout

The previous chapters imply that an universal log analyzer is fairly complicated piece of software that must fulfill many requirements. For the purpose of easy and open implementation, the analyzer is therefore divided into several functional modules as it is shown at the figure below:



The figure proposes one possible modular decomposition of an analyzer. It shows main modules, used files, the user, basic flows of information (solid arrows) and basic control flows (outlined arrows).

5.3.1 Operation

Before a detailed description of analyzer's components, let us enumerate the basic stages of log processing from user point of view and thus explain how the analyzer operates.

1. **Job setup.** In a typical session, the user specifies via an user interface one or more log files, a metadata file and selects modules that should participate in the job, each module for one analytical task.

2. **Pre-processing.** The analyzer then reads metadata information (using lexical and syntax analyzers) and thus learns the structure of the given log file.

3. **Filtration, cleaning, database construction.** After learning log structure, the analyzer starts reading the log file and according the metadata stores pieces of information into the database. During this process some unwanted records can be filtered out or some selected parts of each record can be omitted to reduce the database size.

4. **The analysis.** When all logs are read and stored in the database, the analyzers executes one-after-one all user-written modules that perform analytical tasks. Each program is expected to produce some output in form of a set of variables or the output may be missing—if the log does not contain the sought-after pattern etc.

5. **Processing of results.** Output of each module (if any) is passed to the visualization module that uses yet another type of files (visual plug-ins) to learn how to present obtained results. For example, it can transform the obtained variables into pie charts, bar graphs, nice–looking tables, lists, sentences etc.

   Finally, the transformed results are either displayed to user via the user interface module or they are stored in an output file.

5.3.2 File types

There are several types of files used to describe data and to control program behavior during the analysis. The following list describes each type of file:

- **Control file.** This file can be optionally used as a configuration file and thus eliminates the need of any user interface. It contains description of log source (i.e. local filename or network socket), metadata location, list of applied filtering plug-ins, analytical plug-ins and visual plug-ins.

- **Metadata file.** The file contains information about the structure and format of a given log file. First, there are parameters used for lexical and syntax analysis of the log. Second, the file gives semantics to various pieces of information in the log, i.e. it declares names of variables used later in analytical plug-ins, describes fixed and variable parts of each record (time stamp, hostname, IP address etc.).

- **Log file(s).** The file or files that should be analyzed in the job. Instead of a local file also a network socket can be used.

- **Filtering plug-in(s).** Contains a set of rules that are applied on each log record to determine whether it should be passed for further processing or thrown away. Filter plug-in can also remove some unnecessary information from each record.

- **Analytical plug-ins.** These plug-ins do the real job. They are programs that search for patterns, count occurrences and do most of the work; they contain the 'business logic' of the analysis. The result of execution of an analytical plug-in is a set of variables filled with required information.

  The plug-ins should not be too difficult to write because the log is already pre-processed. Some details about this type of plug-in are discussed later in this chapter.

- **Visual plug-in(s).** Such files contain instructions and parameters for the visualization module how to transform outputs of analytical modules. In other words, a visual plug-in transforms content of several variables (those are output of analytical plug-ins) into a sentence of graphic primitives (or. commands) for a visualization module.

- **Output file.** A place where the final results of the analytical job can be stored, but under normal operation, the final results are displayed to the user and not stored to any file.

5.3.3 Modules

Finally there is a list of proposed modules together with brief description of their function and usage:

- **User interface.** It allows the user to set up the job and also handles and displays the final results of the analysis. The user has to specify where to find the log file(s) and which plug-ins to use.

  User interface can be replaced by control and output files.

- **Main executive.** This module controls the operation of the analyzer. On behalf of user interaction or a control file it executes subordinate modules and passes correct parameters to them.

- **Generic lexical analyzer.** At the beginning a metadata file and log files are processed by this module to identify lexical segments. The module is configured and used by the metadata parser and log file parser.

- **Generic syntax analyzer.** Alike the lexical analyzer, this module is also used by both metadata parser and log file parser. It performs syntax analysis of metadata information and log files.

- **Metadata parser.** This module reads a metadata file and gathers information about the structure of correspondent log file. The information is subsequently used to configure log file parser.

- **Log file parser.** This module uses the information gained by the metadata parser. While it reads line by line the log content, pieces of information from each record (with correct syntax) are stored in corresponding variables according their semantics.

- **Filtering, cleaning, database construction.** This module reads one or more filter plug-ins and examines the output of the log file parser. All records that pass filtration are stored into a database.

- **Database management system.** This is a generic DBMS that stores tables with transformed log files. It would probably operate as a SQL wrapper to emulate data cube and OLAP functions over a classic relational database.

- **Program interpreter.** This module is the hart of the analyzer. It reads user–written modules (analytical plug-ins) and executes them in a special environment. The module would have many features, for example it provides transparent usage of the database; users access the log data in their programs using a pseudo-array of objects instead working with raw data (like records, lines, buffers etc.).

  The module also captures plug-ins output and passes it for visualization.

- **Visualization, report maker.** Analytic plug-ins produce output as a set of variables (numbers, string, arrays). The purpose of this module is to transform such results into human readable form. The module uses visualization plug-in for transformation of raw plug-in's output into graphic commands or into text reports that are later displayed to the user.

  The simplest form is a generic *variable inspector* that displays content of variables of different types in some nice, interactive and structured form.

- **API library.** This module contains many useful functions that can be used directly by user programs in plug-ins. DNS lookup, time calculation and statistical computing are some examples.

5.4 Other Issues

5.4.1 Implementation Language And User Interface

The analyzer would be written in an imperative language, most likely the C language because of its wide support and speed. Pascal and Ada have no special benefits and Java seems to be too slow. After all the intended implementation is experimental so the selection of the language is not critical.

Another posibility is to create a cross-compiler from the language of analysis to the C language. Each *action* in analytical plug-in could be converted to one C function, the `main()` function would contain a loop that gradualy performs SQL commands for *conditions* and executes respective C functions for *actions*.

Yet another challenging possibility is to use PHP language (a C derivate) in connection with web user interface. Because most of the heavy work should be done by the database back-end, there should be no strict speed limits and it could be interesting to implement the tool as a server-side application.

5.4.2 Selection of DBMS

Although the database is an important part of the analyzer, the selection of exact vendor should not be critical. The main criteria are support of sufficient subset of SQL (nested selects, joins, views etc. — exact requirements will be specified in more detail during later stages of design) and DBMS speed. In addition, the engine design could result in a single-table layout, i.e. the SQL commands could be relatively simple.

The main task of the database is to store all log information in the form of data cube, which determines usability of each DMBS. Databases of the first choice are MySQL, Postgress and Interbase, mainly because of their free license policy.

5.5 Discussion

The outlined analyzer represents one approach to the problem of log analysis, nevertheless it could be designed in a different way. The main contribution of this approach is the proposed language of analysis that in fact affects many other features of the analyzer.

Therefore this section demonstrates advantages of the proposed language of analysis in comparison to solutions based on plain imperative languages (C, Pascal) or plain data-driven language (AWK).

5.6 Comparison to C–based Language of Analysis

If the language of analysis was C, then the easiest design is to write one C function for each analytic task. It should not be too difficult and with utilization of built-in pre-processor even the interface of such function could be rather simple. There are three possibilities what data structures to use:

- To describe log format using structures and unions of standard C types, but it leads to a type–explosion and many difficulties when manipulating data of new types (library functions, overloading of operators etc.) This also prevents usage of a database.
- To use C-style strings (arrays of char type) for log content, but in such case we stuck in low abstraction level, i.e. the user functions would be a mesh of string-manipulating functions, nested conditions and program loops.
- To use heterogenous arrays for log data representation with all disadvantages of such arrays and pointer arithmetic. A database could be used via pre-processor macros or for example using in-line SQL.

The problem with C is that C is a low-level language; there are complications mainly with data types and heterogenous array manipulation. The need of execution of a C compiler is also a disadvantage. On the other hand, the advantages are speed and executable binaries for each analytic task.

5.7 Comparison to AWK–based Language of Analysis

If the language of analysis was plain AWK, then it would be very simple to write some analytical programs while some other would be very difficult.

- AWK works at low level with regular expressions, strings, substrings and other lexical elements. Although AWK support variables and functions, working at high level of abstraction (with semantic information) is at lest problematic if not impossible.
- AWK is pure data-driven language with single-pass operation; there is no possibility how to process some parts of input repeatedly.

- There are no structured data types in AWK so handling of sophisticated heterogenous log files could be complicated.

There are also other limitations that raise from the design of AWK program structure. In conclusion, AWK is a perfect tool for text file processing but it requires some improvements to fit better for easy analysis of log files.

6 Conclusion And Further Work

6.1 Summary

The paper provides an overview of current state of technology in the field of log file analysis and stands for basics of ongoing PhD thesis.

The first part covers some fundamental theory and summarizes basic goals and techniques of log file analysis. It reveals that log file analysis is an omitted field of computer science. Available papers describe moreover specific log analyzers and only few contain some general methodology.

Second part contains three case studies to illustrate different application of log file analysis. The examples were selected to show quite different approach and goals of analysis and thus they set up different requirements.

The analysis of requirements then follows in the next part which discusses various criteria put on a general analysis tool and also proposes some design suggestions.

Finally, in the last part there is an outline of the design and implementation of an universal analyzer. Some features are presented in more detail while others are just intentions or suggestions.

6.2 PhD Thesis Outline

The main objective of the ongoing PhD thesis should be design of a framework for log analysis and corresponding flexible and efficient language of analysis.

This work shall continue by further theoretical research in theory of log file analysis followed by detailed design of all mentioned parts. Results shall be finally described and verified in a formal way.

Usability of the language and feasibility of the framework should be also tested by prototype implementation of core modules, i.e. a program interpreter, filtering and cleaning, database construction, metadata description and log file parser. Remaining modules will be developed in parallel but outside PhD work.

This intention is declared in the next paragraph that stands for a proposed abstract of the thesis:

6.3 Proposed Abstract of The PhD Thesis

This PhD designs a method for universal log file analysis. At the most abstract level, the objective of the thesis is to find a formal notation for finding various data relations in heterogenous tables. At a more practical level, the proposed

method is represented by a language of analysis and supporting framework that allows efficient processing of log files at high level of abstraction and flexibility.

Finally, the feasibility of designed method is verified by a prototype implementation of a log file analysis supporting framework.

6.4 Further Work

The further work, in parallel to the PhD thesis, should proceed according to the outlined proposals, i.e. after detailed design of all necessary components it should result in implementation of whole framework. The final result is a generic log analyzer in form of a distribution package. The usability and performance of the analyzer would be finally evaluated using NetFlow log files in a real-word application.

References

[1] J. H. Andrews: *"Theory and practice of log file analysis."* Technical Report 524, Department of Computer Science, University of Western Ontario, May 1998.

[2] J. H. Andrews: *"Testing using log file analysis: tools, methods, and issues."* Proc. 13 th IEEE International Conference on Automated Software Engineering, Oct. 1998, pp. 157-166.

[3] J. H. Andrews: *"A Framework for Log File Analysis."* `http://citeseer.nj.nec.com/159829.html`

[4] J. H. Andrews, Y. Zhang: *"Broad-spectrum studies of log file analysis."* International Conference on Software Engineering, pages 105-114, 2000

[5] J. H. Andrews: *"Testing using Log File Analysis: Tools, Methods, and Issues."* available at `http://citeseer.nj.nec.com`

[6] L. Lamport: *"Time, Clocks, and the Ordering of Events in a Distributed System."* Communications of the ACM, Vol. 21, No. 7, July 1978

[7] K. M. Chandy, L. Lamport: *"Distributed Snapshots: Determining Global States of Distributed Systems."* ACM Transactions on Computer Systems, Vol. 3, No. 1, February 1985

[8] F. Cristian: *"Probabilistic Clock Synchronization."* 9th Int. Conference on Distributed Computing Systems, June 1989

[9] M. Guzdial, P. Santos, A. Badre, S. Hudson, M. Gray: *"Analyzing and visualizing log files: A computational science of usability."* Presented at HCI Consortium Workshop, 1994.

[10] M. J. Guzdial: *"Deriving software usage patterns from log files."* Georgia Institute of Technology. GVU Center Technical Report. Report #93-41. 1993.

[11] Tec-Ed, Inc.: *"Assessing Web Site Usability from Server Log Files White Paper."* `http://citeseer.nj.nec.com/290488.html`

[12] Osmar R. Zaane, Man Xin, and Jiawei Han: *"Discovering web access patterns and trends by applying OLAP and data mining technology on web logs."* In Proc. Advances in Digital Libraries ADL'98, pages 19–29, Santa Barbara, CA, USA, April 1998.

[13] Cisco Systems: *"NetFlow Services and Application."* White paper. Available at `http://www.cisco.com`

[14] Cisco Systems: *"NetFlow FlowCollector Installation and User Guide."* Available at `http://www.cisco.com`

[15] D. Gunter, B. Tierney, B. Crowley, M. Holding, J. Lee: *"Netlogger: A toolkit for Distributed System Performance Analysis."* Proceedings of the IEEE Mascots 2000 Conference, August 2000, LBNL-46269

[16] B. Tirney, W. Johnston, B. Crowley, G. Hoo, C. Brooks, D. Gunter: *"The Netlogger Methodology for High Performance Distributed Systems Performance Analysis."* Proceedings of IEEE High Performance Distributed Computing Conference (HPDC-7), July 1998, LBNL-42611

[17] Google Web Directory: *"A list of HTTP log analysis tools"* `http://directory.google.com/Top/Computers/Software/Internet/ /Site_Management/Log_Analysis`

[18] J. Abela, T. Debeaupuis: *"Universal Format for Logger Messages."* Expired IETF draft `<draft-abela-ulm-05.txt>`

[19] C. J. Calabrese: *"Requirements for a Network Event Logging Protocol."* IETF draft `<draft-calabrese-requir-logprot-04.txt>`

[20] SOFA group at The Charles University, Prague. `http://nenya.ms.mff.cuni.cz/thegroup/SOFA/sofa.html`

[21] SOFA group at The University of West Bohemia in Pilsen. `http://www-kiv.zcu.cz/groups/sofa`

[22] Plasil F., Visnovsky S., Besta M.: *"Behavior Protocols and Components."* Tech report No. 99/2, Dept. of sw engineering, Chasrles University, Prague

[23] Hlavicka J., Racek S., Golan P., Blazek T.: *"Cislicove systemy odolne proti porucham."* CVUT press, Prague 1992.

[24] The GNU Awk `http://www.gnu.org/software/gawk`

[25] The GNU Awk User's Guide `http://www.gnu.org/manuals/gawk`

Author's Publications And Related Activities

[ISM–00] J. Valdman: *"SOFA Approach to Design of Flexible Manufacturing Systems."* Information Systems Modeling Conference (ISM'00), Roznov pod Radhostem, 2000.

[ICCC–2000] A. Andreasson, P. Brada, J. Valdman: *"Component-based Software Decomposition of Flexible Manufacturing Systems"*. First International Carpatian Control Conference (ICCC'2000), High Tatras, Slovak Republic, 2000.

[ISM–01] J. Rovner, J. Valdman: *"SOFA Review – Experiences From Implementation."* Information Systems Modeling Conference (ISM'01), Hradec nad Moravici, 2001.

[AINAC–2001] J. Valdman: *"MAN at The University of West Bohemia."* Invited speech at First Austrian International Network Academy Conference (AINAC), Inssbruck, Austria, 2001.

[DCSE–1] J. Valdman: *"Means of Parallelization in Higher Programming Languages."* A study report, Pilsen 2000.

[DCSE–2] J. Valdman: *"WWW Proxy & Cache."* A study report, Pilsen 2000.

[DCSE–3] P. Hanc, J. Valdman: *"SOFA Techreport."* Technical report of the Department of Computer Science And Engineering, University of West Bohemia, Pilsen 2001.