UNIVERSITY OF TARTU
FACULTY OF MATHEMATICS AND COMPUTER SCIENCE
Institute of Computer Science
Computer Science Curriculum

Marti Taremaa

Off-line Synchronization of Trace Logs

Master's thesis (20 cp)

Supervisors: Ulrich Norbisrath, Ph.D.
prof. Eero Vainikko, Ph.D.

Author: ..... 27th of May, 2008

Supervisors: ..... 27th of May, 2008

..... 27th of May, 2008

Approved

Professor Eero Vainikko: ..... 27th of May, 2008

TARTU 2008

Acknowledgements

This thesis exists in its current form thanks to the support of many people.

I thank my colleagues and fellow students from the Distributed Systems group. Moving towards the completion is much easier when being surrounded with people with the same goals. Special thanks go to my peer review group members from the "Software Engineering in Distributed Systems" seminar.

I thank my supervisors for guiding me towards the solutions and providing extensive feedback. Eero, you have built up a capable team with such a short time and I am glad to be part of it. Ulrich, your contribution to my thesis cannot be overestimated. I already knew to expect a lot of support, but your feedback was just superb.

Finally, I thank my friends and family for offering me a haven to regain my energy and for keeping me sane over the times of peak load.

May 2008

Contents

1 Introduction **7**

2 Debugging in Distributed and Deployed Systems **11**

2.1 Requirements and Challenges 11

2.2 Overview of Existing Tools 14

3 Off-line Synchronization of Trace Logs **29**

3.1 Case Studies: Debugging Deployed and Distributed Systems . 29

3.2 The Concept 31

3.3 Advantages and Disadvantages 34

4 The TraceBrowser **37**

4.1 Functionality of TraceBrowser 37

4.2 Structure, Algorithms and the User Interface 43

4.3 Memory Usage and Performance Evaluation 46

4.4 Future Development 50

5 Conclusion **53**

Chapter 1

Introduction

When networked and distributed applications became central and critical parts of our everyday information infrastructure, rather than just being experimental tools for small user groups, the need for getting and keeping them stable rose significantly. The software developers and system administrators are now facing a challenge of delivering high-quality distributed services under growing user loads. One part of the process of delivering a stable application is the task of debugging. Debugging is a systematic process of finding and reducing the number of defects (often referred to as *bugs*) in a computer program.[29]

This thesis focuses on the debugging in distributed and deployed systems. A special debugging methodology suitable for distributed and deployed systems is described. This methodology is developed by combining and systemizing existing debug approaches and tools. A new tool to simplify the analysis stage of distributed and deployed application debugging – The TraceBrowser – is presented.

A typical debugging tool - a *debugger* - would allow the user to monitor code execution, step manually through it and see and change the contents of variables. An average debugging session starts with observing the execution of program code, collecting trace data (function calls, inter-process communication) and observing the contents of specific variables or making larger memory snapshots. Based on this collected data, conclusions and corrections can be done either on-line (while a program is running) or off-line. As a result, software bugs or performance bottlenecks are located and possibly eliminated. The process can be repeated until the quality of the application is satisfactory. Debuggers that are limited to only tracing the execution of program code, are often called *tracers* or *trace debuggers*. The output of the

trace debugger is called the *trace log.*

Distributed systems are systems consisting of several cooperating processes or applications which, in most cases, are communicating over the network. Some sources are more restrictive in their definition for distributed system, stating that the cooperating processes in a distributed system must be semi-independent parts of the same program, possibly running in different environments. These sources refer to systems described in the first definition as *networked systems.*

Deployed systems are systems that have been deployed into the production environments and are operating under a strict set of rules. In such systems, the set of possible maintenance, testing, and debugging operations is limited. On the other hand, when problems occur in the deployed systems, there is more pressure to find and eliminate their cause. That is why debugging in deployed systems is different and must be specially handled.

I started my research on the topic of debugging networked and distributed systems for practical reasons. When working as a system administrator, I often had to pinpoint bugs in deployed systems. These systems consisted of different applications which were communicating over the network. Not all of them would have qualified as distributed systems, but many of the problems I encountered in the debugging process were similar to those encountered in distributed systems. Further I learned that the successful debugging methods for both networked and distributed systems were very similar. This rose an interest in the field of debugging distributed systems.

The chapter two of my thesis focuses on how debugging in distributed systems and in production environments differs from stand-alone application debugging. It also gives an overview of some of the existing tools for both distributed and networked application debugging. Some single-application debug tools that can be used for debugging distributed and deployed applications will also be covered.

The chapter three gives an overview of debugging methodology Off-line Synchronization of Trace Logs which I have composed from different best practices in distributed and production environment application debugging. This methodology has been developed and adapted to best match the needs of system administrators who are working with live production systems. Some example cases that can be effectively handled by this methodology are made.

The fourth chapter introduces the TraceBrowser application which I have developed to support the Off-line Synchronization of Trace Logs methodology. TraceBrowser is a graphical application for parallel analysis several

timestamped trace logs. TraceBrowser allows the user scroll through logs synchronously, constructing global temporal order for the log events.

The conclusion shortly summarizes the work done so far and outlines future directions for the development of the TraceBrowser utility.

Appendix A contains a short user guide for the TraceBrowser.

The source code of the TraceBrowser is linked in this thesis in the the Appendix B (on page 67).

Chapter 2

Debugging in Distributed and Deployed Systems

There are many different approaches to debugging. Some of them complement each other. In deployed systems, not all of these approaches and the corresponding tools can always be used, because some of them induce unacceptable load overhead or require the application to be paused or restarted by the debugger. Distributed systems introduce some new debugging challenges and new tools are needed for handling the extra complexity. However, besides the special tools, debugging of distributed and deployed systems can still reuse debug methodologies and tools created for single application debugging. There are several aspects that must be kept in mind, and if possible, taken care of, while using existing single-application debug tools on distributed and production environment applications. This chapter will first describe the most significant challenges that have to be faced when debugging distributed and deployed systems, followed by the overview of some existing debug tools that are used or have been created for distributed debugging.

2.1 Requirements and Challenges

The components of distributed systems are located on different devices, therefore the execution traces must also be acquired from multiple locations. Tracing the parts of the same distributed application at different locations is called *multi-point tracing*. Sometimes the tracer tools that are used to perform multi-point tracing are totally independent and not aware of each other. However, to perform better analysis on collected traces, we must gather and

process them together. For many debuggers, the goal is to achieve *global overview* or *global comprehension* - the ability to inspect the state of any component of distributed application at any point in time during the debug process [9, 11]. In the real world, this goal is not always fully reached, but the debugging frameworks must always try to get closer to it.

Debugging utilities increase the system load. This is especially true when tracing distributed applications, because several applications must be simultaneously traced, usually meaning one tracer process for every debugged process must be present. In systems deployed into production environments, special care must be taken to ensure that this extra load – often referred to as *debug overhead* – remains minimal. When debugging applications in a test environment, less attention can be paid to tracing overhead, but even then too much extra load can have an effect on how a debugged system acts or performs.

There are two widely-used debugging approaches that usually are not applicable when debugging live production systems - *synchronous and intrusive debugging*. When debugging synchronously, the application can be paused when errors, watchpoints or breakpoints occur. In an intrusive debug session, it is possible to change the internal state of the application during this pause. It is clear that in production environments, these kinds of actions are unthinkable. This implies the fact that correctly following some *on-line* (real-time) debugging session of deployed application can easily get beyond human abilities – the flow of events is just too fast. Therefore tools must be able to perform the sessions *off-line*, where the debug trace logs obtained from an application running in the production environment can later be stepped through at a slower pace. Off-line debug session has absolutely no effect on the application under observation.

The process of multi-point tracing in distributed systems can produce vast amounts of log data, parts of which will be of no interest to the user. It is vital that analyzer tools provide an option to filter out non-relevant trace records and emphasize relevant ones. Of course, deciding what is relevant and what is not, becomes an art in itself. Therefore the filters must be user-configurable.

As the parts of the distributed applications can have very different tasks in the system, the best tools for debugging them can be different, too. It also is not uncommon to have some components of the distributed system that are closed-source. This makes debugging them even harder, because there is no source code to use as a reference. Sometimes it is better to use specific tools for debugging these closed components. Therefore it is desirable to

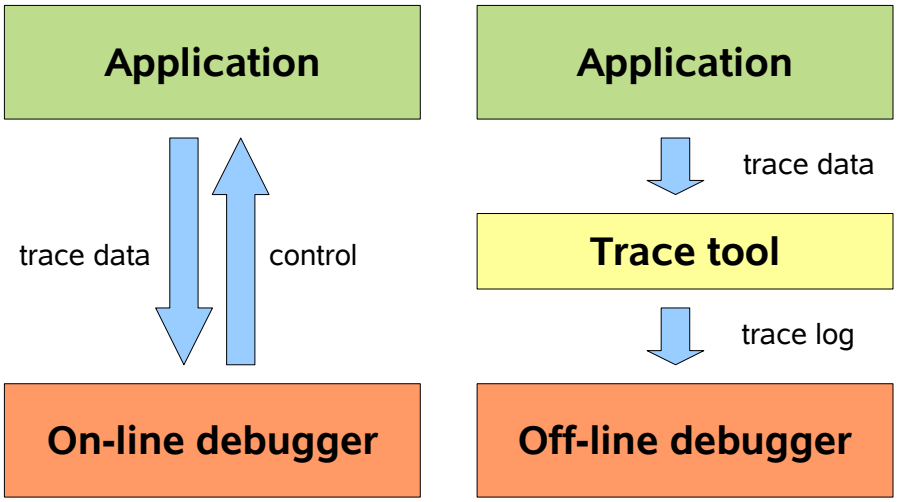

Figure 2.1: On-line and off-line debugging

have a framework which would support many different tools. For example, we could use *symbolic debuggers* – tools which correlate the execution trace of the program with the source code – for open source components. For closed source components, *library trace tools*, capturing the library calls that program makes, can be used. *Network packet loggers* can be used to capture the communication between these components. For the debugging process to be more effective, these tools would have to be centrally coordinated. At least, there should be a component which collects the output of these different tools to a central location and then performs an analysis on the collected data.

When replaying or just browsing through collected traces it is essential to have trace log records in the same global order as they occurred in the system during debugging. When replaying, mechanisms like Lamport Clocks [15] can be used to ensure consistent replay, but some trace-analyzing applications have to rely solely on hardware clocks, because inserting your own code into running applications is not always possible. Computer clocks, of course, can be synchronized using NTP [17]. In many cases this is enough [27], but when large numbers of events occur in a very small time-frame, or trace applications experience abnormal slowdowns due to a bug in the debugged

```
1211710799.236474 connect(4, {sa_family=AF_FILE, path="/var/run/nscd/socket"}, 110) = 0
1211710799.236709 poll([{fd=4,events=POLLOUT|POLLERR|POLLHUP,revents=POLLOUT}],1, 5000) = 1
1211710799.236887 writev(4, [{"\2\0\0\0"..., 12}, {"sonett.mt.ut.ee\0", 16}], 2) = 28
1211710799.237601 poll([{fd=4,events=POLLIN|POLLERR|POLLHUP,revents=POLLIN|POLLHUP}],1,5000)
= 1
1211710799.237642 read(4, "\2\0\0\0\1\0\0\0\20\0\0\2\0\0\0\4\0\0\0\1\0\0"..., 32) = 32
1211710799.237695 readv(4, [{"sonett.mt.ut.ee\0", 16}, {"\301(%\376", 4}], 2) = 20
1211710799.237738 close(4) = 0
1211710799.237787 connect(3,sa_family=AF_INET,sin_port=htons(5781),sin_addr=\
inet_addr("...")},16) = 0
1211710799.238107 send(3, "/tmp/test\0", 10, 0) = 10
1211710799.238437 recv(3, "Nonexistent file:  /tmp/test\n\0", 255, 0) = 29
```

Figure 2.2: sample of `strace` output

system, the global order of the timestamped log messages could be mixed up. It would be useful to have analysis tools that can adjust the order of the collected records either automatically (by detecting some communication patterns, for example) or manually, based on user interaction. The results of my attempt to create such a tool will be presented in the fourth chapter. The third chapter will describe a debug methodology that this tool supports. To create a background for the next chapters, a selection of existing debug tools will be presented in the following section.

2.2 Overview of Existing Tools

2.2.1 strace and ltrace

`strace` is a tool for tracing system calls made and signals received by applications[23]. In Solaris, and possibly some other Unices, `truss` - a tool similar to `strace` - can be used. MacOS X also has a similar tool, called `ktrace`. `strace` is especially useful for detecting I/O errors – what files or sockets is the application trying to open, what are the results. `strace` can be attached to and detached from a running process without having to restart it. Of course, running the program under `strace` will slightly degrade the performance, but in many systems running in production environments, this debug overhead is acceptable when introduced for limited periods. `strace` can timestamp log records it is producing (see Figure 2.2), so it is a good candidate for multi-point tracing with an intention to synchronize the log data off-line, later on.

On many occasions, errors happen on library boundaries instead of in the kernel interface, and `strace` is not the right tool for detecting them. In situations like this, `ltrace` can be used [4]. `ltrace` is a tool very similar

```
1211710762.180311 socket(2, 1, 0) = 3
1211710762.180387 gethostbyname("\377\377"...)  = 0xb7fc7a64
1211710762.181049 atoi(0xbfb6ee30, 1, 0, 0, 0) = 33882
1211710762.181113 htons(33882, 1, 0, 0, 0) = 23172
1211710762.181171 connect(3, 0xbfb6e6a0, 16, 0, 0) = 0
1211710762.181442 send(3, 0xbfb6ee36, 10, 0, 0) = 10
1211710762.181930 memset(0xbfb6e5a1, '\000', 255) = 0xbfb6e5a1
1211710762.181986 recv(3, 0xbfb6e5a1, 255, 0, 0) = 29
```

Figure 2.3: sample of `ltrace` output

to `strace`, but it traces dynamic library calls instead of system calls (See Figure 2.3). `ltrace` is capable of logging system calls, too, so it can be used in the place of `strace`. The downside is that you will find `ltrace` or some equal tool on smaller number of platforms.

2.2.2 tcpdump

When working with distributed and networked applications, having a tool for observing network communications is mandatory. Many people use `tcpdump` (on Solaris, `snoop` can be used) for this purpose. `tcpdump` is a universal network debugging tool that can be used to capture network traffic with several levels of detail [26]. Actually, the name "`tcpdump`" can be a bit misleading, because it can capture a lot more than just TCP segments: Ethernet packets, UDP datagrams, IP and ARP packets, etc. Like `strace` and `ltrace`, `tcpdump` can timestamp logged packets (see Figure 2.4). It features a powerful filtering system. If the flow of packets matching the filter expression is too fast for on-line observation, `tcpdump` can offer more than just saving the (timestamped) output and filtering it off-line with some other tools - it can do raw packet capture to a file. This capture file can later be played back off-line by `tcpdump`, as if it were an on-line session, but the difference is that the user can change the filter expression and play it back again as many times as needed.

Some GUI analyzers for the `tcpdump` capture files exist, of which Wireshark [7] is the most well known. Wireshark allows the user to navigate through a compact packet log and see detailed information of selected packets (see Figure 2.5).

2.2.3 GDB - The GNU Project Debugger

GDB, The GNU Project Debugger, is a powerful source code level debugger for programs written in C and C++. With some limitations, it supports

```
1211708495.678957 arp who-has 193.40.36.6 tell 193.40.37.254
1211708495.679057 arp reply 193.40.36.6 is-at 00:06:3e:68:64:87
1211708502.359485 IP 193.40.37.254.33319 > 193.40.36.6.3456:  udp/vat 4294967289 c0 0 s10
1211708502.359665 IP 193.40.36.6.22 > 193.40.37.254.56216:  P 7569:7617(48) ack 1536 win
112 <nop,nop,timestamp 2197380196 794282346>
1211708502.359677 IP 193.40.37.254.56216 > 193.40.36.6.22:  .  ack 7617 win 31776
<nop,nop,timestamp 794285110 2197380196>
1211708508.702836 IP 193.40.37.254.33319 > 193.40.36.6.3456:  udp/vt 53 372 / 25
1211708508.703043 IP 193.40.36.6.22 > 193.40.37.254.56216:  P 7617:7713(96) ack 1536
win 112 <nop,nop,timestamp 2197381782 794285110>
1211708508.703056 IP 193.40.37.254.56216 > 193.40.36.6.22:  .  ack 7713 win 31776
<nop,nop,timestamp 794286696 2197381782>
```

Figure 2.4: `tcpdump` packet capture tool sample output

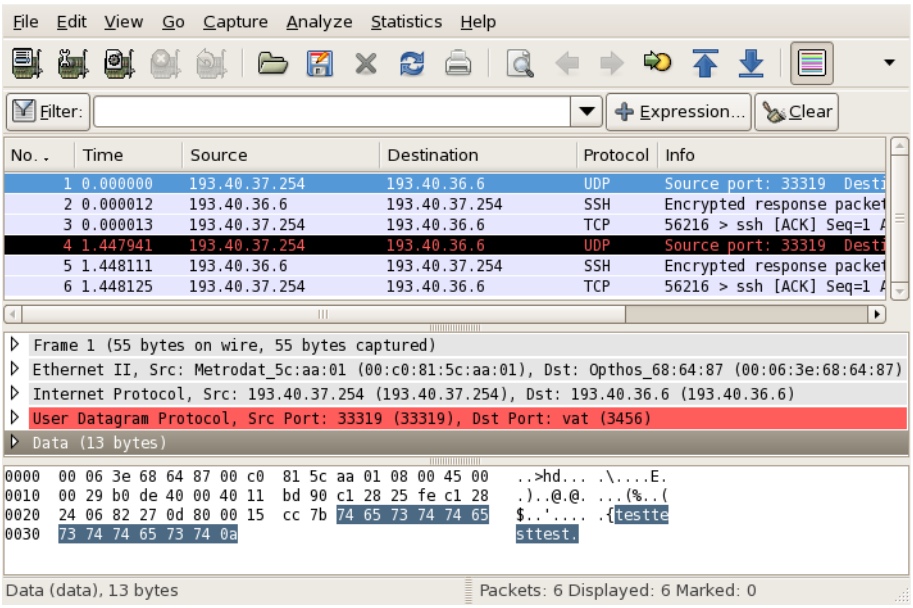

Figure 2.5: Viewing a `tcpdump` packet capture file in Wireshark

```
GNU gdb 6.6
Copyright (C) 2006 Free Software Foundation, Inc.
...
(gdb) break 32
Breakpoint 1 at 0x80487ba:  file client.c, line 32.
(gdb) run localhost 33882 /tmp/test
Starting program:  /foo/bar/client localhost 33882 /tmp/test
Breakpoint 1, main (argc=4, argv=0xbffda384) at client.c:32 32 servAddr.sin_port =
htons(atoi(argv[2]));
(gdb) print argv[2]
$1 = 0xbffdbe18 "33882"

(gdb)
```

Figure 2.6: A sample GDB session

programs written in Fortran. Support for Pascal and Modula-2 is partially available [8]. To fully use the features offered by GDB, programs under observation must be recompiled to support debugging. This makes GDB less useful in the cases where source code is not available. Re-compiling some components of live production systems may also turn out to be infeasible. A short sample debug session with GDB is shown in Figure 2.6. GDB is usually used as an intrusive, on-line symbolic debugger, but it also features commands for capturing trace data and analysing the capture data later on.

In the cases where re-compiling the code is possible and the debug overhead is acceptable, GDB is a very capable tool. It is by far the most popular debugger amongst free software developers. Many debugging suites that, unlike GDB, have graphical user interfaces, actually use GDB as their internal "debugging engine".

2.2.4 NightStar LX

NightStar LX is being developed by the Concurrent Computer Corporation. NightStar LX is actually a collection of tools. Components that belong to the NightStar LX tool-set are:

- NightView source-level debugger,
- NightTrace event analyzer,
- NightProbe data monitor,
- NightTune system and application tuner. [5]

All of the tools in NightStar LX have graphical user interfaces.

The Concurrent Computer Corporation does not declare the NightStar tool-set to be specially intended for debugging distributed systems. The focus is on time-critical applications instead. Nevertheless, the tools in the NightStar tool-set have many properties that are useful in distributed application debugging.

In the context of this thesis, NightView and NightTrace are the most relevant of the NightStar LX tools and I will describe them in a more detail.

The NightView source-level debugger offers typical features, like breakpoints, stepping through code, symbolic debugging, but also some advanced features like *hot patching* – changing the code while debugging it, and application-speed eventpoint conditions – debug conditions that may include time constraints. Debugged application must be compiled with debug information. There is no special support for distributed applications running on different hosts, but debugging multiple processes is supported. NightView supports C/C++, Fortran, and Ada. The debugging environment can be mixed, meaning that some of the code being debugged can be written in one language and some of the code can be written in another language.

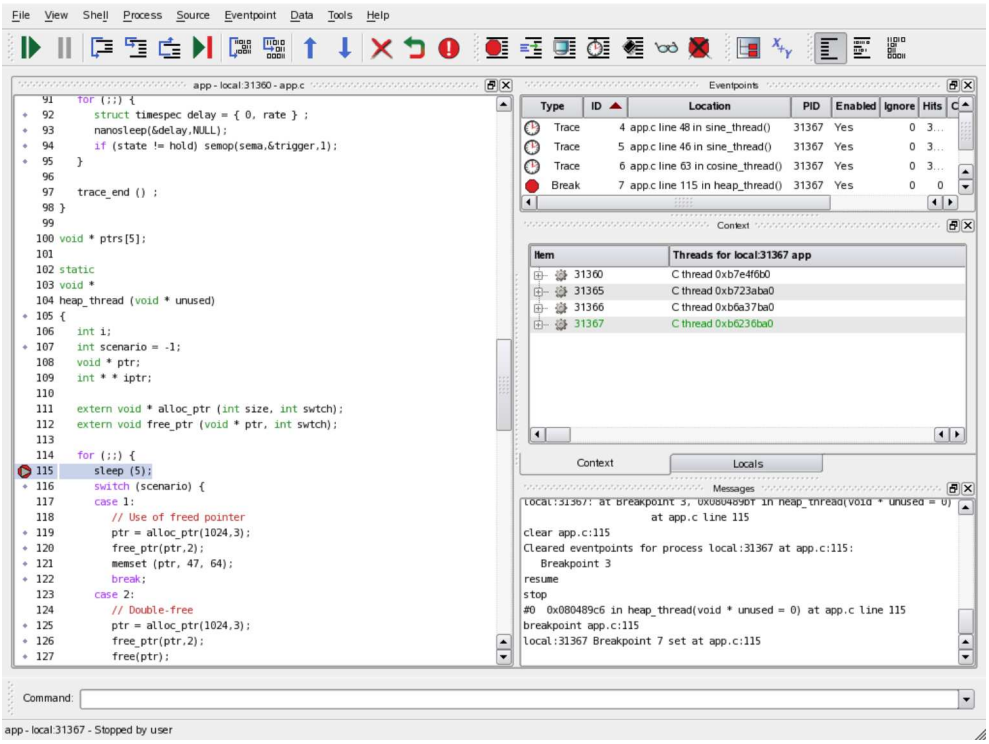

Figure 2.7: NightView user interface

NightTrace is a graphical event analyzer that can be used to visualize user-defined events, called *tracepoints*, in many different ways. Tracepoints can store values of some variables from the code being debugged so you can, for example, visualize how the value of some variable changes through execution of the code. Applications monitored by NightTrace have to be linked with the NighTrace API library. NightView and NightTrace are closely integrated - NightView allows to define tracepoints on the code being debugged. The tracepoints could also be written directly into your code. NightTrace can run event collecting daemons on multiple hosts.

While it is probably possible to find tools that could outperform NightStar in many of the areas altogether, there is, to my current knowledge, no integrated debugging package that could match the NightStar's overall performance and functionality. It also installs with minimal effort, ships with a very good user manual which includes both tutorials for new users and many details for experienced users, and has a customizable user interface built with the Qt toolkit [3]. Of course, all this comes with an appropriate price tag. And in the area of distributed application debugging, using just NightStar tools is probably not enough.

2.2.5 Allinea Distributed Debugging Tool

The Allinea Distributed Debugging Tool (DDT) is a graphical distributed debugger developed by Allinea Software [22]. It supports software written in C, C++, and Fortran (including Fortran 90/95/2003). The code being debugged must be compiled with debug information. Multi-process applications debugged with DDT must use MPI. Allinea DDT also supports debugging of jobs submitted to some kind of queue/batch systems.

DDT allows the user to debug distributed MPI applications on-line, synchronously, and at source level. It is an intrusive debugger - you can pause the process, then change the contents of the variables or data structures being watched and un-pause the process. Basically, DDT starts up multiple symbolic debuggers (on different hosts, if needed), and then coordinates them. The state of the debuggers is reflected in the DDT user interface. DDT's user interface has windows for source code, watchpoints, breakpoints, variable view, registry values, call stack, etc. Preparing complicated MPI debug sessions can be quite an effort. In DDT, the settings of the debug sessions can be saved.

For a programmer who writes distributed applications using MPI, DDT is a debugger definitely worth taking a look at. Those looking for a single-process

debugger with an easy-to-use graphical user interface, would possibly benefit from using DDT, too. In the cases where debug information and source code packages are available for these applications, Allinea DDT can give a more in-depth view of the process than the trace tools could. Like the NightStar LX tool-set, Allinea DDT is available commercially.

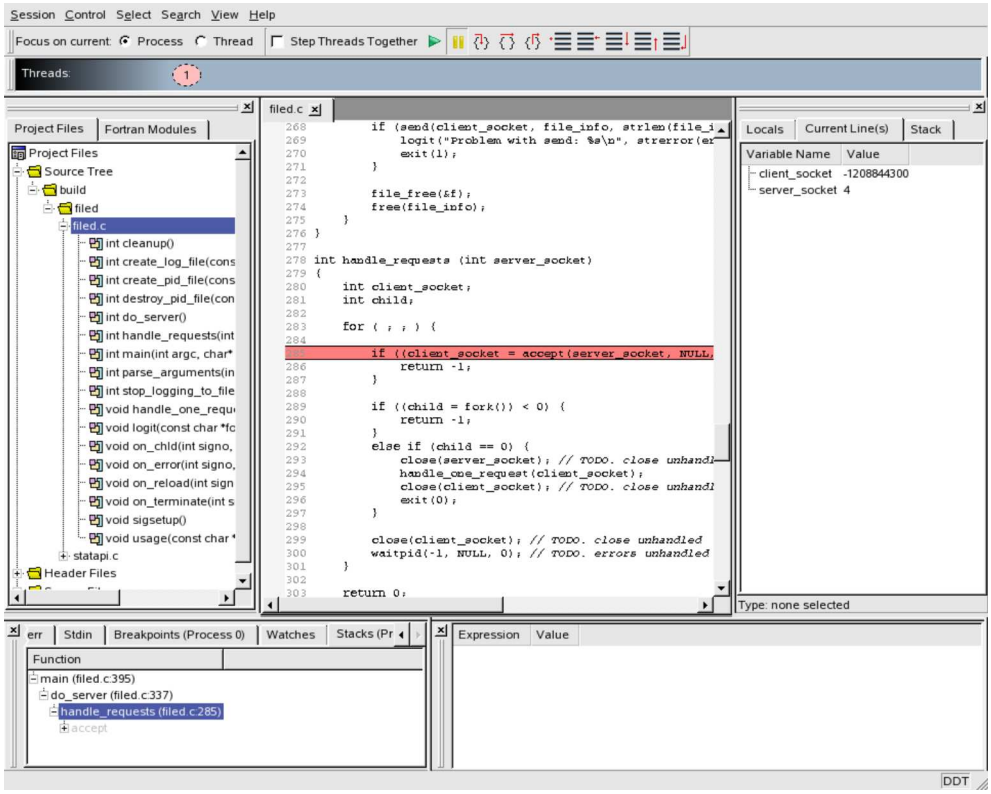

Figure 2.8: Allinea DDT user interface

2.2.6 NetLogger Toolkit

NetLogger is a methodology for performance analyzing and debugging distributed systems. A set of tools to support this methodology (the NetLogger Toolkit) is also provided by authors of the methodology, but in theory you could just follow the methodology and use your own tools for creating, collecting and analyzing logs. [27]

For most of the other tools and approaches, described or mentioned in this thesis, the main purpose is to reduce the amount of bugs in the application functionality. Improving performance of an otherwise normally-functioning

program is not their main goal. This is different for the NetLogger methodology. Here the main aim is to find out if the distributed application has notable performance bottlenecks, and where these bottlenecks occur.

There are four components in the NetLogger Toolkit: API and libraries for programming languages (C, Java, Python, ...), a set of tools for generation, collecting and sorting of event logs, a system for event archiving, and a visualization tool. To use NetLogger, you must modify your distributed application components to produce timestamped event logs at critical points. Event logs are then collected and correlated with the NetLogger visualization tool [12].

NetLogger uses the Universal Logger Message (ULM) [1] format. The ULM format consists of required fields (for example, `DATE` field for timestamp) and user-defined fields. NetLogger introduces some extra fields which must always be present, but from ULM standard's point of view these are ordinary user-defined fields. All the components of the distributed system being monitored must be modified to output their log messages in the ULM format. To make this modification easier, API and support libraries are provided.

Producing the logs in common format is one of the first steps in the NetLogger methodology. Usually, the following step is to filter and collect the produced logs. For this, the NetLogger Toolkit uses two kinds of agents: agents, who start and stop monitoring, and brokers, who control these agents and collect and filter the logs. The agent architecture helps to keep the volumes of log data under control by activating logging on-the-need basis.

The final phase in the NetLogger methodology is the analysis of the collected event logs. For this, the NetLogger toolkit offers a graphical application called `nlv` - NetLogger Visualization. The visualization tool plots time against a set of events from the applications. The set of sequential events from one process is called a *lifeline*. For example, a lifeline of an unsuccessful file I/O operation could consist of the following events: opening the file, writing the data, getting a write error, closing the file.

The `nlv` tool allows the user to choose, which events from which server should be displayed. Then the user can zoom in on areas of interest. `nlv` can work both off-line and on-line. A Sample of the `nlv` user interface can be seen on Figure 2.9.

At the current status, `nlv` is more of a experimental tool to illustrate the NetLogger approach. It is, like the NetLogger methodology, a good subject to explore and to learn from, but not a tool for every-day practical tasks.

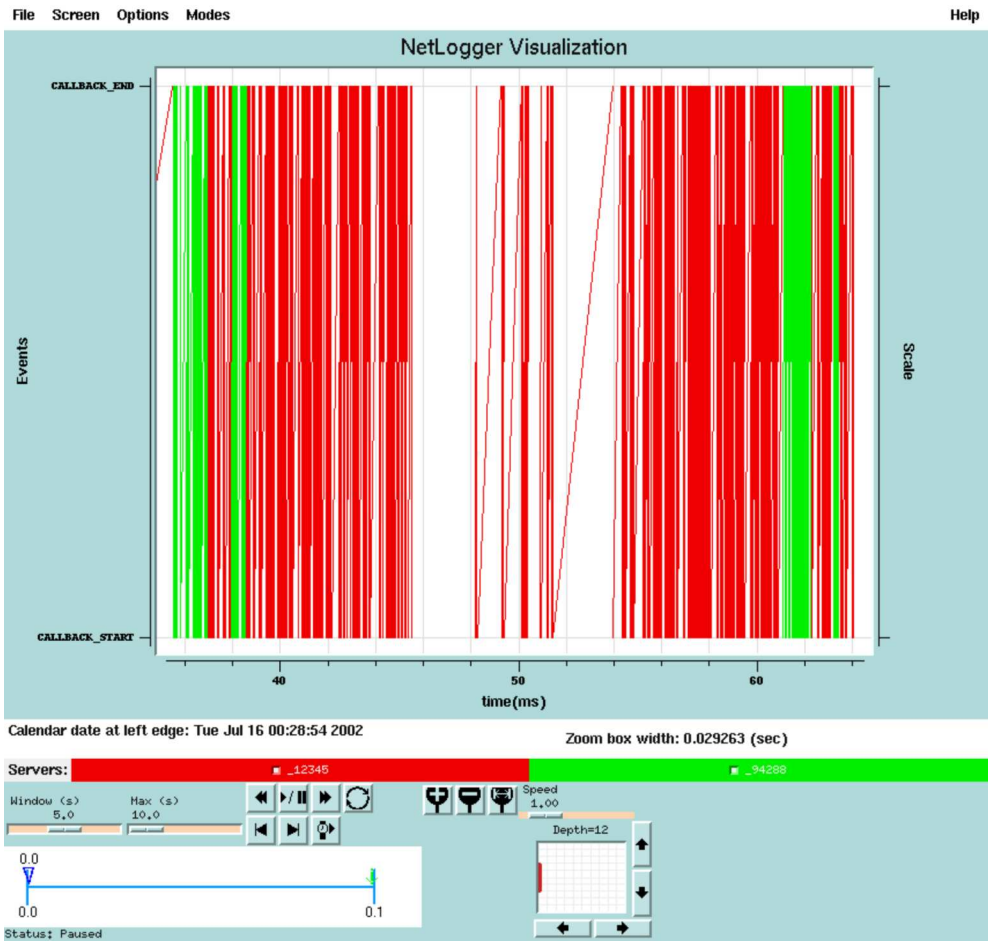

Figure 2.9: The `nlv` user interface

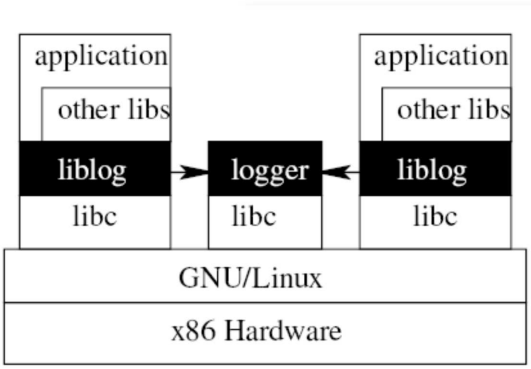

Figure 2.10: logging with `liblog` (taken from [10])

2.2.7 Friday

Friday is a *replay debugging* tool for distributed C/C++ applications [9]. It consists of a C library (`libc`) call interceptor/logger – shared library called `liblog` [10, 11] - and a replay console called `Friday`. `Friday` uses the GNU Debugger (GDB) for controlling the replay of the processes and it also has a GDB-like user interface. It is being developed at the University of California, Berkeley. The aim of the creators of `Friday` is to develop a non-intrusive, off-line distributed debug and replay tool for deployed (and live) production applications. They are especially interested in finding bugs, that only appear when the components of the distributed application are cooperating. These bugs are called *distributed bugs*.

`liblog` is preloaded by the linker before the debugged application, so no modification of the existing binary (and access to the source code) is needed. The platform has to be Linux/x86 and only POSIX C/C++ applications are supported. As the library can only be loaded before the application starts, the authors of `Friday` suggest that the `liblog` library would be kept pre-loaded all the time. This introduces some overhead, but `liblog` is designed to keep it as low as possible. There are benefits from keeping the `liblog` library pre-loaded – for example, this is the way to catch rare and slowly-developing bugs.

Not all of the software participating in the distributed application will have to be monitored by `liblog`. As `liblog` logs the contents of all incoming messages for the processes under monitoring, replay of some subset of distributed application processes is possible without replaying all the other processes communicating with this subset. This makes it possible to debug large

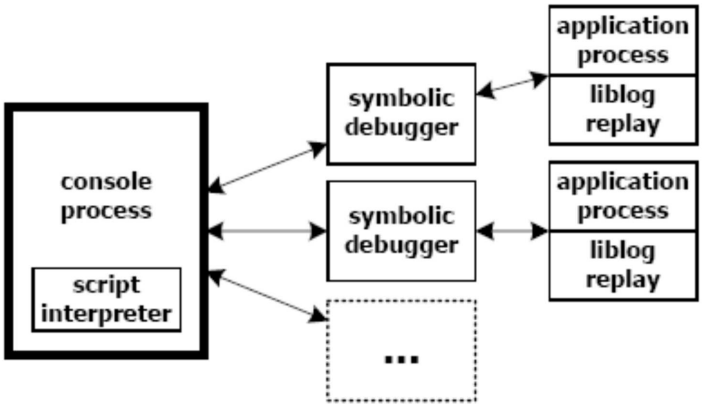

Figure 2.11: Overall architecture of `Friday` (taken from [9])

multi-vendor systems.

Because independent processes are replayed individually, some kind of mechanism is needed to coordinate between the replays and keep them globally consistent. `liblog` embeds Lamport clocks[15] in all outgoing messages for that reason.

Trace logs produced by `liblog` are gathered into central location by the `logger` process and then replayed (see Figure 2.10). Unlike most of the other distributed debuggers, the logs are replayed off-line and also *off-site* – not necessarily on the same machines, where the debugged processes ran. This enables the user to carry out thorough and intensive debugging sessions without disturbing the production system, where the logs were obtained from. One of the biggest costs of this approach is that the hardware and system software on the replay machine must be exactly the same as on the hosts where the logs were gathered from.

Replay in `Friday` is controlled by a symbolic debugger, currently GDB. Each process node is replayed by an independent debugger and `Friday` acts as a supervising process, coordinating symbolic debuggers to achieve global consistency (see Figure 2.11). It is possible to define global watch- and breakpoints and to attach user-defined scripts to those watch- and breakpoints.

At the moment, `liblog/Friday` is far from being a final product, but it already offers some interesting features not found in the existing debug and trace tools.

From all the tools described in this thesis, the philosophy behind `Friday`

has probably been the most influencing one to the trace synchronizing tool I am developing. However, the main difference between `Friday` and my application is that my application is a "black-box" trace log browser, while `Friday` can fully inspect the internal state of the replayed nodes (processes).

2.2.8 RAC Alert Consolidation

At the starting stage of TraceBrowser development, I searched for the tools with a similar purpose and user interface. One of the tools found was RAC Alert Consolidation (RAC) [18]. From all the tools I found, RAC had the goals closest the TraceBrowser's. RAC is a tool for correlating alert logs produced by Oracle Real Application Clusters [6]. From the little information provided on the RAC web-page, it has a scrolling interface similar to TraceBrowser's. Most of the features TraceBrowser already has (timestamp offsets, filtering), are filed under RAC's "To Be Implemented" list. Unfortunately, the development of the RAC appears to have stalled at an early beta stage.

2.2.9 Kompare

Kompare is a graphical difference viewer that allows you to visualize changes to a file [14]. It is mainly a tool for developers who want to compare different versions of source code. The user interface of Kompare shows two versions of the text file in parallel, visualizing the changes in a very distinct and intuitive way (See Figure 2.12). The text views of the versions are scrolled with a common scrollbar. While Kompare is not a debug tool, it gave me the inspiration for developing the scrolling engine for TraceBrowser.

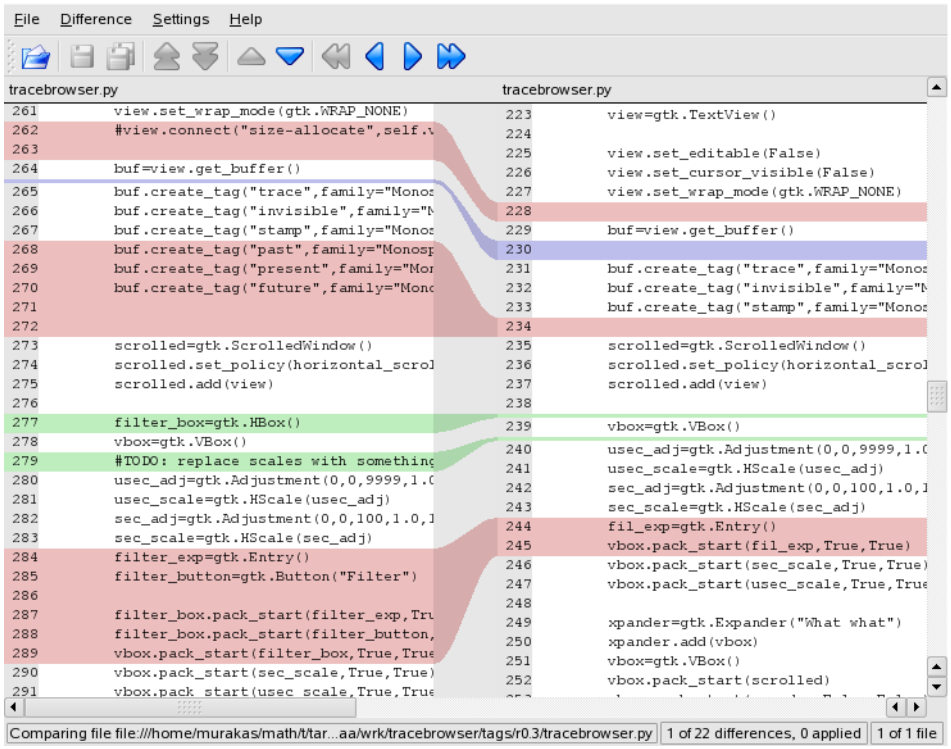

Figure 2.12: Comparing versions with Kompare .

2.2.10 More tools

Of course, there are more tools or projects that deal with various problems in the area of distributed debugging and execution replay. Here is a list of tools, concepts and libraries that were left out of my thesis, but are important enough to mention here:

Paradyn - a performance measurement tool for parallel systems [16],

Pablo - Performance Analysis Environment [19],

Jockey - a library for record-replay debugging [21],

Flashback - a lightweight OS extension for deterministic replay [24],

DejaVu - a replay debugger for Java [2],

Pip - a infrastructure for comparing expected and actual behavior in distributed systems [20].

I have reviewed the existing trace and debug tools for several reasons. Some of them, like `strace` and `tcpdump`, are a good base for my debug approach. They can be used as the tools for creating the trace and packet dump logs of the components of the distributed systems, to be later analyzed with my synchronization utility. Some of them, like `Friday` and `nlv`, can be used to learn new ways of debugging and analysis.

Chapter 3

Off-line Synchronization of Trace Logs

This chapter will describe an approach to debugging distributed and networked systems, called *Off-line Synchronization of Trace Logs*. Over the past few years I have combined existing tools, problem solving and data analysis methodologies from the field of debugging and other closely related fields. As a result, I have worked out a debugging methodology which can be used on both distributed and networked systems. It is also suitable for debugging in production environments. In the essence, Off-line Synchronization of Trace Logs is a simple methodology with low overhead, which uses existing trace tools and an off-line *event synchronizing* tool for debugging multiple processes possibly deployed into production environments. Events are synchronized by their timestamps, making it easier to determine error causality.

3.1 Case Studies: Debugging Deployed and Distributed Systems

During the years working as a system administrator, I have been responsible for solving problems in various kinds of networked and distributed application environments. In most of the cases, the applications involved were from different sources. Some of them were closed-source. For many of them, recompiling with debug support would have taken a lot of time and effort, and in almost all cases, they were running in production environments. In the case of trivial problems, these applications would give meaningful error

messages that were useful in the problem solving process. However, when the problems became more complex, especially when the bugs behind the errors were distributed bugs, the error messages thrown by applications became less meaningful. Sometimes there were no error messages at all. It was necessary to get an insight on the application execution internals and also on communication between different components of the system to solve the problem.

The descriptions of some of the cases follow:

Case 1: After the migration of a web server to new a version of the Apache `httpd` software at our site, some users started to complain, that when they clicked on links to PDF files, they got an error message stating that the file was corrupt. This only occurred with certain combinations of browsers, Adobe Acrobat Reader plug-ins and PDF files. As the files were in the correct format and corruption due to network problems was very unlikely, the real underlying problem had to be a bit more complex. The error message from the Acrobat Reader plug-in was the only error that any of the participating components would give. To get to the underlying cause of this problem, I set up two sessions: one with a client that worked correctly, and one with a client that gave the error message. Then I produced the trace logs of the test sessions (both server and client side) and recorded all of the network traffic. When correlating trace logs and packet dumps and comparing two sessions, I found out that the web server responded to the partial queries (the queries that request only some parts of the file) with an answer that had the correct data in it, but the header of the answer was invalid. Specifically, the header had an invalid `Offset` field. This was causing some of the browsers to overwrite a previously received PDF header with some other block from the partial answer received later. The reason why this error was emerging only on some cases was that the browsers did not always make partial queries for PDF files and some browsers seemed to ignore the offset field of the answer's header. An Apache bug report was posted and the problem got fixed.

Case 2: Anyone who has supported an installation of the Torque [13] batch processing system, probably knows that Torque is really sensitive on how the node hostnames resolve. Forward and reverse resolutions must exactly match each other and also the values on the configuration files. If the master node happens to have several network interfaces with different addresses and names, troubleshooting of the non-working Torque installation becomes a bothersome task. I have worked on a case where Torque was installed as a part of and configured by a cluster management software package. Because the configuration and the structure of our cluster was not entirely typical, the configuration part apparently failed and resulted in a non-working Torque

installation. Torque diagnostic tools reported that the cluster was working, but the submitted jobs were either left waiting in the queue forever or disappeared without any output.

By using combination trace and network traffic dump tools, I established exactly, which queries were made and which configuration files were consulted by Torque components. Then I customized the Torque configuration files and name resolution records according to the overview I got from the process of debugging.

In both of the described cases, the solutions were trivial to devise and implement once the cause of the problems was discovered. Finding the cause was the hard part. While the debugging tools provided me with valuable information, processing the large volumes of data to find this information was slow and error-prone. Special tools for supporting this kind of debugging approach would have made the process faster and probably more effective.

3.2 The Concept

The basis of the Off-line Synchronization of Trace Logs debugging concept is that it is actually possible to use most of the time-proven, flawlessly working single-application debug tools for debugging the networked and distributed systems, too. Execution traces and network messages can be logged independently for several processes in the distributed system and then combined and analyzed together. For this combining to be possible, timestamps have to be used on the trace records. The hardest parts are coordinating the whole process, and extracting the useful information out of the large volumes of collected results.

To have a faster and more effective analysis phase, I decided to switch from an almost-manual process to a semi-automated process. The tool for the automation of the analysis of the debug data is currently under active development and the current results are presented in the third chapter of this thesis.

There is another strong reason for me to develop my own concept: most of the distributed system's debugging frameworks or applications known to me are developer-centric (and this is perfectly understandable) – they are for people who are trying to improve their own code. A system administrator, unlike a developer, works with (installs, configures, integrates) applications created by other people. The focus in debugging is shifted from discovering internal bugs (e.g algorithmic bugs, bad coding) to finding the bugs that

occur at the boundaries of cooperating applications, at the boundaries of the application and the libraries, or at the boundary of the application and the operating system (inter-process communication, library calls, sys-calls).

My current distributed application tracing approach can be divided into four steps:

1. producing timestamped trace logs with tools of choice,
2. evoking the bug or waiting for it to occur,
3. collecting the logs,
4. synchronizing and correlating the logs to find any abnormal behavior.

As the first step, I identify and choose the processes I want to trace and also decide, what network traffic needs to be captured. Right now, setting up a trace session consists just of the manual preparation of the corresponding command lines. This process is a good candidate for automation, too. Then all of the trace processes are started. If the distributed application being debugged needs to be restarted or not, depends on the tools used. For most cases, I use tools that do not require a restart.

Actions on the second step depend on whether there is a known way to replicate the error or is it just necessary to wait for it to occur. Replicable bugs are, in general, easier to be dealt with. In addition, having a replicable bug means smaller volumes of trace data has to be analyzed. If the error in question has occurred, the trace processes can be detached and stopped after logging the data. If needed, and possible, the debugged application can also be restarted.

For the third step, all the logs have to be collected before the analysis can start. When possible, I already arrange the logs to be saved to the same network location at the first step. It must be kept in mind that writing logs to a network file system can slow down the tracer and the application being traced. When logging over the network is not possible, I usually use `scp` with public key authentication to collect log files.

The final and most significant part of the process is analyzing the logs. For the analysis, it is essential that the trace logs are synchronized (correlated) in some way. After the logs are synchronized, the person analyzing the logs can clearly see in which order the events occur, and hopefully understand, how the parts of the distributed application are connected and how the bug evolves and how the consequences of this propagate to other processes.

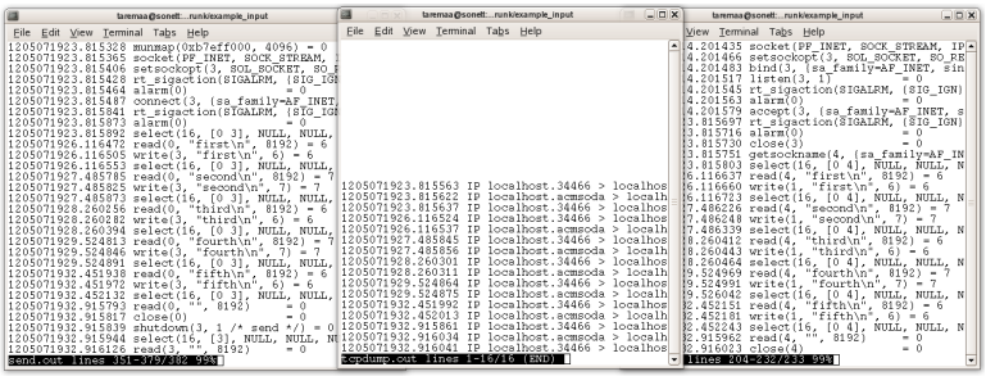

Figure 3.1: Using multiple terminal windows for manual trace log correlation

```
connect(3, {sa_family=AF_INET, sin_port=htons(9876), sin_addr=inet_addr("193.40.36.106")}, 16) = 0
arp who-has adeliae.mt.ut.ee tell tawaki.mt.ut.ee
rt_sigaction(SIGALRM, {SIG_IGN}, {SIG_IGN}, 8) = 0
arp reply adeliae.mt.ut.ee is-at 00:0c:f1:bf:d3:2f (oui Unknown)
alarm(0) = 0
select(16, [0 3], NULL, NULL, NULL) = 1 (in [0])
IP tawaki.mt.ut.ee.51247 > adeliae.mt.ut.ee.sd:  S 2429153632:2429153632(0) win 5840 <mss 1460,sackOK,timestamp 16021176
0,nop,wscale 2>
IP adeliae.mt.ut.ee.sd > tawaki.mt.ut.ee.51247:  S 3998576838:3998576838(0) ack 2429153633 win 5792 <mss
1460,sackOK,timestamp 310975826 16021176,nop,wscale 2>
write(3, "Hello World\n", 12) = 12
IP tawaki.mt.ut.ee.51247 > adeliae.mt.ut.ee.sd:  .  ack 1 win 1460 <nop,nop,timestamp 16021177 310975826>
IP tawaki.mt.ut.ee.51247 > adeliae.mt.ut.ee.sd:  P 1:13(12) ack 1 win 1460 <nop,nop,timestamp 16021177 310975826>
IP adeliae.mt.ut.ee.sd > tawaki.mt.ut.ee.51247:  .  ack 13 win 1448 <nop,nop,timestamp 310975826 16021177>
getsockname(4, {sa_family=AF_INET, sin_port=htons(9876), sin_ addr=inet_addr("193.40.36.106")}, [16]) = 0
IP tawaki.mt.ut.ee.51247 > adeliae.mt.ut.ee.sd:  F 13:13(0) ac k 1 win 1460 <nop,nop,timestamp 16021177 310975826>
read(4, "Hello World\n", 8192) = 12
```

Figure 3.2: Color-coding trace messages

As there are vast amounts of data and several parallel traces, manual synchronization (See Figure 3.1) will grow out of hands quickly. Browsing through two logs simultaneously just by hand is achievable without much effort, but starting with three or more parallel logs to step through, some better approach is required.

So far, I have handled the cases of three or four parallel traces with color-coding (e.g with ANSI codes [28]) the logs and then merging them to one file, sorted by timestamps, but this is not efficient enough. Only a small part of the total trace is visible on the screen, inter-process communication is harder to follow and it gets harder to follow the trace of one process, because it is now split up.

A notable weakness in this debug approach so far has been the lack of dynamic filtering. The `grep` utility and some AWK scripts are of great help, but they would have to be called before the process of browsing through the logs can start. If I wanted to change some filter, the whole synchronization and analysis process would start from the beginning and the current position

(in trace records) would be lost.

After some time of using this debug approach I did discover yet another problem that needed to be addressed. It is not uncommon to discover, that based on the timestamps, events appear to have occurred in illogical order – for example, in inter-process communication a read from a socket appears to be done before the corresponding write. There can be different reasons for this. First, the administrator to start the trace session could simply forget to synchronize the clocks before the trace logs are produced. Second, at the time when the logs are produced, there could be no intention of correlating them with the other logs from some different sources later on. Third, there are very small delays (usually only a few microseconds) between the moment that the event happens and the moment that the timestamp is acquired from the system clock. Sometimes, when events happen in very small time frames (for example, library calls) and the system is under heavy load, the delay between the actual event and its timestamp can result in anomalies in the correlation/synchronization process.

Using NTP for clock synchronization is not the final answer to this problem, because the problem is not that the operating system's clocks are not accurate enough, but the timestamping process itself is not always precise enough for timestamping massively occurring parallel events. It must be kept in mind that it is more important to recreate the exact order that the events happened than to have the exact time when each event happened.

For these reasons I did start developing my own helper application, called TraceBrowser. With the TraceBrowser tool I plan to cover the final step of my debug process - synchronizing and analysis.

TraceBrowser allows the user to load multiple parallel traces and then view and scroll them together (see Figure 3.3). Scrolling is done synchronously, based on timestamps rather than line numbers. Dynamic filters for both rows and columns can be applied to different log-files, rows can be color-coded. It is possible to define timestamp offsets for corrections in the event order. Also, automatic inter-process communication detection can be used to do this automatically. The TraceBrowser is described in more detail in the third chapter.

3.3 Advantages and Disadvantages

The Off-line Synchronization of Trace Logs approach (and largely, the Trace-Browser tool) is general enough to be compatible with all timestamped trace

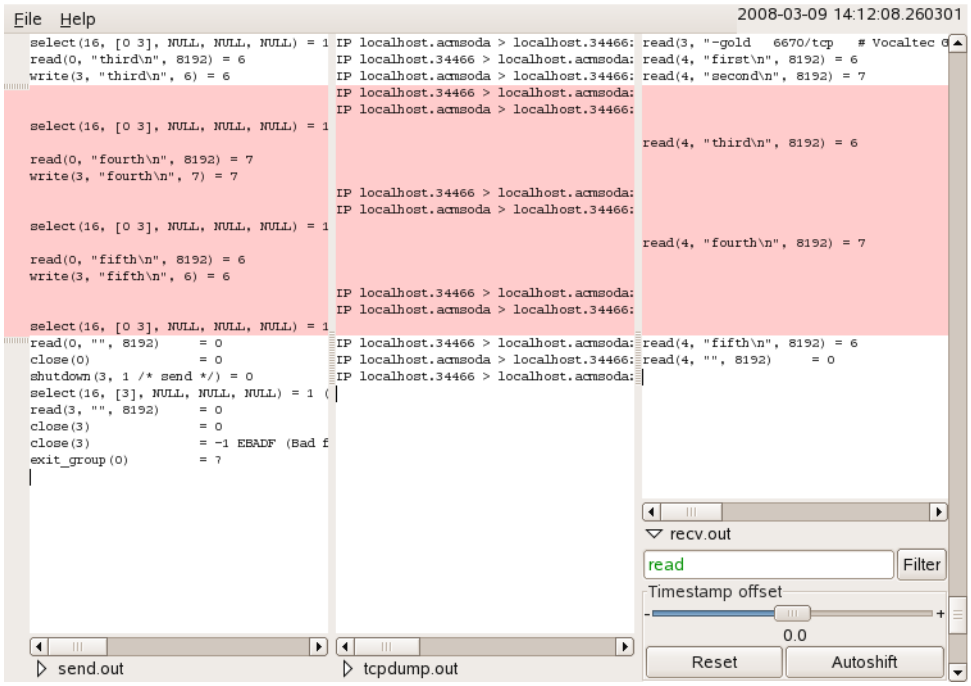

Figure 3.3: Using TraceBrowser for automatic trace log correlation

logs. That means you can use any tool to provide the traces or some other relevant debug information. The format of the timestamps has even not to be unified between tracers, because converting timestamps to a common format is quite trivial. Being independent from the tools used makes this approach platform independent.

When sticking to tools that can trace the execution of binaries, there is no need to recompile code or pre-load your own libraries before starting the application. However, recompiling the code to include some sort of debug support is not prohibited either. In more demanding situations, these different approaches to debugging and tracing can be combined to produce a better overview.

With the introduction of the TraceBrowser, the biggest shortcomings in this simple manual approach should be eliminated. Automatic synchronizing, synchronous scrolling, filtering and timestamp offsets are introduced to reduce the manual labor and offer a better overview of the application execution that was traced.

As the approach is quite simple, it has also a fair number of disadvantages.

When using this approach, synchronous and intrusive debugging are not possible. This is a setback for those who are debugging their own code, or

have the application's code at hand. For binary-only applications there is a somewhat lesser, but not nonexistent, need for being able to pause the execution and change the contents of the memory.

My approach and the TraceBrowser are suitable for off-line debugging only. It must be noted though, that off-line debugging is not a bad thing in itself, but the lack of on-line debugging can be. Because of this, making memory snapshots at critical moments is at least harder, if not impossible. One could make the memory snapshots at regular intervals and when debugging, use the snapshot closest in time to the event being watched. This solution has abnormal storage needs, and the memory snapshot being closest to the event is not necessarily accurate for the exact moment the event occurred. Variable watches can be set up in the same way, but with the same down-sides.

Problems of memory snapshots and variable watches could be overcome by using replay debugging, replaying the execution of whole processes as close to the original execution as possible. Unfortunately, the detail level of data collected with standard trace tools is usually not sufficient for accurate and consistent replay of distributed applications.

Chapter 4

The TraceBrowser

The TraceBrowser log synchronization tool supports the synchronization and analysis part of the Off-line Synchronization of Trace Logs debug methodology. TraceBrowser allows the user to step or scroll through several log files in parallel, while showing strict order of the log records in time without loosing a clear distinction between different log files. One of the goals of the TraceBrowser user interface is to enable the user to easily switch between correlating events from different logs and observing one particular log in a closer detail. This chapter will give an overview of the functionality of TraceBrowser, a detailed description of its structure and a general evaluation of its performance. Future development goals are also laid out.

4.1 Functionality of TraceBrowser

The core of the TraceBrowser is built around the synchronous scrolling function. At first, the user must load log files; this can be done from graphical user interface or by providing filenames as arguments on the command line. When the logs are loaded, they will be scrolled to the first event. The logs have a common scrollbar for scrolling them and log records are scrolled together over moments in time. Scrolling is implemented by a component called Past-Present-Future Scrolling Engine (PPF scrolling engine, see Figure 4.1). The PPF engine splits the log views vertically into three areas: topmost is called *past*, center is called *present* and the bottom area is called *future*. When the logs are scrolled to a certain moment in time, the *past* area holds all the log events that have happened (were stored) before this moment, *future* holds the log events that did happen after this moment, and *present* area holds the event that did happen at the exact moment plus (configurable) number of

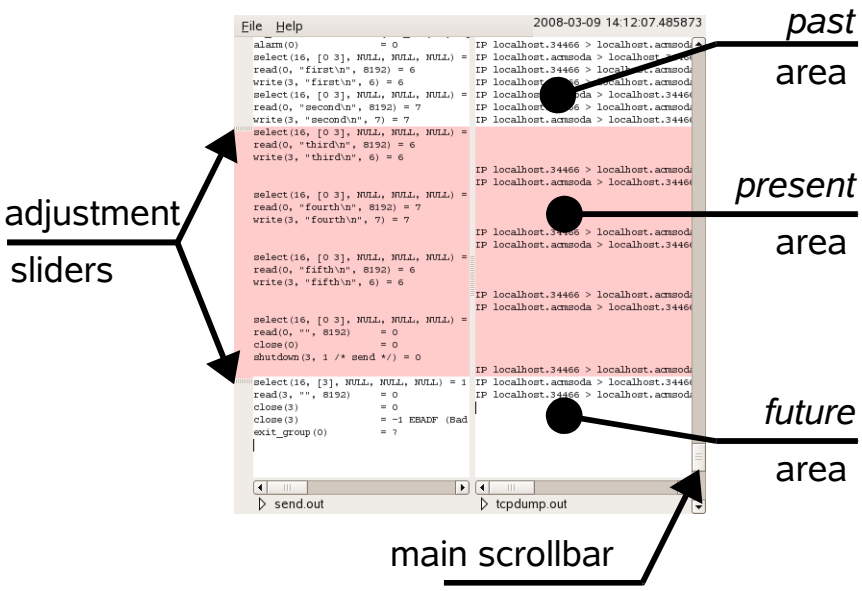

Figure 4.1: the interface of PPF scrolling engine

events that did happen right after this moment. The records in the *present* area are in strict temporal order. For example, if the record for a *write()* event is placed higher on *present* area than the record for a *read()* event, then this record for the *write()* event has earlier timestamp than the one for the *read()* event, irrespective to what logs do these events belong to. Records on *past* and *future* areas are locally time-wise ordered, meaning that when focusing on just one log, the events happened in the same order that they are displayed on these areas, but this may not be the case when viewing two or more logs in parallel. In the user interface, PPF areas are distinguishable by color; the background color of the *present* area differs from the background colors of the *past* and *future* areas.

The advantage of the PPF scrolling engine over just having all the records in a strict temporal order is a significantly more effective usage of the screen space. The PPF scrolling engine also plays an important role in enabling the user to frequently and seamlessly switch between the tasks of following one particular log and following several logs (and their interactions) in parallel. However, if the need for all the records to be in the strict temporal order should raise, the user can just adjust the height of the *present* area to be the height of the whole window. The height of the *present* area of the PPF scrolling engine is adjustable both through the `ppf_height` configuration

```
syntax = {'strace': re.compile('''^
                                (?P<sec_ep>\d+)\.
                                (?P<usec_ep>\d+)\s+
                                (?P<ev>.*)
                                $'''
                                ,re.VERBOSE),
          'apacheen': re.compile('''^
                                  (?P<ip>[^\[]+)\[
                                  (?P<day>\d{2})/
                                  (?P<month_name>\w{3})/
                                  (?P<year>\d{4}):
                                  (?P<hour>\d{2}):
                                  (?P<minute>\d{2}):
                                  (?P<second>\d{2})\s+
                                  [^\]]*\]
                                  (?P<rec>.*)
                                  $'''
                                  ,re.VERBOSE
                                  )
          }
```

Figure 4.2: defining log file syntax with regular expressions

parameter and the sliders in the user interface. The PPF scrolling engine is the result of purely practical work – the need for it became clear when testing the first versions of TraceBrowser and developing it was mostly a case of try, evaluate, and redesign.

As the core components of the TraceBrowser deal with temporal ordering of events from different logs, it is essential for every one of the log records to have some kind of timestamp. This is one of the three properties that the input log files must have. The other two are that the records must be in the temporal order within one log file and that one input line carries one log record. Everything else can be specified in the configuration file by the user. For every new type of log file, the user must describe its syntax in the form of a regular expression. This syntax expression maps each log record to timestamp and actual event data. Of course, different log files in the same TraceBrowser session can have different syntaxes. It is not (yet) possible to have records with different syntax within the same input file. New syntaxes can be described in the configuration file as the members of a Python dictionary. Perl-style regular expressions are used. See Figure 4.2 for a TraceBrowser syntax dictionary with two members: syntaxes for strace output format and Apache common log format.

If in the early versions of the TraceBrowser, a user opened a new input file, the syntax of this file had also to be chosen. The indexer needs to know the syntax for extracting the timestamps from the log records. However, because the syntaxes are described as regular expressions, it is possible to auto-detect

the log file syntax by simply doing a series of test-matches when a new log is opened. The first line of each of the input files will be matched against all the syntax expressions defined in the configuration file. When a match is found, the syntax attribute of the trace log is changed accordingly. This syntax auto-detection is available in the current version of the TraceBrowser.

Debugging can be a time-consuming process. This applies also to TraceBrowser sessions. The time must be used efficiently so that more effort goes into the actual analysis rather than opening files and tweaking configuration options. In the case of long and complicated trace browsing sessions, it is essential to be able to save and later on, restore, your work in progress. TraceBrowser allows the user to save *trace sessions*. One trace session consists of the set of the currently opened trace logs and all the options applied to them: filters, timestamp offsets, syntax, also global options like height for the PPF *present* area. It is important to understand, that the *Save Session* function does not save log files currently opened, only the filenames and options applied to them are saved. If, for example, one of the trace log files in a saved session should change, be replaced or deleted, the TraceBrowser would not load the version of the file that it had at the moment when the session was saved. Instead, a new version of this log file would be displayed. If this log file has been deleted or moved, the user would get an error message.

A prerequisite of getting a correct temporal ordering of the events is the synchronization of the clocks used to timestamp the log records. TraceBrowser expects that the computers used to acquire the trace logs use some means of clock synchronization, for example, NTP[17]. However, from my practical work with the Off-line Synchronization of Trace Logs methodology I have learned that from time to time you encounter log data that has timestamps of different logs out of sync. While intrusive debug technologies have several mechanisms to fight this problem – checkpoints and clocks embedded in the inter-process messages – these mechanisms cannot be used when using non-intrusive off-line methodologies. The only obvious solution is to try and correct the timestamps afterwards.

TraceBrowser offers two functions for correcting the timestamps – manual shifting and inter-process-communication (IPC) detection. Manual shifting is suitable if the user is able to detect the anomaly (events in the incorrect temporal order) and has a clear understanding in what order the events in question should be. Manual shifting is implemented by *+- sliders* (see Figure 4.3), one slider per every log view. The user will just have to move and hold the slider to the + side, if it is necessary to increase the timestamps of the log records, and the log will start scrolling. If the correct ordering is reached, the slider has to be released. Decreasing timestamps works in the same way.

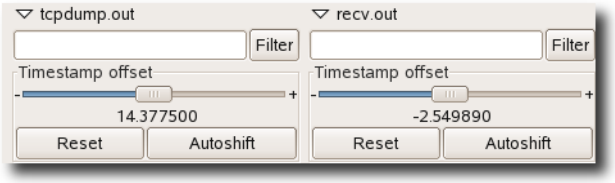

Figure 4.3: sliders for manual timestamp shifting

The other way to correct the timestamps is by using the automatic IPC detection rules. In the TraceBrowser, this function is called *Autoshift* and can be activated by clicking the Autoshift button in one of the log areas. The reason behind analyzing different logs together is that the processes that we are tracing influence each others behavior (or at least we suspect that this is happening). Inter-process communication is the most obvious way for processes to influence each others behavior. In the case where the timestamps are out of sync, the user will probably notice anomalies in the event order, but if this goes unnoticed, a wrong conclusion could be drawn. To fight this, the TraceBrowser tries to correlate events of sending of the messages (from one log) to the events of the receiving side. If the message appears to be received before it is sent, then the timestamp offset on the receiving side will be corrected, so that the sending event will have timestamp that is earlier than the timestamp of the receiving event. The IPC event correlation is based on a set of rules. Currently, there are two types of IPC detection rules. First type of rules select the potential send events from the log; second type of rules try to correlate these potential send events with other logs to find corresponding receive events. The rules for IPC detection are closely tied to the syntax of the trace log that they work on. The ruleset is defined in the configuration file by using regular expressions. The user can add own rules to the IPC detection ruleset. Sample rules can be seen on a Figure 4.4.

```
ipc_rules = [   (       re.compile('''
                                connect\((?P<valid>\d+)
                                ''',re.VERBOSE),
                        re.compile('''
                                close\((?P<invalid>\d+)
                                ''',re.VERBOSE),
                0),
                (       re.compile('''
                                getsockname\((?P<valid>\d+)
                                ''',re.VERBOSE),
                        re.compile('''
                                close\((?P<invalid>\d+)
                                ''',re.VERBOSE),
                0),
                (       re.compile('''
                                write\((?P<v_des>\d+),
                                   \s*\"
                                   (?P<data>[^"]*)\",
                                   \s*\d+\)\s*=\s*
                                   (?P<size>\d+)
                                ''',re.VERBOSE),
                        re.compile('''
                                read\(\d+,\s*\"
                                   (?P<data>[^"]*)
                                   \",\s*\d+\)\s*
                                   =\s*(?P<size>\d+)
                                ''',re.VERBOSE),
                1)
                ]
```

Figure 4.4: defining the IPC correlation rules with regular expressions

Although the PPF scrolling model makes maintaining the overview of combination of several trace logs a great deal easier, it alone is not always sufficient. Trace logs have usually high volumes of records, and it makes getting the needed information out of the large data-set a time-consuming and error-prone process. To fight these negative effects of having high volumes of data to present to the user, the majority of applications offer some kind of filtering functionality. TraceBrowser is no different, it allows the user to enter regular expression filters for the logs being viewed. When filters are activated by entering the filter expression and clicking the Filter button, the user interface acts as if the log records matching the filter are the only existing records. At the moment I do not feel the need for some kind of simplified filtering interface to complement regular expression filtering. The target group for TraceBrowser are IT professionals like system administrators or software developers and testers. These groups should already have a sound experience of using regular expressions.

4.2 Structure, Algorithms and the User Interface

4.2.1 Platform choice

TraceBrowser is written using the Python programming language. I chose Python because it is well suited for prototyping and trying out new designs, without the downside of having to switch the platform when the application moves to a more mature stage. Python comes with broad range of standard libraries providing a lot of supporting functionality, such as file I/O, regular expressions, list and string processing. This enables the programmer to focus on core parts of the program and use the libraries for needed support functions. Also, Python runs on most of the popular platforms today.

The graphical user interface of the TraceBrowser is built using the GTK+ library. The GTK+ library is available for Python through the PyGTK wrapper library. The reason behind this choice was that the GTK+ library supplies the user with assortment of advanced widgets while remaining relatively easy to use. I needed some "above-basic" widgets and signal handler functionality for writing the TraceBrowser. PyGTK enabled me to write my application without the need to write my own widgets.

4.2.2 TraceBrowser Core Classes

TraceBrowser has four core classes: TBGui, TBView, TraceSession and TraceLog (See Figure 4.5). A TBGui is the class that implements the TraceBrowser graphical user interface. It also implements one of the central elements of the TraceBrowser - the PPF scrolling engine. Scrolling is essentially a GUI function, so TBGui is a natural place for its implementation.

The TraceSession class represents the parts of the trace session that are not GUI related - it holds the references for all of the opened trace logs and a global index for the PPF scrolling engine, but unlike the trace session concept presented to the user, it does not hold the information about the GUI settings. The TraceSession class is closely tied to the TBGui class. Most of the code taking care of their interaction is held at the TBGui side.

The TraceLog class represents one of the actual trace logs being currently analyzed. It takes care of the actual file I/O (through the TraceFile class), holds and processes indexes for the trace log. Other attributes of the trace

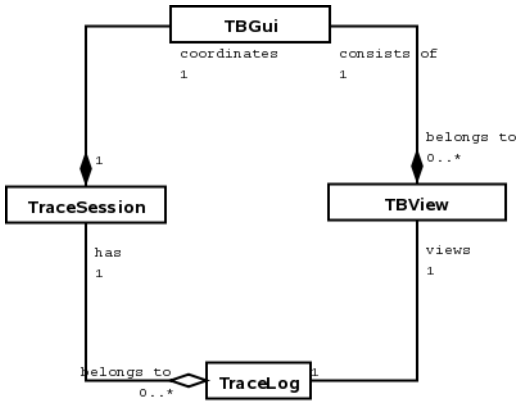

Figure 4.5: TraceBrowser class diagram

log, like time offset, filter expression and the syntax of the log are also kept here.

The TraceView class is the representation of the TraceLog in the graphical user interface. It holds the actual text views with log data and provides the user with controls to change TraceLog attributes.

4.2.3 Graphical User Interface and Log Processing Engines

The graphical user interface of TraceBrowser is designed to use the available screen space efficiently. Log views are the first priority when allocating the space. Views have to hold large amounts of log records and the more of the records are visible at the screen, the better. The controls to change trace view attributes are organized as an expander widget, so they can be hidden if not needed. All the future functions to be added to the TraceBrowser are probably going to be accessible from a context menu or expander widget. A toolbar with most-used functions is also under consideration.

To fully understand how the TraceBrowser works, an overview of the concept of *moments* and the way TraceBrowser uses indices to do its work is required. TraceBrowser looks at one input log file as an ordered sequence of events in time. The actual timestamps in this sequence may shift, if the user uses the time shifting function, but the number and ordering of the events remain constant at all times. If a trace log file is being opened in the user interface, two indices are created for this log. First, an index of the event records with the original timestamps, called *real index*, is created. For each

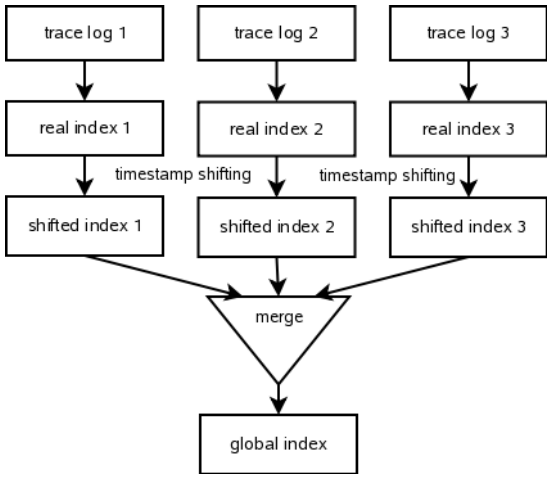

Figure 4.6: indices for the PPF scrolling engine

log record, the real index stores its timestamp and position in the user interface, implemented using the GTK+ TextMark widgets. Based on the real index, a similar index, called *shifted index*, is created. Shifted index has been corrected with the timestamp offset. When the shifted index is made invalid by changing the timestamp offset, it is refreshed. Furthermore, both real and shifted indices hold the data of the visible (not filtered) records only. When a filter is applied to log, both of its indices will be refreshed.

A Trace session can consist of a number of trace logs. Again, TraceBrowser looks at one trace session as an ordered sequence, in this case holding *moments*, rather than events. In one given moment, one or more events can happen. To handle this sequence, a third type of index, called the *global index*, is created. To create the global index, shifted indices of all the trace logs are merged, combining one or more events with exactly equal timestamps as one moment (See Figure 4.6). For every moment, the global index stores its timestamp, a list of the trace logs that have records with this timestamp, and list of user interface positions (TextMark widgets) for these records.

The PPF scrolling engine makes heavy use of the global index. The main scrollbar is in fact scrolling over the moments in the global index, not over the lines or columns of text like an ordinary scrollbar/scrolling engine would. The PPF engine is set to handle scrolling signals from the main scrollbar. When one of the scrolling events occurs, it is passed the desired moment (as an index position) as the input data. The scrolling engine reads needed user interface positions (TextMarks) for each of the log views from the global index, scrolls the views to these positions and then adjusts them according to the height of the *present* area.

Two of the most compute-intensive components of the TraceBrowser are the filtering engine and the Autoshift IPC detection engine. The filtering engine has to process all of the records in the log that it is working on. In the current version of the TraceBrowser, filtering engine does a full reconstruction of one log view. It matches the log data line by line, and creates new TextMark objects for the records that match the regular expression. The filtering engine also creates new indices for the trace log. When the log is processed, filtering engine refreshes the global index.

The IPC detection engine has to first process a log that it is activated on. It compiles a list of the potential sending events, then processes all the remaining logs. For every remaining log, it tries to correlate each line to all of the potential sending events. If a match is found, the corresponding sending event is excluded from the list and timestamps of the matching sending and receiving event are compared. In the case of an anomaly, the timestamps of the receiving are corrected.

4.3 Memory Usage and Performance Evaluation

To evaluate the TraceBrowser performance, I tested three key aspects – the areas which will supposedly have the biggest impact on the TraceBrowser overall performance. For each component, I did ten tests with ten different-sized input file sets. The results are presented here in the form of graphs.

The first key aspect of the TraceBrowser is the memory usage: how does the overall size of the input files influence the size of the TraceBrowser virtual memory size? As the current version of the TraceBrowser does not use buffering of any sort, the best case scenario is a linear relation with the coefficient of 1. In a real life scenario, this coefficent is unlikely. For a better overview, I estimated the amount of memory taken by the Python interpeter and created an alternative graph that shows only the memory used by the TraceBrowser data structures.

Memory performance tests (See Figure 4.7) show that the relation between the input log size and TraceBrowser virtual memory size is linear, like expected. However, the coefficent is approximately 7. For the tool in an early development stage, this is acceptable, but to achieve a mature product status, this coefficent would need to be lower – a number between 2 and 3 would be acceptable.

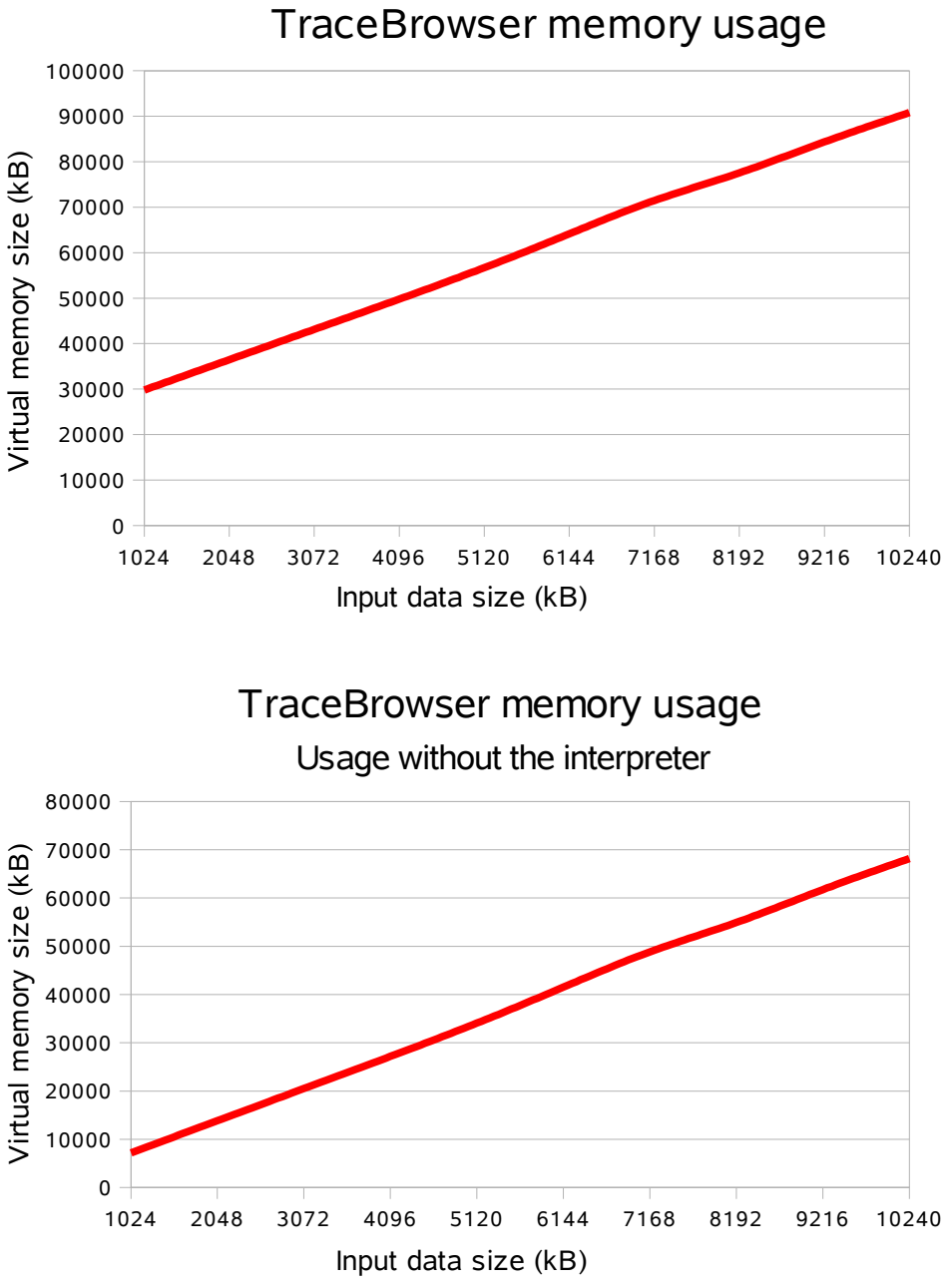

Figure 4.7: the TraceBrowser memory usage

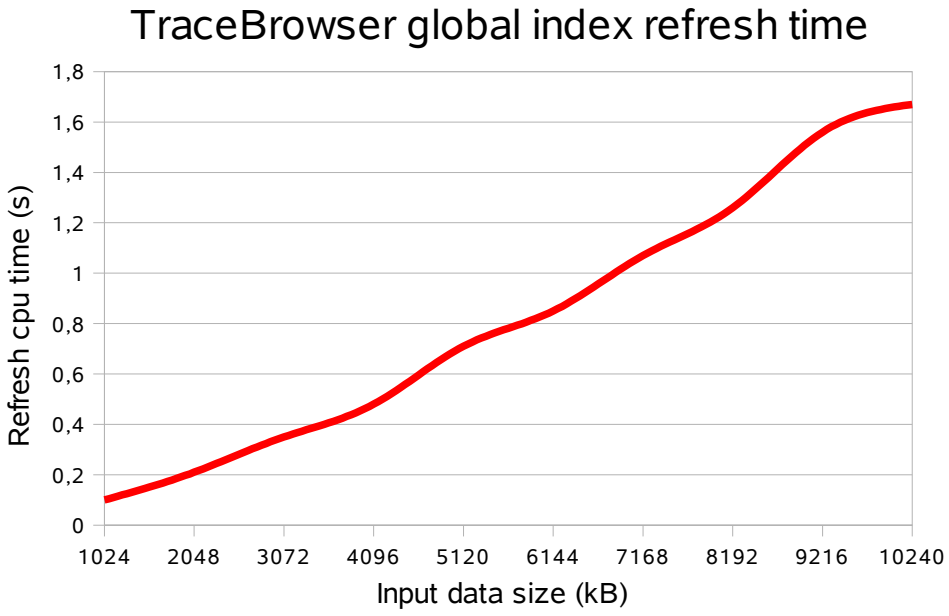

Figure 4.8: Index refresh performance

The core of the TraceBrowser, the PPF scrolling engine, uses indices to do its work. If one of the log indices becomes invalid, it has to be recalculated and a merge operation to create a new global index will have to be performed. I tested how the size of the input data influences the time it takes to refresh the global index (see Figure 4.8).

On the positive side of the results, the relation between input data size and the global index refresh time is linear, like it should be. However, my real user experience with the TraceBrowser shows that refresh times over 0.5 seconds will clearly degrade the user experience when using the manual timestamp shifting sliders.

The PPF scrolling engine event handler function is the function most frequently called in the TraceBrowser. Because it directly affects how fast the user interface reacts to using the scrollbar, it has to be a quick function and ideally should not be influenced by the input data size. I tested if this is true in the TraceBrowser (see Figure 4.9).

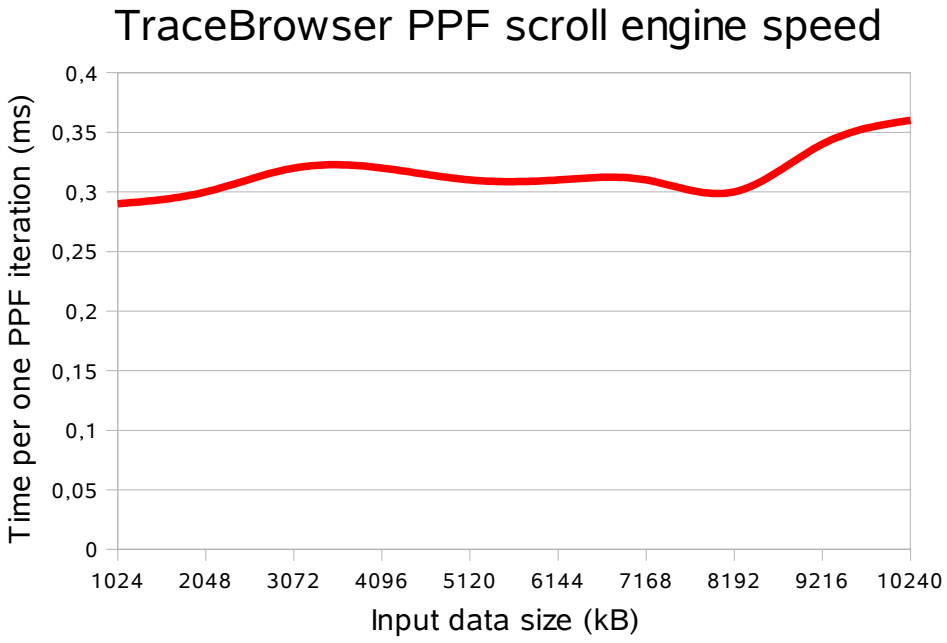

Figure 4.9: PPF scrolling engine: event handler performance

Although the results for this test show a slight relation between the size of the input data and the amount of time it takes to execute one iteration of the PPF scrolling engine, the real change in scrolling engine execution speed is marginal. The difference of 50-60 microseconds is nowhere near to be noticed by the user.

When analyzing all the performance tests together, a clear trend emerges – most of the components sacrifice their performance (by having to do more preparation tasks) to have a very effective scrolling engine. Now there is a need to optimize these components, while loosing as little as possible in the scrolling engine performance.

From the general usability and functionality standpoint, the TraceBrowser has already proven itself to be an useful tool. I have tested it on the trace logs similar to the ones I had produced for the cases described at the beginning of the third chapter. TraceBrowser practically frees the user of the burden of keeping the trace logs in the correct temporal order. At the beginning stages of the TraceBrowser development, I declared that this tool would be best suited for the cases where there are three or more parallel logs to follow, and that for the cases with two parallel logs, the manual approach was already sufficient. However, it has turned out that the TraceBrowser simplifies the

synchronization of two parallel logs beyond my expectations, making it very easy and fast to process. By using the TraceBrowser, I can really focus on the actual analysis rather than keeping the track of the timestamps.

4.4 Future Development

The current version of the TraceBrowser loads all the logs into the memory. Adding to this memory usage are the indices for the scrolling engine. When trying to find a replicable bug, the trace logs are probably small enough in size to fit into the memory of an average workstation. However, to find non-replicable bugs, long trace sessions are needed, possibly producing trace logs that wont fit into the memory. Sure enough, these logs can be pre-filtered or splitted before using the TraceBrowser on them, but that would mean the trace analyzing session would be more difficult and worse, less efficient. For that reason, one of the near-future development goals for the TraceBrowser is the addition of a buffering function. Portions of the log data large enough to fit into the memory would be loaded and the rest would be read from the filesystem when the need arises. This buffer can be implemented as a part of the PPF scrolling engine. The biggest challenge is to redesign various indices that the scrolling engine uses. At the moment, all the indices together use amounts of memory roughly comparable to the size of the log data.

`strace` and `ltrace` can produce logs which contain traces for all the sub-processes or threads in one application. As the system and library calls for these sub-processes share the same trace log file, the current version of the TraceBrowser would load this trace into one trace view. Usually, it is desirable to have separate sub-processes and threads in separate trace views. For now, I pre-split the log to separate files before loading them into the TraceBrowser. Because this is relatively simple functionality, it is highly probable, that I decide to include this additional step in a future release of the TraceBrowser.

If the screen space becomes a valuable resource, all means of using it more effectively must be considered. One widely-used way to present more data with the same amount of text is by using different colors. In the context of the trace logs, one way of coloring the trace logs is syntax highlighting. For example, different colors could be used for the system call name, argument and the result. There are other ways of using colors on the trace logs: colors could reflect the system call's result (success, error) or the time that system call takes (with gradients). This possible addition to TraceBrowser is not critical, but could enhance the user experience significantly.

One of the future challenges of the TraceBrowser development will be enabling the event records to be multi-line. At the moment, all the logic and data structures inside the TraceBrowser relies upon the assertion that all the input logs have exactly one log record per line. Conveniently for the TraceBrowser, many log formats are compatible with this. The number of log formats using multi-line event records seems to be on the rise and it would not be a good idea to ignore it in the TraceBrowser development process. To support the case where all the logs being analyzed have multi-line events with the same fixed number of lines, the TraceBrowser would need only some minor changes. However, to support a mixed set of different-height events, most of the log processing engines (PPF, filtering, IPC detection) would need a major re-write. This will not be an easy task to complete.

Timestamps of the events on different trace logs can be "out of sync" for different reasons. When the shifts in the timestamps are caused by non-synchronized clocks, setting a constant timestamp offset for all the records will correct the problem perfectly. When the shifts in the timestamps are caused by the way a timestamping system works, combined with high or changing system loads, constant offsets might not be the solution. There should be an option to distribute event records into groups and assign different timestamp offsets to these groups. The TraceBrowser does not yet support this. Internally, this kind of function would be quite easy to implement, but the problem lies in the user interface design – how to present this functionality to the user in a intuitive and compact way?

TraceBrowser supports many different log formats because it does not demand much more from the log file than just having single-line timestamped records. Remaining event data for the log record does not go through any extra processing and the TraceBrowser does not have any knowledge about the semantics of the records. This means the user must interpret the information without any assistance. It would be better if the analysis software had more internal knowledge about the data, so it can assist the user in tasks like interpretation, filtering. One way to do it without loosing the support for a wide range of formats would be a plug-in interface. For the logs formats we want the program to know more about, we use plug-ins, while other log formats are supported too. For plug-ins, already existing software could be used. For example, the `tcpdump` tool allows the user to save all captured traffic to a file in raw format. The user could open the raw `tcpdump` capture file in the TraceBrowser, the TraceBrowser would then start up `tcpdump` for interpreting and filtering the raw input file, providing a wider range of possibilities than just having a static output of the `tcpdump` tool stored in a trace log file.

Besides implementing new functionality, a successful software development project must also focus on fixing the bugs and improving the performance. TraceBrowser is a fast-developing application and therefore it probably has a number of bugs, performance and stability issues in addition to those described in the performance evaluation section. TraceBrowser has already shown that it has the potential to be a useful tool, now it is the time to realize that potential by improving its performance and scalability. This is not possible without a sound user-base and development group, so there is need to announce TraceBrowser to interested groups.

Chapter 5

Conclusion

While debugging in distributed and deployed systems is inarguably a harder task than debugging a single application, tools and approaches designed for these systems are constantly gaining new grounds.

The Off-line Synchronization of Trace Logs debug methodology shows how the combination of the existing debug tools, when used in a simple framework and complemented with a new analysis tool, can turn out to be effective in debugging distributed and deployed systems. Re-using existing concepts and tools makes this methodology less development-intensive.

On the downside, the simplicity of the Off-line Synchronization of Trace Logs debug approach is one of its disadvantages – it lacks many debug features, like synchronous and symbolic debugging or variable watches. This may mean that when the problem is deep enough, the Off-line Synchronization of Trace Logs is not the right approach. However, when the bugs appear on the application-application, application-kernel or application-library boundaries, this methodology turns out to be pretty effective. In the deployed systems, more complex debug tools cannot be used anyway, so this approach is one way to debug in the production environments.

The TraceBrowser tool presented in this thesis, regardless of it still being in an early development stage, can already be used to illustrate and facilitate the Off-line Synchronization of Trace Logs methodology.

I have laid down some future goals with the development roadmap for the TraceBrowser tool, but ultimately, the future of the TraceBrowser depends on the user acceptance. The more interested users the TraceBrowser will have, the more it makes sense to allocate resources to its development. When the project leaves the early stages, it makes sense to try to attract more developers to it. The tool is actively developed and available from [25]. For the cases similar to the ones described in the third chapter, the TraceBrowser has already proven to be a helpful tool.

Silumislogide sünkroniseerimine

Magistritöö (20AP)
Marti Taremaa

Resümee

Silumine hajussüsteemides on keeruline töö. Selleks otstarbeks on praeguseks välja töötatud mitmesuguseid lähenemisi ja tööriistu, mis püüdlevad erinevate eesmärkide poole ning sobivad rakendamiseks paljudel hajusrakendustel. Siiski on küllalt ruumi nii täiesti uute metodoloogiate loomiseks kui ka olemasolevate uutele aladele laiendamiseks.

Selles magistritöös antakse esmalt ülevaade hajusüsteemide silumisel esinevatest probleemidest. Probleemikirjeldused on illustreeritud näidetega igapäevatööst ning samuti tutvustatakse hetkel eksisteerivaid töövahendeid hajussüsteemide silumiseks. Töö põhiosaks on süsteemihalduritele sobiva silumismetodoloogia "*Off-line Synchronization of Trace Logs*" – "Silumislogide sünkroniseerimine" – ning autori poolt arendatud seda lähenemist toetava tööriista TraceBrowser tutvustamine.

"*Off-line Synchronization of Trace Logs*" seisneb hajusrakenduse erinevatest komponentidest kogutud *trace*-logide sünkroniseerimises ja analüüsis. Logide genereerimiseks kasutatakse võimalikult suures osas juba eksisteerivaid silumistööriistu. Loodud logid kogutakse kokku ning analüüsitakse paralleelselt. Mitme logifaili üheaegseks analüüsiks on vaja neis olevaid kirjeid ajaliselt järjestada. Kuna töö autori kogemuste põhjal teeb ajalise järjestuse jälgimine logide analüüsist väga aeganõudva töö, on vaja tööriista, mis selles etapis abistaks. Just sel põhjusel arendab autor rakendust nimega TraceBrowser.

Logide analüüsimiseks loodud TraceBrowser võimaldab mitme *trace*-logi üheaegset sünkroonset sirvimist, kusjuures sünkroniseerimine baseerub logikirjete ajalisel järjekorral. Ajalise järjestuse visuaalne kujutamine toimub ilma

logisid ühendamata. Nii on võimalik kiiresti lülituda logide paralleelselt analüüsilt ühe logi analüüsile ning vastupidi. TraceBrowser toetab kõiki logiformaate, kus ühele reale vastab üks kirje ning iga kirje sisaldab oma tekkimise aega. Peale ajalise sünkroniseerimise pakub TraceBrowser ka filtreerimise ja logide "ajas nihutamise" funktsioone. Hoolimata oma eksperimentaalse rakenduse staatusest on TraceBrowser juba tõestanud oma võimekust reaalsetest süsteemidest kogutud logide sünkroniseerimisel. Piisava kasutajate hulga tekkimisel on autoril plaanis selle töövahendi arendust jätkata.

Bibliography

[1] J. Abela and T. Debeaupuis. Universal format for logger messages : Internet draft. Available from World Wide Web: `http://tools.ietf.org/html/draft-abela-ulm-05`.

[2] Bowen Alpern and et al. Dejavu: Deterministic java replay debugger for jalape no java virtual machine. Available from World Wide Web: `http://citeseer.ist.psu.edu/424170.html`.

[3] Trolltech ASA. Qt - cross-platform rich client development. Available from World Wide Web: `http://trolltech.com/products/qt/features/index`.

[4] Juan Cespedes. ltrace project page. Available from World Wide Web: `http://ltrace.alioth.debian.org/`.

[5] Concurrent Computer Corporation. Nightstar linux debugging and analysis tools. Available from World Wide Web: `http://www.ccur.com/isddocs/NightStarTools.pdf`.

[6] Oracle Corporation. Oracle real application clusters 11g. Available from World Wide Web: `http://www.oracle.com/technology/products/database/clustering/index.html`.

[7] Gerald Combs et al. Wireshark - network protocol analyzer. Available from World Wide Web: `http://www.wireshark.org/`.

[8] Free Software Foundation. Debugging with gdb. Available from World Wide Web: `http://sourceware.org/gdb/current/onlinedocs/gdb_toc.html`.

[9] D. Geels, G. Altekar, P. Maniatis, T. Roscoe, and I. Stoica. Friday: Global comprehension for distributed replay. 2007. Available from World Wide Web: `http://www.cs.berkeley.edu/~galtekar/debugging/nsdi07.pdf`.

[10] D. Geels, G. Altekar, S. Shenker, and I. Stoica. Replay debugging for distributed applications. 2006. Available from World Wide Web: `http://www.cs.berkeley.edu/~galtekar/debugging/usenix06.pdf`.

[11] D. M. Geels. *Replay Debugging for Distributed Applications*. PhD thesis, 2006. Available from World Wide Web: `http://www.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-163.pdf`.

[12] D. Gunter, B. Tierney, K. Jackson, J. Lee, and M. Stoufer. Dynamic monitoring of high-performance distributed applications. Available from World Wide Web: `http://dsd.lbl.gov/publications/HPDC02-HP-monitoring.pdf`.

[13] Cluster Resources Inc. Torque resource manager. Available from World Wide Web: `http://www.clusterresources.com/pages/products/torque-resource-manager.php`.

[14] Joshua Keel and Jeff Snyder. Kompare. Available from World Wide Web: `http://www.caffeinated.me.uk/kompare/`.

[15] Leslie Lamport. Time, clocks and the ordering of events in a distributed system. *Communications of the ACM*, 21(7), 1978.

[16] Barton P. Miller, Mark D. Callaghan, Jonathan M. Cargille, Jeffrey K. Hollingsworth, R. Bruce Irvin, Karen L. Karavanic, Krishna Kunchithapadam, and Tia Newhall. The paradyn parallel performance measurement tool. Available from World Wide Web: `ftp://ftp.cs.wisc.edu/paradyn/papers/Miller95Paradyn.pdf`.

[17] D. Mills. Simple network time protocol (sntp), 1995.

[18] Michael Moeller. Rac alert consolidation. Available from World Wide Web: `http://www.miracleas.dk/index.asp?page=168&page2=159&page3=399`.

[19] D. Reed, R. Aydt, T. Madhyastha, R. Noe, K. Shields, and B. Schwartz. An overview of the pablo performance analysis environment, 1992. Available from World Wide Web: `citeseer.ist.psu.edu/reed92overview.html`.

[20] Patrick Reynolds, Charles Killian, Janet L. Wiener, Jeffrey C. Mogul, Mehul A. Shah, and Amin Vahdat. Pip: Detecting the unexpected in distributed systems. Available from World Wide Web: `http://issg.cs.duke.edu/pip/nsdi06preprint.pdf`.

[21] Yasushi Saito. Jockey: A user-space library for record-replay debugging. 2005. Available from World Wide Web: http://www.hpl.hp.com/techreports/2005/HPL-2005-46.pdf.

[22] Allinea Software. Allinea distributed debugging tool. Available from World Wide Web: http://www.allinea.com/?page=48.

[23] SourceForge. strace project page. Available from World Wide Web: http://sourceforge.net/projects/strace/.

[24] Sudarshan M. Srinivasan, Srikanth Kandula, Christopher R. Andrews, and Yuanyuan Zhou. Flashback: A lightweight extension for rollback and deterministic replay for software debugging. 2004. Available from World Wide Web: http://www.usenix.org/events/usenix04/tech/general/full_papers/srinivasan/srinivasan_html/paper.html.

[25] Marti Taremaa. Tracebrowser - google code. Available from World Wide Web: http://code.google.com/p/tracebrowser.

[26] tcpdump/libpcap team. tcpdump/libpcap public repository. Available from World Wide Web: http://www.tcpdump.org/.

[27] B. Tierney, W. Johnston, B. Crowley, G. Hoo, C. Brooks, and D. Gunter. The netlogger methodology for high performance distributed systems performance analysis. Available from World Wide Web: http://dsd.lbl.gov/publications/NetLogger.HPDC.paper.ieee.pdf.

[28] Wikipedia. Ansi escape code. Available from World Wide Web: http://en.wikipedia.org/wiki/ANSI_escape_code.

[29] Wikipedia. Debugging. Available from World Wide Web: http://en.wikipedia.org/wiki/Debugging.

Appendix A

Short user guide for the TraceBrowser (version 0.5)

This user guide collects all the information concerning the usage of the TraceBrowser from the thesis, complements this information with technical data and presents it in a compact form.

System requirements

TraceBrowser has been tested to work with:

- Python >= 2.4
- GTK+ >= 2.8
- pyGTK >= 2.8

Starting

Starting the TraceBrowser from the command line:

```
$ cd /directory/where/tracebrowser/was/extracted/to/
$ ./tracebrowser.py
```

or

```
$ ./tracebrowser.py logfile1 logfile2 ...  logfileN
```

TraceBrowser has no limits on the number of input files. However, when running on a typical computer display, TraceBrowser can accommodate 4-5 logs in parallel; more than that can get awkward. Due to the performance issues described in Chapter 4.3 it is recommended that the sum of the input file sizes remains below 10MB. While the files are loading, TraceBrowser will

display a "Please wait..." message on the upper right corner of the window. The syntax of the input files will be automatically detected.

Currently the configuration file defines two possible syntaxes:

- logs with records starting with a POSIX timestamp - (*<seconds_from_the_epoch.microseconds>*). This is the syntax of `strace` (command line option `-ttt`), `ltrace` (command line option `-ttt`), and `tcpdump` (command line option `-tt`) output.
- Apache `httpd` common log format

Adding new syntaxes is described briefly in the TraceBrowser configuration file (see also "The TraceBrowser configuration file" section).

Adding a new trace log to the session

New trace log files can be added to the session by choosing **File->Add Trace**. If none of the defined syntaxes match, the log will show up as empty.

Using the main scrollbar and PPF scrolling engine

After the logs have finished loading, they will be initially scrolled to the first event, and it is possible to scroll them using the main scrollbar (see Figure 1). The main scrollbar coordinates the past-present-future (PPF) scrolling engine. The PPF scrolling engine scrolls the log events over moments in time rather than over the lines of text. The PPF engine splits the log views vertically into three areas: topmost is called *past*, center is called *present* and the bottom area is called *future*. When the logs are scrolled to a certain moment in time, the *past* area holds all the log events that have happened (were stored) before this moment, *future* holds the log events that did happen after this moment, and *present* area holds the event that did happen at the exact moment plus a (configurable) number of events that did happen right after this moment. The records in the *present* area are in strict temporal order. For example, if the record for a *write()* event is placed higher in *present* area than the record for a *read()* event, this record for the *write()* event has an earlier timestamp than the one for the *read()* event, irrespective to what logs these events belong to. Records on *past* and *future* areas are locally time-wise ordered, meaning that when focusing on just one log, the events happened in the same order that they are displayed on these areas, but this may not

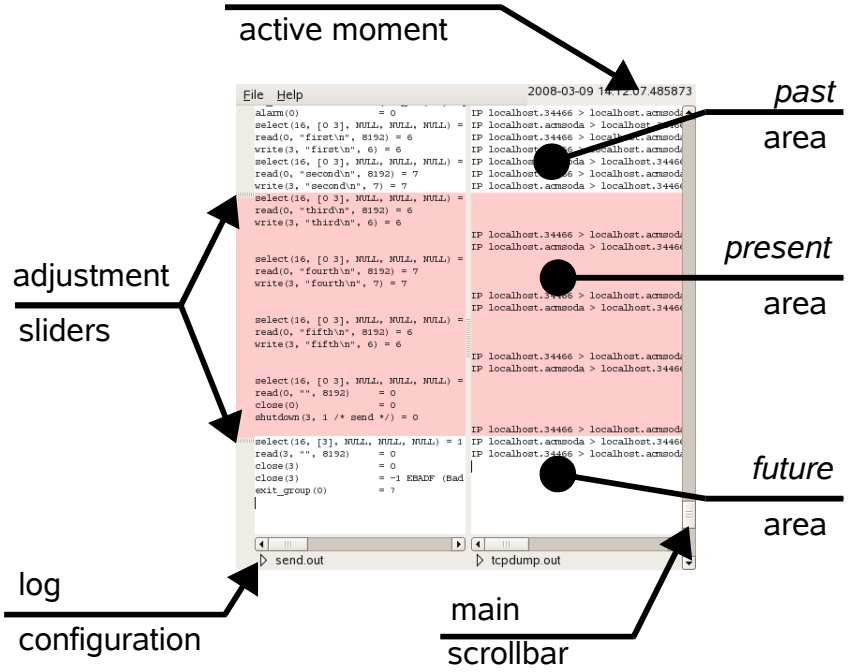

Figure 1: TraceBrowser user interface

be the case when viewing two or more logs in parallel. In the user interface, PPF areas are distinguishable by color; the background color of the *present* area differs from the background colors of the *past* and *future* areas. The color of a PPF area can be defined in the configuration file.

Adjusting the Past-Present-Future areas

For greater flexibility, the height and position of the present area can be changed by adjusting the sliders on the left side of the TraceBrowser window (see Figure 1).

Applying filters

To use filters and timestamp shifting functions, the user has to first expand the configuration area (see Figure 2) by clicking on a log's name or the triangle next to the log's name.

For every log, filters can be defined using Perl-style regular expressions. Filters are activated by clicking the "Filter" button or pressing the Enter key in

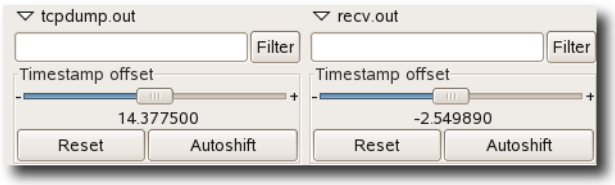

Figure 2: Log configuration area

the filter entry field. TraceBrowser gives the user instant feedback concerning the regular expression syntax. If the regular expression being entered is not a valid Perl-style regular expression, it is displayed in red. If the expression is valid, it turns into green.

Manual timestamp shifting

The timestamp offsets are set either manually or using the Autoshift function. For manual timestamp shifting, user has to move the +- slider. The user has to move and hold the slider to the + side, if it is necessary to increase the timestamps of the log records, and the log will start scrolling. If the correct ordering is reached, the slider has to be released. Decreasing timestamps works in the same way. The speed of the shift depends on how far away from the center the slider is.

Automatic timestamp shifting

The Autoshift function tries to detect possible timestamp anomalies by correlating the logs using the correlation rules defined in the configuration file. The default configuration includes some inter-process communication detection rules for the `strace` and `ltrace` logs. The user has to designate one log as a *master* log by clicking its Autoshift button. The log is then checked for system calls writing to a network socket. For all the matches, corresponding read functions are searched from the other logs. If a match is found, timestamps are compared to assure that the write function takes place before the read function. If neccessary, the timestamps are shifted.

Saving and loading the trace session

Currently active trace sessions can be saved to a file by choosing **File->Save Session**. One trace session consists of the set of the currently opened trace

logs and all the options applied to them: filters, timestamp offsets, syntax, also global options like height for the PPF *present* area. It is important to understand, that the *Save Session* function does not save log files currently opened, only the filenames and options applied to them are saved. When loading a saved session by choosing **File->Open Session**, and this saved session refers to files that are changed, the new versions would be displayed. If the log files in a saved session should have been deleted or moved, the user would get an error message when loading the session.

Using the examples

The TraceBrowser tarball includes a small number of examples to illustrate the usage of the TraceBrowser. Examples are provided as TraceBrowser trace session files and can be found in the `example_sessions` directory of the TraceBrowser tarball.

The TraceBrowser configuration file

The TraceBrowser configuration file is located in the same directory as the TraceBrowser executable. The file is called `tbconfig.py`. The configuration file allows the user to set various default values (window size, colors, ..) and also to define own rules for syntax detection and Autoshift correlation. A short guide for writing Syntax and Autoshift rules is provided in the configuration file comments.

Appendix B

The TraceBrowser source code

The source code for the TraceBrowser is available at:

- `http://dougdevel.org/tracebrowser/tracebrowser.ddoc` (digitally signed)
- `http://dougdevel.org/tracebrowser/tracebrowser.tar.gz`