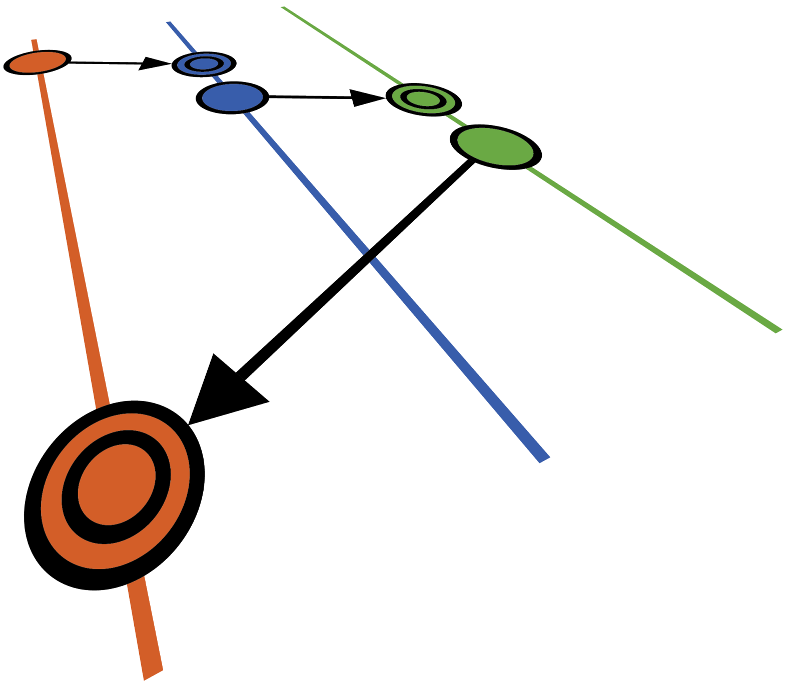
CHALMERS



Visualization of Message Passing in an Embedded System

Design and Implementation of Threadmill

Master of Science Thesis in Computer Science and Engineering

Daniel Kvarfordt

Erik Bogren

Chalmers University of Technology
University of Gothenburg
Department of Computer Science and Engineering
Göteborg, Sweden, September 2014

Visualization of Message Passing in an Embedded System
Design and Implementation of Threadmill

DANIEL KVARFORDT
ERIK BOGREN

Examiner: JAN JONSSON

Chalmers University of Technology
Department of Computer Science and Engineering
SE-412 96 Göteborg
Sweden
Telephone + 46 (0)31-772 1000

Department of Computer Science and Engineering
Göteborg, Sweden September 2014

Abstract

Debugging process communication in embedded systems can be a very complex and time consuming task, especially when dealing with non-determinism. The traditional approach is to use print statement debugging or text logs, but these methods do not cater well to the parallel nature of concurrent systems.

In this thesis we research ways to capture messages in a specific embedded system at Ericsson, and general guidelines on how to best do so in general. We then research and propose a number of visualizations of message passing and argue about their various strengths and weaknesses. Further, we select one visualization and detail the design and implementation of Threadmill, a tool aiming to help developers understand and debug process message communication.

We conclude that the developed visualization, a variant of UML sequence diagrams, shows great promise when debugging message passing applications.

Acknowledgements

We would like to give our thanks to Amin Nahvi and Gustav Evertsson who have been our supervisors at Ericsson and made this thesis possible. They have helped us with many of the practical difficulties we have faced and given us key feedback. We would also like to thank Jan Jonsson, our supervisor at Chalmers, who has helped us with questions of the more academic nature.

Lastly we would also like to give a big thank you to Felix Yao at Ericsson who has given us vital insight into Linux kernel programming.

Erik Bogren and Daniel Kvarfordt, Gothenburg 2014-09-03

Contents

Chapter 1

Introduction

Multithreaded programming is a well established paradigm that makes it easier to develop systems made up of many independently running software components. Concurrency is particularly advantageous when a program needs to deal with multiple external input sources that communicate sporadically, which is commonly the case for embedded systems.

While multithreaded programming has many advantages, developing concurrent applications is often a much more complex task than developing sequential programs for a number of reasons. Firstly, concurrency means different processes execute logically at the same time, and need ways to communicate while avoiding stepping on each other's toes. There are many approaches to this communication, but the first and most basic form of communication is to let the processes access shared memory, which must be done with great care to avoid conflicts. Secondly, it can be important to synchronize threads to make sure that program statements are executed in the intended order. Lastly, threaded programs can have an element of non-determinism, since the order threads execute depends on the scheduler and can vary from one program run to another [1].

The mutual access control and synchronization is often done using locking primitives such as semaphores and monitors, to restrict access or wait for another thread to be in a certain state. However, excessive locking might lead to bad performance since threads spend a lot of time waiting for resources to become free.

The non-deterministic element means that there can be many more possible execution flows of the program compared to single threaded applications. There are exponentially more possible states of the system as a whole, which makes it hard to reason about the execution flow from looking at the code alone. Therefore it is often useful to analyse the flow of multiple runs of a program to help understand its behaviour and to ensure it is correct [2]. Traditionally the way to analyze and to get an understanding of what happens in a multithreaded system has been to make use of print statements or to review plain text logs.

Shared memory communication has many pitfalls that have motivated the development of higher level communication schemes such as message passing, the model discussed in this report. With message passing, threads communicate by sending messages.

When a message is sent it is placed in a queue in the receiver thread and handled by the thread either immediately or at a later point in time. This simplifies thread communication since threads don't necessarily need to read and write shared data, making locking unnecessary. In practice however threads often have access to some shared data for practical and performance reasons.

While message passing solves some of the problems, it can also be hard to debug. Messages might not be handled immediately when they are sent, known as asynchronous execution. This means an error caused by a specific message can arise at a later point in time, when the message is received. The traditional approach to debugging message passing applications is, like before, print statement debugging, but the difficulty in interpreting such logs can be even greater due to the potentially asynchronous nature of message passing.

There are also some inherent issues with using logs as a debugging tool for threaded applications. The first is the amount of data. Without filtering, logs tend to grow large in a short amount of time, making them hard to overview. The second is their structure. Logs are made up of sequential events that don't capture the concurrent nature of threaded execution very well. The third is that they are completely textual. Text is very expressive but requires a lot of mental effort and is slow for humans to interpret. Finding a specific piece of information requires the programmer to scan the text sequentially. This can be remedied to some extent by formatting the text so that it's easier to interpret, but is still far from optimal. All of these issues motivate the need for tools to aid debugging and analysis of multithreaded applications.

1.1 The MPES system

The system considered in this report is an embedded system that uses message passing as the process communication model. We will refer to it as the MPES (short for Message-Passing Embedded System) throughout the report. The software in the MPES unit has many advanced features, and with a software system of this scale it gets increasingly difficult to get an overview of the system behavior. The operating system in the MPES is based on Linux, and has a run-time support system that uses message passing to let threads communicate. The system is derived from a different more complex product, reusing a large portion of the codebase. The legacy code means there are parts the engineers don't know very well, making it possibly hard to debug. There may even be parts that are superfluous in the current system and could be removed. This is however very hard due to the complexity of the system, which can to some extent be attributed to the thread communication involved. The main method of debugging thread communication for the engineers has been to use traditional print statements or logs, an often hard and time consuming task. It is therefore of interest to have tools to better understand the thread communication behavior of the system.

1.2 Purpose

The purpose of this master thesis is to investigate how to best capture and visualize message passing flow data of multithreaded embedded software, and to design and implement a tool that can do so in order to ease the analysis and debugging process and to minimize the time consuming print statement and log file debugging. Our research question is as follows:

How does one best capture message passing in concurrent programs, and how can this information be visualized to help developers understand and debug programs easier?

1.3 Limitations

The system we will work against does not use true parallelism (programs running on multiple processor cores), only threaded programs on a single processor core. We consider a visualization tool for use with true parallel systems to be out of scope for this project. We want to visualize how messages are passed between threads, their content and their context. We do not plan to visualize the full program execution of the threads. For the visualizations we will try to make use of existing graphics libraries to avoid implementing everything from scratch. While we aim to make the tool general and reusable, our main concern is making it work with the MPES system.

1.4 Planned result

In addition to answering the research question stated above, we intend to deliver a working tool that could be used to evaluate the idea and possibly be used by the developers at Ericsson.

Chapter 2

Background

BEFORE work was started on this project the field of visualization was surveyed. We found a strong support for the use of specialized tools to allow for manipulations of the visualization but also to provide other functions helping the debugging process.

2.1 Program visualization

The field of program visualization concerns graphical illustrations of aspects of computer programs. Visualizations can be created by hand, or they can be generated by tools from code or live running programs. Such tools can be dynamic, displaying execution flow, thread scheduling or thread communication, or they can be static and show source code structure [3]. A well known instance of program visualization is UML diagrams, for instance sequence diagrams displaying execution flow, but there are many others, some of which are described in the related work section below.

It's possible to support the analysis of message passing by using visualization tools [4]. Presenting the information graphically makes it potentially much easier to grasp the event flow and to recognize patterns and relations between events. For example, a visualization can make it clear from a glance how much time has passed between events, or to what degree two processes communicate. Thus, program visualization is a valuable technique that is becoming more important [5, 6].

2.2 Related work

In this section we describe some related work and tools. We will refer to our tool as *Threadmill.*

J. Chassin de Kergommeaux et al. [7] discuss in their paper a visualization tool, Pajé, which is similar to the application described in this paper. But as opposed to our tool Threadmill, it focuses mostly on visualising thread execution on a system with multiple nodes i.e. a distributed system. More attention is given to systems with shared data than systems making use of message passing. The main purpose of their tool is to fine tune a large system and improve load balancing, whereas Threadmill is meant to improve understanding and help in debugging. Some interesting points are being made about limiting the intrusiveness of tracing as much as possible, something which is of great importance in Threadmill as well since the system is running on lean hardware. Other interesting aspects of this paper includes the option of inspecting events but lacks the inspection of data values which is an important feature in Threadmill. Pajé creates a visualization that serves as an inspiration to Threadmill but is overly cluttered to be easily overviewed.

Taxonomies of Visual Programming and Program Visualization by Brad A. Myers considers different classifications of program visualizations. The paper describes two main axes: whether the visualization is *static* or *dynamic*, and whether the visualization displays *code, data* or *algorithm* [3]. Myers define dynamic visualizations as "systems that can show an animation of the program running", and static visualizations as "limited to snapshots of the program at certain points." The tool we develop doesn't fit perfectly into any of the categories, since it shows a static view of a dynamic process, and displays both data and to some extent algorithm. A more accurate description might be a *static visualization of process*.

In the paper *Visualization of Message Passing Parallel Programs* [4], Thomas Bemmerl and Peter Braun describe methods to visualize the execution of parallel programs, and the tool VISTOP (VISualization TOol for Parallel systems). In order to visualize the behaviour of a program, it has to be developed using a parallel programming framework called MMK. It takes snapshots of the state of objects, and animates smoothly between the states. VISTOP visualizes other information in addition to message passing, such as object creation and deletion, and it can be used in a distributed computing environment in contrast to our tool. While VISTOP seems more powerful than Threadmill, it puts bigger requirements on the system to be visualized.

Optimal Tracing and Replay for Debugging Message-Passing Parallel Programs by Robert H.B. Netzer and Barton P. Miller [2] begins by stating that a common debugging method is to repeatedly execute a program in search of potential bugs. The problem with this method they say is that message passing programs are often times non-deterministic by design, which introduces a problem of recreating the same execution over and over again to track down a bug. They also discuss a method that allows them to only capture messages that introduce nondeterminacy, which are the type of messages that are the hardest to reproduce in a certain order. Netzer and Miller makes an interesting case for the performance gains and replayability of scenarios in their method. The use case for Threadmill would be slightly different in that the user would use it to try to pinpoint an error, Netzer's and Miller's approach seems to require that the user knows where to look for an error. Nevertheless conserving system resources and finding bugs is something very interesting in our work with Threadmill.

Chapter 3

Theory

To better understand the challenges we face in this project we need some deeper understanding of some of the concepts related to real-time systems and visualization. In our project the most critical knowledge concerning the system is that of the inner workings of a concurrent system with message passing. We also need to have insight into how to structure the visualizations and the graphical tool to make them as effective and usable as possible.

3.1 Real-time systems

A real-time system is a system that must conform to certain timing constraints when performing its computations [8]. These constraints arise from the fact that a real-time system often is connected to various types of sensors that provide fresh data within a given time frame. If the system is unable to perform its computations on this data before new data has arrived then the computations may no longer be valid or useful.

Many of the systems are tailored to perform specialized tasks depending on the application and are very different from the home computers we are used to. These systems rarely offer very high performance, instead the emphasis is on robustness i.e. fault tolerance and to perform a specific task very well. In some applications this becomes extra important, such as in the automotive industry where the reliability of the system can make the difference between life and death.

3.1.1 Inter-process communication

Software that can achieve the appearance that execution is happening at several places simultaneously are called concurrent. In an operating system (OS) such as Linux, processes are started to allow many programs to execute concurrently. The creation of processes on the OS level introduces a great deal of overhead that is not necessary on the application level. Instead, applications can use light-weight processes known as threads, that share memory within the application. Linux provides a library called POSIX[9] threads or pthreads for short. A thread can be initiated quickly without much overhead but introduces some unwanted complexity in the program.

In concurrent programming a frequent problem is how to assure that no errors are triggered because of threads interfering with each others work. There can be critical regions in the program code that should only be accessed by one thread at a time.

To solve these kinds of problems constructs such as locks, semaphores and monitors are used to ensure that only one thread at a time is able to execute the code of a critical region.

3.1.2 Message queues and message passing

In multithreaded system design, threads need to cooperate to ensure that the data they are working with remains consistent. The threads do this by informing other threads when they have finished with a piece of data. In message passing, threads can send messages to other threads. The messages will arrive in the receiving thread's queue and will be read whenever the receiving thread is scheduled to execute, possibly at a later point in time. Message passing thus allows asynchronous communication between threads, meaning that the execution of the sending thread does not block while the receiving thread handles the message, see Figure 3.1 for an example.

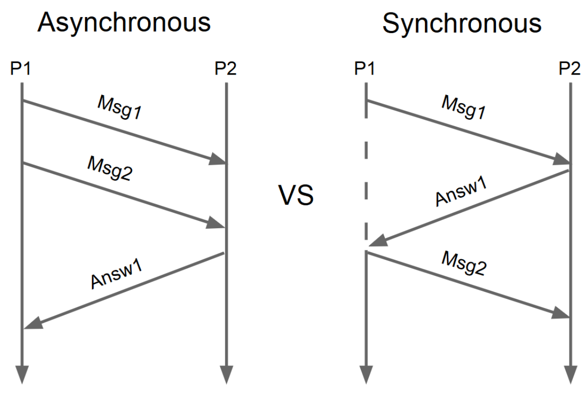


Figure 3.1: The difference between asynchronous and synchronous communication.

3.1.3 Process scheduling

Because no threads in a multithreaded system can actually execute at the same time, the threads have to wait their turn. To determine which thread should be allowed to execute at any given time the system uses a scheduler. There are a variety of different algorithms to accomplish an appropriate scheduling, which one is preferable depends on the system itself. Some threads have time critical tasks and therefore have a higher priority and other threads are not as time critical and thus have a lower priority. One such scheduling algorithm is known as round robin where each thread has a fixed window of opportunity to execute. Which thread is next to execute is often determined by using a first-in-first-out (FIFO) queue. If a thread has not finished executing within the fixed time frame, it will be suspended and put at the back of the queue to let another thread work. There are of course variations on this scheme, for instance the scheduler might want to choose a thread with higher priority to get precedence and some threads may be non-preemptive, meaning they cannot be suspended in the middle of executing.

Most operating systems have a built-in scheduler that handles the scheduling of its own processes. An embedded system could run on a Linux kernel but might have other scheduling needs than the kernel scheduler can provide for. In these circumstances it is an advantage to be able to write your own specialized scheduler that fits your purposes.

3.2 Data visualization

Visualizations can allow the human brain to interpret a large amount of information at a glance. Compared to text they make it much faster and easier to interpret large amounts of data, and to find complex relationships which can otherwise be very difficult to find. This makes them potentially very useful when trying to understand the behaviour of something as complex as communication in an embedded system.

3.2.1 The benefits of visualization

Text is an incredibly powerful tool that has been critical in the sharing of knowledge and ideas, and in reaching the technologically advanced civilization we live in. Its power comes from its high level of abstraction that allows it to express arbitrary information. The downside of this high degree of abstraction is, however, that text takes more effort to interpret than images, whose structure is rapidly interpreted by our visual system [10].

When trying to convey large amounts of data, graphical visualizations can have benefits over sequential formats such as text. They allow data to be summarized in a single picture which can be seen all at once. They can allow for interaction that lets the user filter out irrelevant data, or adjust the level of detail in order to identify both small and large scale patterns. They also make it easier to find unexpected patterns that may lead to new insights, or find problems or errors in the data [6].

While text generally does not distinguish between different types of information, graphical visualizations can indicate meaning of parts using properties such as color, size or shape. They can also very succinctly indicate hierarchies, relationships between different entities and even attributes of the relationships [10].

3.2.2 The costs of visualization

While very powerful, visualization tools can take considerable effort and money to develop. One has to consider whether the benefits of doing so outweigh the costs. For many types of data there are ready-made tools that can be used, graphs in spreadsheet programs being a simple and common example. For other types of data tools may have to be developed from scratch.

One also has to take into account the cost for the user learning to use the tool. If the tool is hard to learn, understand or use, the users will most likely not consider it worth their time. This fact should not be underestimated. The cost of learning the tool can be decreased by choosing well established and familiar types of visualizations and following usability best practices. One should only use novel visualization designs when the benefits of using them outweigh the costs [10].

3.2.3 Visualization stages

There are four main stages of the visualization process:

1. Data collection, where data is collected and stored.
2. Data pre-processing, where the collected data is processed into a format that is more suitable to visualize, for example by filtering out irrelevant data.
3. Mapping the data to a visual representation. This is where the data is turned into an image and a visualization is generated.
4. Perceiving the visualization.

Once the visualization is perceived by the user, systems can allow the user to change the mapping from data to visualization (step 3) in order to explore the data, for example by zooming, panning or changing parameter ranges. This exploration helps the user find what they are looking for, and to better understand the data [10].

3.2.4 Preattentive processing

There are certain properties of glyphs (graphical symbols) in a visualization that are processed faster than others. For example it is very easy to distinguish a red circle among green circles, see Figure 3.2. This phenomenon is called preattentive processing [11]. By using preattentive cues, important parts of visualizations can be highlighted and very quickly perceived by the user, for example when searching or making selections of a subset of items. This is especially beneficial when the amount of data is large and ease of search is important [10].

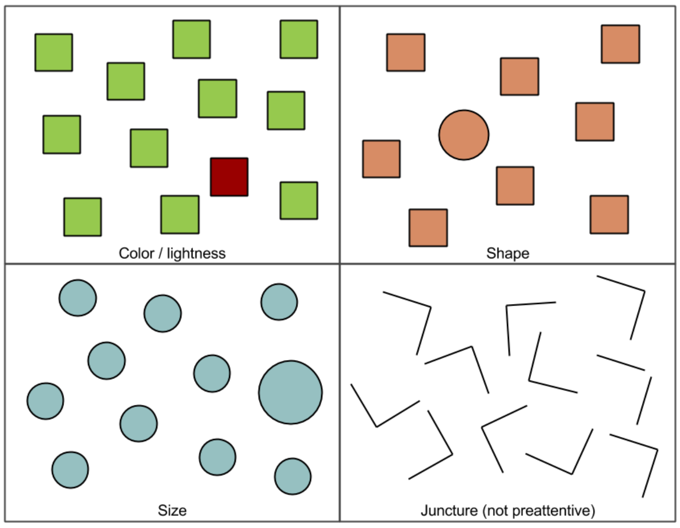


Figure 3.2: Examples of some preattentive and non-preattentive properties.

3.2.5 Gestalt laws

The Gestalt laws are a set of rules on pattern perception. They concern how humans tend to perceive certain cases of visual arrangements and patterns, and can be useful when deciding how to design visualizations to be as easy to interpret as possible [10]. Figure 3.3 illustrates a selection of Gestalt laws that are of relevance to this project.

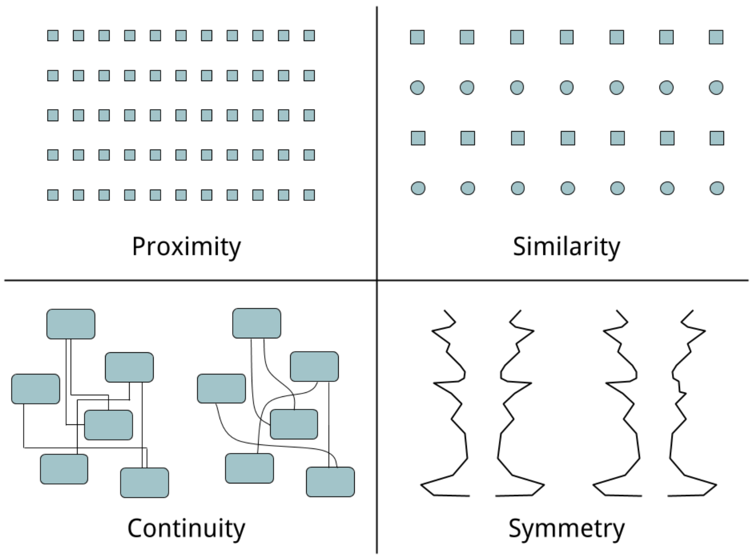


Figure 3.3: Examples of some preattentive and non-preattentive properties.

Proximity

The Gestalt law of spatial *proximity* concerns how we tend to group objects that are close together. This tendency is very strong and hard to mentally escape even if we try. The top left image in Figure 3.3 shows how the rows of more closely placed squares seem to be grouped together. By placing related items closely in visualizations they can be effectively perceived as a group.

Similarity

Similar to the law of proximity, the Gestalt law of *similarity* describes how items that are similar in shape are perceived as belonging to the same group. The top right image in Figure 3.3 shows how the square and circle items appear to be of separate groups.

Continuity

The law of *continuity* says that smooth, continuous elements are more likely perceived as belonging together. Continuous lines are easier to follow than lines with abrupt bends. The lower left image in Figure 3.3 illustrates this effect, where it's easier to connect the nodes in the right graph with smooth lines than in the left.

Symmetry

Symmetry can be used to show how similar or dissimilar items are by placing them in a similar orientation. For small data sets humans can quickly find differences in for example graphs if they are placed in a symmetrical fashion. See the lower right image in Figure 3.3, where the difference in the two lines on the right is easily perceived.

3.2.6 Visualizations of concurrent systems

Debugging concurrent applications traditionally involves analyzing execution flow data in the form of textual output. However, text is being sequential and does not lend itself very well to describing non-sequential concepts. Visualization is a useful tool for understanding complex software, and is very common in the design phase in the shape of UML describing the structure of the system. UML also supports real-time systems to some degree with its sequence diagrams, statecharts, communication diagrams, timing diagrams and activity diagrams.

While there have been attempts to extend UML to better represent real-time systems and concurrency [12], it is generally argued that UML is sufficiently capable as it is [13, 14]. UML also supports "profiles" that allow adding domain specific elements. UML is thus a promising starting point for designing visualizations of concurrent systems.

Chapter 4

Planning

THE PROJECT was planned to be carried out in 20 weeks, divided into four phases: Research, Prototyping, Development and Evaluation. Before starting the project we established the project plan shown below. The project followed the schedule rather closely, with some minor adjustment for certain dates.

4.1 Phases

The phases of the work are described below along with their planned calendar weeks.

- *Research, week 8-11*: Literature study and planning, familiarizing ourselves with the platform and setting up the tools. Finalizing the research question.
- *Prototyping, week 12-16*: Actual project work begins. Prototyping, development of proof-of-concept back-end and front-end. Capturing to file (non real-time visualization). Evaluate different visualization frameworks.
- *Development, week 17-21*: Develop the system with the features we have decided on.
- *Evaluation, week 22-24*: Evaluation and alpha/beta testing of software in real world conditions. Adding eventual missing functionality or new functionality, fixing bugs etc. Finalizing report, preparations for presentation.

4.2 Schedule

Below is our planned schedule for the thesis work. Milestones are marked in **bold**.

Phase	Date	Item
Research	February 17	Thesis work begins
	March 12	Send in the first draft
	March 14	Thesis writing workshop
	March 17	**Planning report ready**
Prototyping	March 17	The prototyping phase beings
	March 20	Attending the first thesis presentation
	April 15	**Working capture $\Rightarrow$ visualization chain prototype**
	April 16	Evaluation of prototype
	April 17	Specify final requirements and scope
Development	April 22	The development phase begins
	April 28	Attending the second thesis presentation
	May 23	**”Release candidate”, tool ready for evaluation**
Evaluation	May 26	The evaluation phase begins
	$\sim$ May 30	Attending the second presentation and performing the opposition
	June 6	Wrap-up of testing and bug fixing
	June 13	**Final report is ready**
	$\sim$ August 12	**Presentation given at Chalmers, master thesis ends**
	$\sim$ August 20	Presentation given at Ericsson

Chapter 5

Method

THE PRACTICAL GOAL OF THE PROJECT was to implement a tool to visualize message passing. The tool would run as an application on a desktop computer and would communicate with a back-end component that we would integrate into the existing MPES system to be able to capture the necessary data.

We began by researching relevant literature, and by familiarizing ourselves with how the MPES system works and what the Ericsson employees' work flow was like in order to better understand the problem we should solve. Very early on, from the first meeting with our supervisors, we tried to come up with concepts for visualizations. The concepts were then revised many times during the project as we learned more about the needs of the users and the limitations posed by the system.

5.1 Work process

We chose an iterative work process because of the uncertainties we faced: the problem definition was loosely defined, there was a lack of concrete requirements and the existing system was quite complex.

We began each week with a meeting with just the two of us, where we discussed the status of the project, what we had achieved the previous week and what we should focus on during the current week. This allowed us to have a good overview of the progress of the project, and be agile in our task choices.

Each week we also had a scheduled meeting with our supervisors at Ericsson, to show them our progress and get continuous feedback.

Since we were studying at different master's programmes, Computer Science and Interaction Design, the division of work was rather clear. We did however share the work as much as possible across the disciplines. Daniel was responsible for the front end tool and visualization, while Erik was responsible for the back end integration of message capturing in the existing system.

5.2 Research

The first part of the project was the research phase, where we researched relevant literature in the fields of real-time systems and visualization, made ourselves familiar with the systems and development environment at Ericsson and tried to get a better understanding of the task at hand.

The results from our literature research are detailed in the ”Previous work” section and ”Theory” chapter. In this section we describe other relevant findings and lessons from this phase, such as how the MPES works in general, how we might capture messages, possible approaches to communicating with the system and possible use cases for the tool.

5.2.1 Description of the embedded system

The MPES system leans towards being a soft embedded real-time system, but has some hard real-time system aspects. The threads used for concurrency in the system are based on Linux pthreads, but have many proprietary extensions. We will refer to the threads as processes in the context of the system. Processes communicate by passing messages, and each process has one or more queues that receive messages. The run-time support system (RTSS) uses a central dispatcher that receives and forwards messages between processes, and decides which process should run (handle a message from one of its queues) next. The dispatcher uses a priority-based scheduling strategy where processes have different priorities, and messages within a queue also have different priorities that govern which process and message is chosen for execution. A token is handed to one process at a time, allowing that process to get write access to shared data. Processes should then return control within a specific time frame to allow another thread to run. The dispatcher works along side the Linux scheduler and is in charge of some of the threads.

Processes can send a message synchronously, which means that the receiving process gets activated immediately and handles the message, or they can send a message and continue executing. In the latter case the message will be handled at a later time when the dispatcher schedules the receiving process.

The processes may have some data attached. The owning process has read and write access to this data, while other processes only have read access.

A process may either be non-preemptive or preemptive. Preemptive processes may at times get interrupted mid-execution, while non-preemptive processes can be sure to run until they choose to wait. During long asynchronous operations, such as file writes, non-preemptive processes may choose to release the token temporarily to allow other processes to execute in the mean time. Preemptive processes may attempt to take the token to get write access, at which point they become non-preemptive until they drop it.

5.2.2 The structure of messages

There are two types of message data structures used within the system, one which is only used in kernel space and another which is only used in user space.

BYTE rcv_qid
WORD snd_qid
WORD snd_proc
BYTE snd_pri
BYTE[] data

Table 5.1: A kernel space message

A kernel space message wraps around a user space message, adding additional header information that is needed in order to let the dispatcher properly handle it. The user space message is the byte array declared as ”data” in Table 5.1 and is free form with no restriction on size.

rcv_qid
snd_qid
snd_proc
snd_pri
field1
field2
field3

Table 5.2: A user space message

A user space message consists of two parts, the header struct which contains fields giving detailed information such as from which queue the message is being sent, which queue is to receive the message and so on. Then there is the body struct which contains free form data that is specific to the process sending the message. See Table 5.2 for an illustration of a user space message.

None of the fields in the existing message structure could be used to uniquely identify a message. Neither could we use the message memory addresses to identify them since they are copied multiple times when being passed between user and kernel space. For this reason we had to modify the kernel message structure by adding an extra field containing a generated unique id number (UID). The first attempt at creating this id number was using a simple counter. Since we believed that the dispatcher had a single thread passing around messages to the different queues, it was somewhat perplexing to see that many messages with the unique id included had the same value. It took some time before realizing that it was not the case that a single thread was executing the functions in the dispatcher instead as it turns out, all processes in the system were making calls to functions in the dispatcher. The first solution to this problem was to use some type of locking primitive but we were advised against relying to heavily on the use of semaphores because of unnecessary locking overhead. The solution was to make use of functions in the atomic library of the Linux kernel. This allowed a value to be atomically incremented without the use of semaphores.

5.2.3 Message capturing approaches

Without data, visualizations are impossible, so the first and initially most important task was to determine in what ways we could capture messages. Without a lot of insight into Ericsson specific domains and with sparse documentation, this proved to be more of a challenge than we expected at first. Some of the approaches we considered are detailed below.

Capturing component

Before having much insight into how the MPES system works our idea of how to capture messages was to create a special capture component. This component would be called from the part in the system that all messages pass through, and store or delegate data about the message.

Alternatively, the component could act as a sort of router that would act as a middle man to all processes. A process would send its messages to the middle man, the middle man would then log this message before sending it to the right destination. There were two main reasons why this approach was never pursued, first it would mean that we would have to change the way all processes send their messages and secondly after learning more about the system we realized that this middle man already exists in the form of the dispatcher.

Binary trace program (Btrace)

The MPES system already had a very efficient event logging program we will refer to as Btrace, that logs events in binary form in a circular buffer. This has two advantages, the memory usage of each event can be reduced by storing it in binary and by using a circular buffer the memory use is kept constant throughout the execution of the system. To illustrate, logging the message type HANDSHAKE_MSG can cost 2 bytes (an integer id) with binary logging in MPES, compared to 13 bytes as a char string, a large difference in memory use. After capturing events, the buffer can be read and parsed by a separate script to produce a human-readable textual log. In this parsing step the script uses a mapping text file to convert the event id's to human readable events.

This is essentially what we needed, so to avoid reimplementing this functionality ourselves we decided to use the binary trace program to do the hard work for us. Our intention was to modify the MPES system to use Btrace to log the events and data we needed, and to control the Btrace program from our front-end. The modifications would be enabled with a preprocessor macro in C (#define VISTOOL) in order to keep the original system behavior intact if compiled without tool support. Using Btrace would save us a lot of work since it was already optimized and in use, and an additional advantage was that we would not use any extra memory, since the circular buffer was already allocated. This approach showed promise, so we decided to attempt it first.

5.2.4 Tool-system communication

One of the first priorities was to establish a method of extracting captured messages from the device. Preferably this method would not distinguish between running the system in a simulator or on a physical machine. There were three main ideas that came up during this research. One was to connect to a physical serial port on the unit, another was to use the already existing method of connecting to the device via SSH and the last was to use sockets to communicate via TCP/IP.

Physical connection

To connect to the device via a serial port was found to have drawbacks that would severely hamper the usability of the tool. Mostly because the simulator is run from a machine in a server cluster to which there is no easy way of connecting physically. But also because it would require the user to walk over to where the device is located to get the captured data which would quickly become tiresome.

Socket connection

A socket connection was mainly considered as a way of extracting the captured message data without first writing it to a log file. Since an important goal was to minimize the impact of logging on normal system operation, we decided against this. Sending capture data in real time over a network connection would be too unreliable and could impact the performance of the system.

SSH connection

Because the system is running a modified Linux kernel features like connecting via SSH and transferring files using SCP are already present. Furthermore the de facto standard of connecting to the MPES unit to perform configurations and other tasks is by an SSH connection. This made the choice of relying on standard Linux features an easy one. Much because of the need to quickly be able to connect and copy data.

Intermediary trace data format

We wanted to decouple the tool as much as possible from the proprietary system. For this reason we early on decided that the device specific trace data should be converted into a device independent format in an adapter layer on the computer running the tool, before being visualized in the front-end. This makes the tool more general and enables easier adaptation for other systems that use message passing.

5.2.5 Tool use cases

We were never able to conduct any formal requirements analysis or user studies. However we tried to understand the work flow used at Ericsson, what problems the developers face with their current methods and tools and their thoughts on how our tool might help them.

We talked to our supervisors to get a better picture of the use cases for the tool, and found the following points to be the most important.

1. **General troubleshooting**. The tool would be useful in the early stages of troubleshooting, to get an overview of how processes communicate.
2. **Debugging timing issues**. The tool might be very useful to find issues with the timing of different messages. For example it could be used to investigate anomalies for reoccurring events such as timer subscription messages.
3. **Follow the flow of data**. Another use case could be to follow the data sent in a message, to see if and where it changes in an erroneous way or gets lost.

5.3 Prototyping

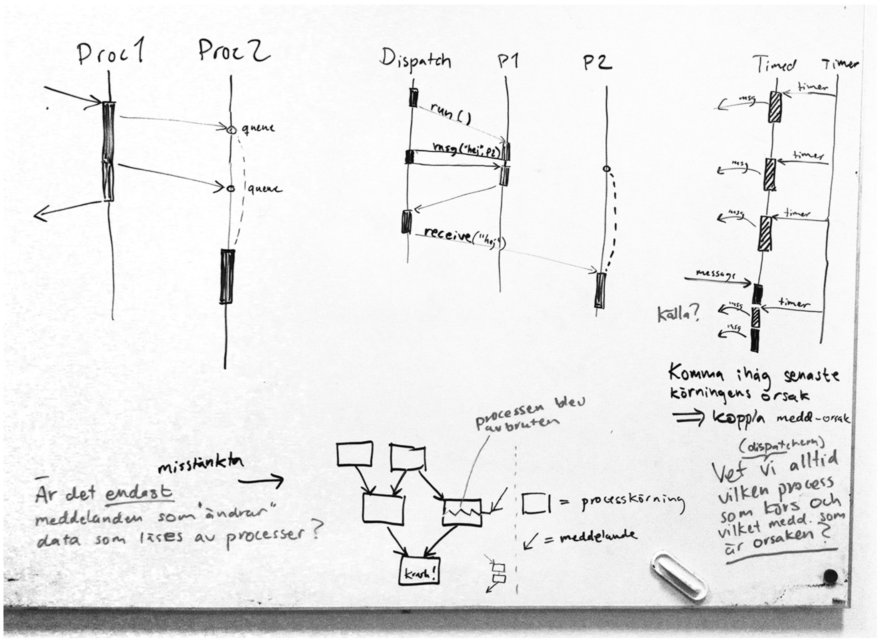
To quickly evaluate our approach and find out which problems we needed to solve, we spent the first weeks working on a prototype that contained the vital parts of the system in their most basic form, what is often referred to as the Minimum Viable Product (MVP). We wanted the prototype to be fully functional in that it could capture, transfer and visualize message passing data in a basic way. The idea was that doing this would let us identify the main problems we needed to solve early on, in order to avoid unexpected issues later in the project.

Right from the start of the project we began trying to come up with ideas for how the visualizations could look. We had various different ideas, a selection of which we describe in this chapter. We rather quickly settled on the basic idea of what became the final visualization, partly because we found it to be intuitive and satisfactory for the use cases found in the research phase, and partly because the time constraints that didn't allow us to explore the design space further.

5.3.1 Ideas for visualizations

In this section we describe a number of ideas for visualizations, and the one we chose to implement into our tool.

The structure of the intended visualization was very loosely defined, so there was a lot of freedom with regards to its design. To come up with ideas we performed a number of brainstorming sessions, where we sketched ideas on a whiteboard, see Figure 5.1 and 5.2. After the sessions we evaluated the ideas based on the insight we had from the research, and kept a few candidates that we refined further. We also showed the ideas to the supervisors at Ericsson to get feedback and figure out what data was and wasn't possible to capture, and to settle on a direction for the visualization.



Proc1
Proc2
queue
queue
Dispatch
P1
P2
run()
msg("hej",P2)
receive("hej")
Timed
Timer
msg
timer
message
Källa?
Komma ihåg senaste körningens orsak
⟹ koppla medd-orsak (dispatchern)
Vet vi alltid vilken process som körs och vilket medd. som är orsaken?
processen blev avbruten
krash!
□ = processkörning
↙ = meddelande
misstänkta
Är det endast meddelanden som "ändrar" data som läses av processer?

Figure 5.1: Whiteboard sketches during the ideation process.

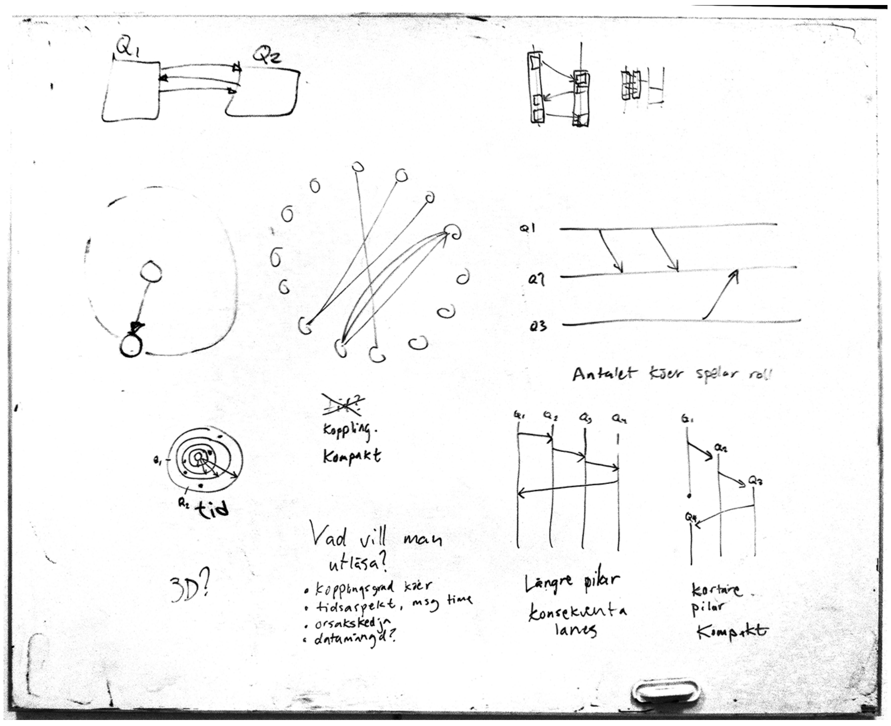


Figure 5.2: Whiteboard sketches during the ideation process.

Message cause-and-effect trees

One of the use cases was *being able to follow the flow of data.* We came up with the concept of message cause-and-effect trees, which are trees with process activations as nodes and messages as edges. Visualizing this structure would allow the user to see what chain of events is the cause of a specific process activation, and thus the potential source of an error. See Figure 5.3 for an illustration of a cause-and-effect tree.

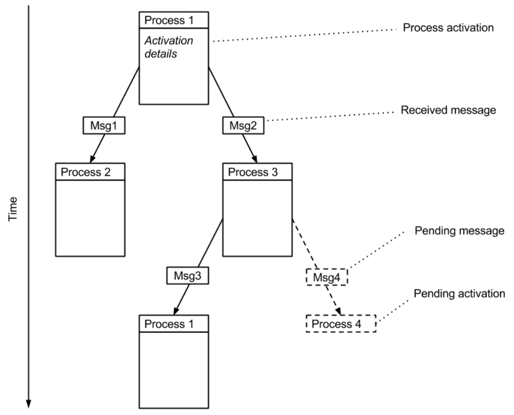


Figure 5.3: An illustration of a cause-and-effect tree.

Gantt-chart style idea

One idea was a visualization inspired by Gantt charts, see Figure 5.4. In the figure processes are represented by horizontal lanes, activations of processes are represented by horizontal boxes, message send events are represented by the vertical lines ending in a circle on the receiving process lane and message receive events are represented by horizontal lines with a circle on the receiving process. For few messages this visualization shows clearly which message send event corresponds to which subsequent activation and it clearly shows the waiting messages in queues. However it quickly becomes hard to overview when there are many messages awaiting activation in a queue at once. We had many variations on this concept, trying to show the messages in different ways, but didn't manage to find a solution that scales well with many messages.

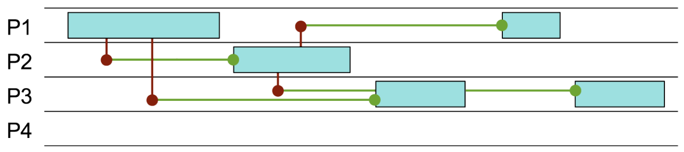


Figure 5.4: Visualization idea inspired by Gantt diagrams.

Chord diagram

Another possible visualization is the chord diagram, see Figure 5.5. It has nodes placed in a circle, and shows relations with bands stretching between the related nodes. However we quickly realized that the amount of useful information that was possible to encode in this diagram in a clear manner was very limited, and it catered poorly to the use cases. This kind of diagram works better for illustrating higher level state of the system, such as the number of messages waiting in queues or the amount of communication or data being sent between pairs of queues.

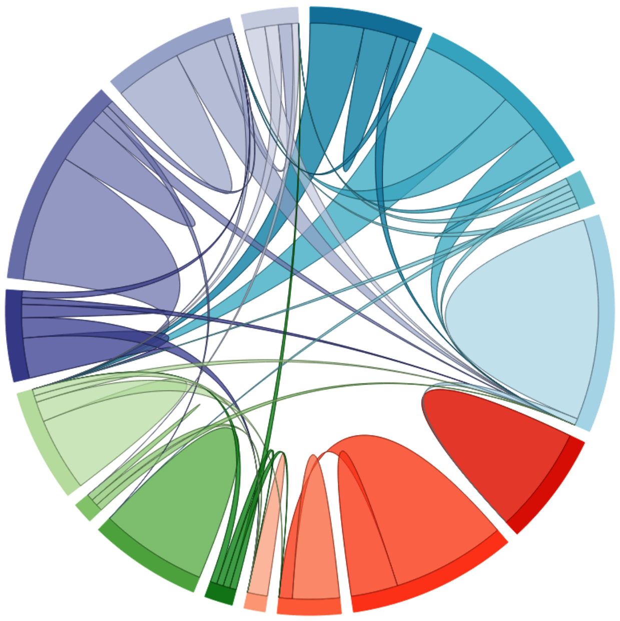


Figure 5.5: Chord diagram generated using D3.js.[15]

Sequence diagram idea

We early on had the idea to use UML sequence diagrams[16] as a basis for the design. Sequence diagrams display interactions between actors over time, which is exactly what we are trying to depict. Additionally, they are very common in software engineering which means that users are likely familiar with them, making the tool easier to learn.

The familiarity and flexibility of sequence diagrams convinced us that they were a good choice. We decided not to restrict ourselves to pure UML however, rather we used it as a basis and an inspiration for our visualization. We believed the benefits of a custom tailored visualization would probably outweigh the cost.

Messages sent to a process are queued immediately by the dispatcher, but there can be a significant time duration before the process is allowed to handle them. We considered two different options when it comes to visualizing this. The first was to illustrate the message enqueueing event with a horizontal arrow going immediately to the target queue, while the second was to draw the arrow from the sending queue to the time instance where the message was processed, see Figure 5.6. According to the Gestalt principle of continuity described in the Theory chapter, it is much easier to follow lines that are smooth and continuous than lines with abrupt changes in direction [10], so we decided to go with the later approach.

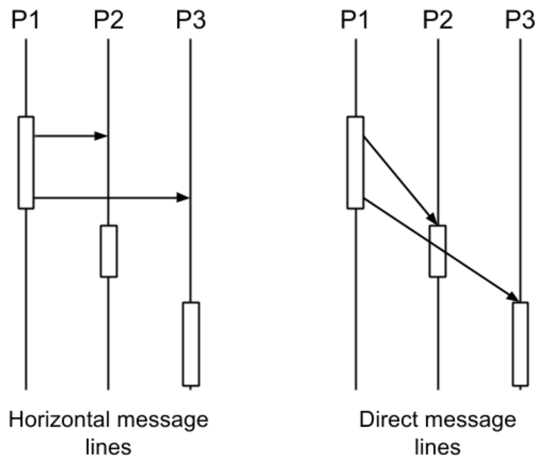


Figure 5.6: Horizontal versus direct message lines

We initially thought we could indicate the extent of individual process activations with elongated boxes, but soon found out we couldn't reliably identify individual process activations in the back-end. For this reason we instead had square boxes for each send or receive event, showing only the information we could be confident in, see Figure 5.7.

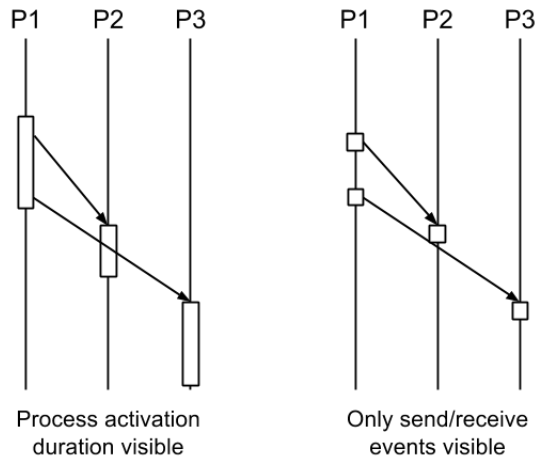

P1
P2
P3
Process activation duration visible
P1
P2
P3
Only send/receive events visible

Figure 5.7: Process activation boxes versus event boxes

Swimlane diagram

Another idea we tried implementing, was to have hierarchical swimlanes where queues belonging to a certain process are grouped under it, and can be collapsed into one lane showing all the events for the process. See Figure 5.8 for an example. This could make the visualization more compact if some processes have a lot of queues, and it gives a more high level view of the communication between processes which might be preferable in some cases. For example processes might use one queue to receive certain messages, and another to respond, which would be more obvious when processes lanes can be collapsed. However the queue-process associations given by the system were incomplete and we found that the added process information was not worth the extra complexity and clutter of the additional graphical elements. We did some test implementations, but stuck with the previous visualization for the final application.

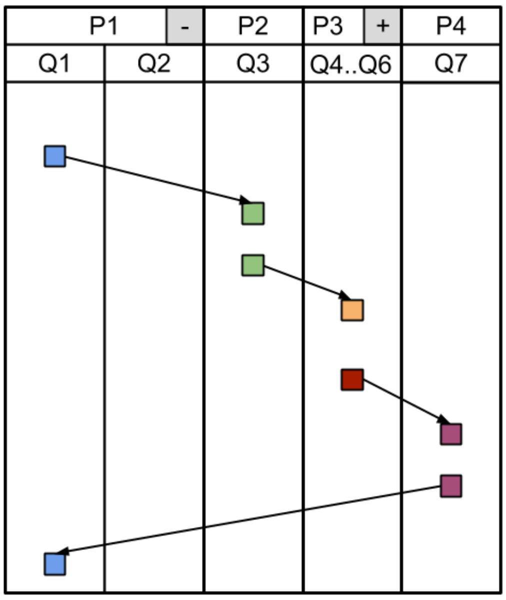


Figure 5.8: Swimlane diagram. The different colors correspond to different queues. The lanes for process 3 has been collapsed so that all its queues (queue 4 to 6) are in a single lane.

5.3.2 Technical choices

In this section we explain some of the significant technical choices we made.

Tool platform

Ericsson wished for the tool to run under Linux since most of their development of the MPES system is done there. Some of the options considered were Java, Processing[17] and HTML5 with JavaScript. We soon settled on making the tool in Java and Swing, which meant it would be platform independent and run under both Linux and Windows. We also had quite a lot of experience with Java from before.

Visualization frameworks

Since we didn't want to implement the graphics drawing for visualizations from scratch, we researched different graph and visualization frameworks. The impression we got was that Java today was not the prime platform for visualization, because most projects were old and not any longer maintained. However, the most promising candidates we found were *Prefuse*[18], *JUNG*[19] and *JGraphX*[20]. If we wouldn't have had Java as a platform we would likely have used a newer visualization framework such as D3.js[15] for JavaScript.

We initially tried to use Prefuse, a predecessor to D3 written in Java. It renders the graphics using Java2D, and supports rich interactive visualizations and a declarative approach to defining the data to be visualized. However, after some evaluation with test data it became clear that it was too hard and error-prone to extend for our purposes. Thankfully we had built the prototype to be modular so that we could easily use other frameworks interchangeably, and we began looking for alternatives. We did not try JUNG in practice, since it seemed aimed at graph processing and algorithms, which we were not after. Rather JGraphX seemed like a good choice. JGraphX was aimed at displaying and editing graphs, and was actively maintained. It uses Swing framework components for displaying the graphs, which we found to be more familiar, reliable and easy to use and so we decided to use it as the main framework for the visualizations in the tool.

Real time capture and visualization

We initially thought we might visualize captured messages in real time. We imagined this to be useful since it would make it possible to witness system behavior as it was happening, for example seeing the immediate effects of sending a specific command to the system.

The capture component would act as a server that the tool would connect to and stream message data from. To implement this, two approaches were considered, either the dispatcher thread would handle both capturing messages and sending them across the network connection or a new thread would be created to just handle network communication.

Both of these approaches had some problems. The main concern was that they would interfere with normal system behavior. Since certain events in the system needs to occur at precise time intervals, we feared that sending network data using the dispatcher thread would affect performance. Sending the data using a new thread would likely also interfere too much with normal system operation.

We decided to drop the feature since it was too risky for the project.

Inspection of message data fields

The MPES system has over a hundred different processes and many of these communicate with each other using a large number of different message types. Processes need to know how to interpret the messages they receive, and distinguish their data fields. That is, they need to have access to the message type definitions.

Threadmill allows the user to inspect raw message data in hexadecimal and ASCII format. To figure out what the data means the user has to have knowledge of the C language struct for that particular message. For example if the user knows that the first field is an integer (corresponds to 4 bytes on most systems) then he knows that the first 4 bytes in the inspection panel of Threadmill correspond to the value of the field.

Naturally, this is not always a practical solution since it quickly becomes bothersome to look up or memorize the many different message structures. It would be preferable if the tool could identify the different fields automatically and present them to the user.

For technical reasons this was not practical as there was no central location containing the message types in the system.

Storing capture data

There were several options under consideration of how to store data on the MPES system. It was desirable to take advantage of a format that was widely used and where well documented API's could be used to interpret the data.

The simplest of the considered formats was CSV (comma separated values) were as the name implies, data values are written to a text file and commonly delimited by a comma. Other formats that was considered was JSON (JavaScript Object Notation) and XML. Both these formats use the idea of storing data as objects. To store a message event as an object seemed like a good approach. Irrespective of the language used to implement the front end, there would probably exists API's to read the data stored as either XML or JSON.

Listing 5.1: Storing a message (JSON)

```
"MessageEvent":
{
    "MessageId"    :  007,
    "Sender"       :  0x8A,
    "Receiver"     :  0x32,
    "Priority"     :  25,
    "MessageData": {...}
}
```

Listing 5.2: Storing a message (XML)

```
<MessageEvent>
    <MessageId>   007  </MessageId>
    <Sender>      0x8A </Sender>
    <Receiver>    0x32 </Receiver>
    <Priority>    25   </Priority>
    <MessageData> ...  </MessageData>
</MessageEvent>
```

In Listing 5.1 and 5.2 our ideas of how to represent message events in JSON and XML are presented. There are a number of advantages in using these formats.

- Third-party libraries aids in extracting data. Eliminates the need for parsing code.
- The formats are structured in a way which is easy to read if there is a need to do so.
- Further improves on the generality of the tool making it easier to be used in another system.

In the end none of these formats were chosen even though they have some advantages. Instead the Btrace tool with it's binary format that can be translated to a human readable text similar to CSV was chosen. This decision was made mostly due to the time it would save us not having to implement already existing functionality. This however, meant that we lost more or less all the advantages of using either JSON or XML.

Listing 5.3: Storing a message in the Btrace format

```
123 MESSAGE_SEND    Uid:006  Sender:0x4F  Receiver:0x13
124 MESSAGE_RECEIVE Uid:006  Sender:0x4F  Receiver:0x13
156 MESSAGE_SEND    Uid:007  Sender:0x8A  Receiver:0x32
156 MESSAGE_DATA    Uid:007  Data:....
167 MESSAGE_RECEIVE Uid:007  Sender:0x8A  Receiver:0x32
```

In Figure 5.3 an approximate example of the Btrace (human readable) format is shown. Notice the similarities with CSV, only here the information fields are separated by a tabular. This somewhat makes parsing the data easier since information on each line can be extracted by splitting on the tabular that sits between the fields.

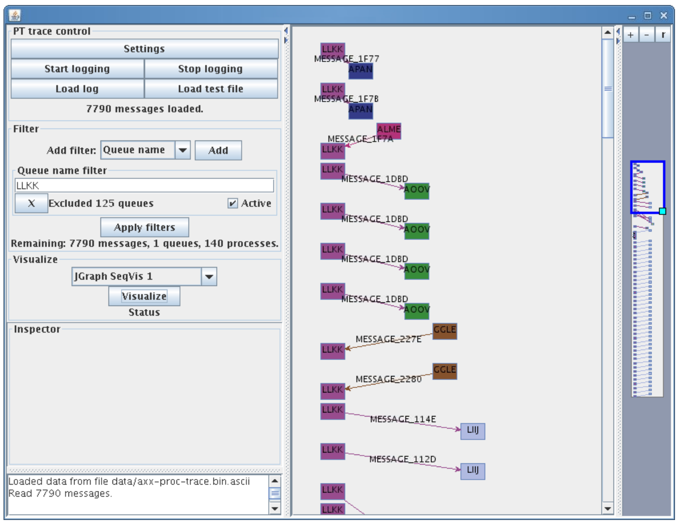
5.3.3 First working prototype

At the end of the prototyping phase we had a limited but working prototype as planned, see Figure 5.9.

At this point we could connect to the device, start and stop logging and download and parse the log in order to visualize it. There was basic filtering support to filter on queue and message names.

The prototype was functional but had a number of problems. Queues were rendered as squares which made them unnecessarily bulky and made them use a lot of space. Since queue lanes were not yet used, the queue names had to be displayed on every square representing a queue, which made the visualization cluttered. There was also no way of telling sending and receiving queues apart by a quick glance, because they all had the same shape. Furthermore, it was not possible to easily assess how messages relate in time as there was no vertical time scale.

However, the goal of the prototype was to have something we could use to evaluate our ideas and demonstrate to supervisors. Much of the functionality that was added in later stages of the project was derived from input and ideas we got during these demonstrations.



7790 messages loaded.

LLKK

Excluded 125 queues

Remaining: 7790 messages, 1 queues, 140 processes.

```
Loaded data from file data/axx-proc-trace.bin.ascii
Read 7790 messages.
```

Figure 5.9: The interface of the first prototype

5.4 Development

This system consists of two main parts. The first is a C-component that integrates with the existing modules on the embedded system and enables the capturing of messages sent between threads in the system, creating a log of the different events that occur. The second part is a visualization tool written in Java that retrieves the captured message data from the device and displays this information in an informative way to the user, and lets them filter and explore the data in different ways. There is an adapter layer that connects the two, and mediates communication and data conversion.

5.4.1 Tool architecture

The tool is divided into three layers, see Figure 5.10. The bottom layer captures messages sent between processes, and is specific to the MPES system at Ericsson. Data about the captured messages is then fetched over an SSH connection to a computer running the

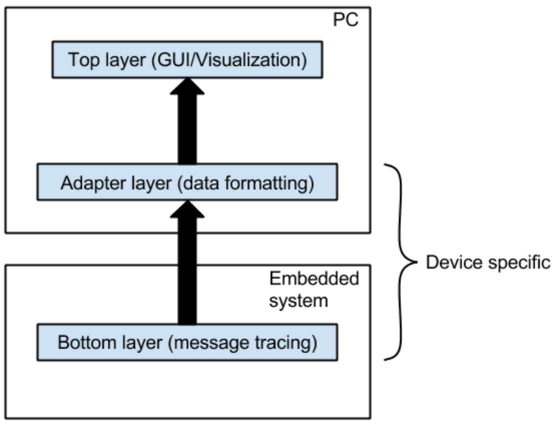

PC
Top layer (GUI/Visualization)
Adapter layer (data formatting)
Embedded system
Bottom layer (message tracing)
Device specific

Figure 5.10: Architectural layers of the tool

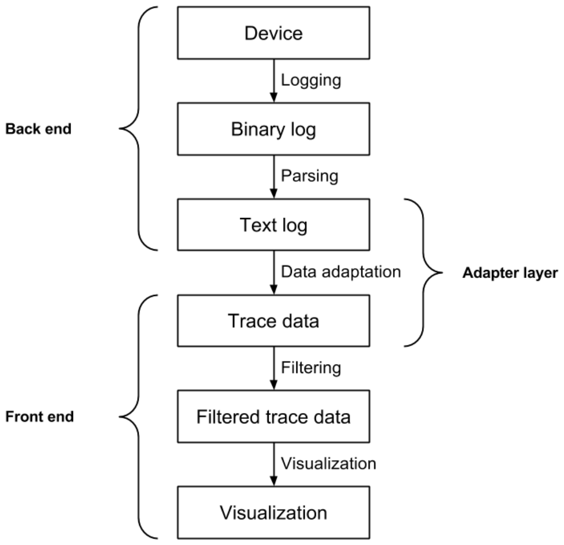


Figure 5.11: Trace data flow in the tool

upper layers of the tool. In the adapter layer, the data is transformed into a suitable device independent format before being presented to the user in the top layer. There the user is able to see graphical visualizations of the data. The division into layers is intended to separate concerns and make each part modular and replaceable. This means the uppermost layer can be reused for a different system.

5.4.2 Back end implementation

The back end consists of C program code, implemented as part of the existing MPES system. On top of that is a Bash script to control and configure capturing. The biggest change of the MPES system which has been made has been to the structure of the system messages. To be able to distinguish messages from one another, an additional field was added to carry a unique identifying number.

All messages in the system travel through the dispatcher, making it a good place to make modifications. Messages are intercepted when a process requests to send or receive a message by making calls to functions in the dispatcher. From the message, useful information to display in the visualization can be extracted, header information and message data.

Data about the captured message is then logged to the circular buffer using the Btrace tool, and later written to file.

Bash script

As previously mentioned the MPES unit is running a flavor of Linux with a small footprint aimed at embedded systems. Linux makes it trivial to connect using SSH. One of the goals during the prototyping phase of the project was to allow for some basic type of communication with an MPES unit. The options were to either let communication functionality reside in a Java class or make use of a bash script. We decided to implement the functionality in bash to keep the implementation simple and to promote modularity.

The bash script consists of a number of different functions that each perform a specific task. All of the functions first establish an SSH connection to the MPES unit before making a call to one of the controller programs on the PT. Among the different functionality is the ability to customize message capturing. For instance there is the possibility to fully exclude message data, which means that message header information is the only part captured. After messages have been captured the data has to be downloaded from the MPES unit. This is done by using SCP.

Typical usage of the bash script looks something like this:

1. The user makes calls to functions in the bash script to first customize the way that messages are captured.

2. The user then controls when to start and when to stop capturing messages.

3. In the last step the user commands the script to make the conversion from binary to a human readable format and then download the data.

Device specific control programs

To allow the front end application to send commands and requests to the back end it was necessary to write specialized control programs in C.

Vistool-conf is one of those C programs available from the MPES unit commandline. Its use is as a tool to provide methods to configure how to capture messages. It's functionality is accessed by calling the program with one of the available command flags.

Filter queues The user can choose to only include message data for a specific set of queues.

Message data The user may completely turn off the capturing of raw message data.

Exclude messages of type The user may find some type of messages uninteresting and can exclude those messages based on the type of message.

Another such program is vistool-proc-trace which is used to control when to start and stop capturing messages. It also has a useful function to clear the ring buffer where captured data is temporarily stored. To be able to clear the buffer is important between starting and stopping capturing, if a clear is not performed then old capture data will be present in a new capturing session.

5.4.3 Adapter component implementation

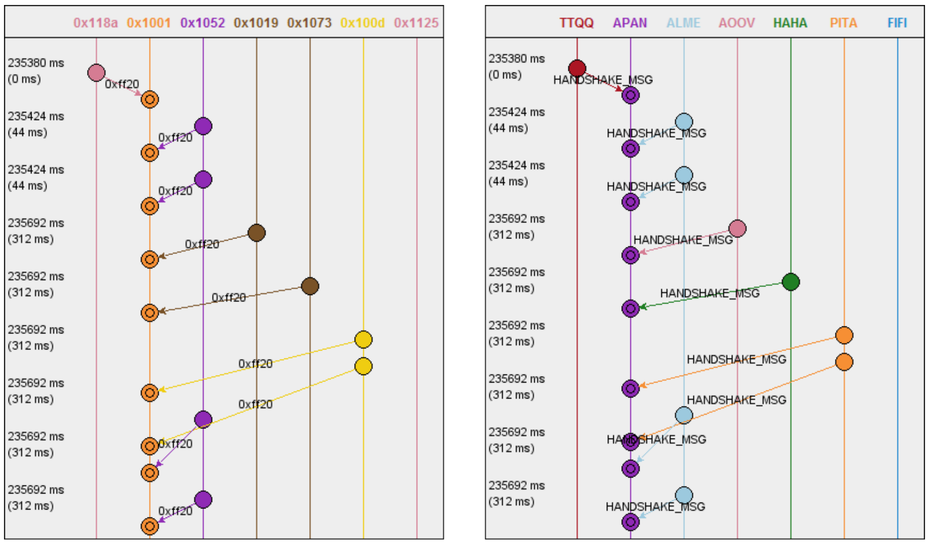
The adapter component is written in Java and runs as part of the front end application. It handles the communication with the node and converts trace data into a device independent format that can be visualized by the front end. The communication is delegated to the bash script that handles the SSH and SCP connections to the node.

Once a log file is downloaded from the node it is parsed and used to build a device independent TraceDataSession Java object. The TraceDataSession object contains queues and messages identified by numeric ids, without any human readable names.

In order to add queue names and message types to the TraceDataSession, the adapter loads and parses parts of the source code of the system, identifying the C language *structs* and *defines* necessary to convert the numeric ids into more descriptive names. This is done in a separate step, which means that the data can still be visualized without string identifiers if the source code is not available.

The TraceDataSession object is then handed over to the front end to be visualized.

Once this was implemented the visualization instantly got much easier to understand and a lot more informative. Suddenly it was possible to get a general idea of what was going on in the system by looking at the visualizations, even for us who didn't have a deep understanding of the processes and messages in the system, see Figure 5.12.



Without message and queue string identifiers. With message and queue string identifiers.

Figure 5.12: The visualization before and after adding string identifiers. Note: Mockup identifiers.

The front end specifies a number of interface methods that the adapter component must implement in order to provide the necessary functionality.

5.4.4 Front end implementation

The front end is a Java application that receives message trace data and visualizes it for the user. It has a graphical user interface where the user can control the device and download message capture data. The data is displayed in a graphical visualization and can be filtered and explored in different ways to find the information of interest.

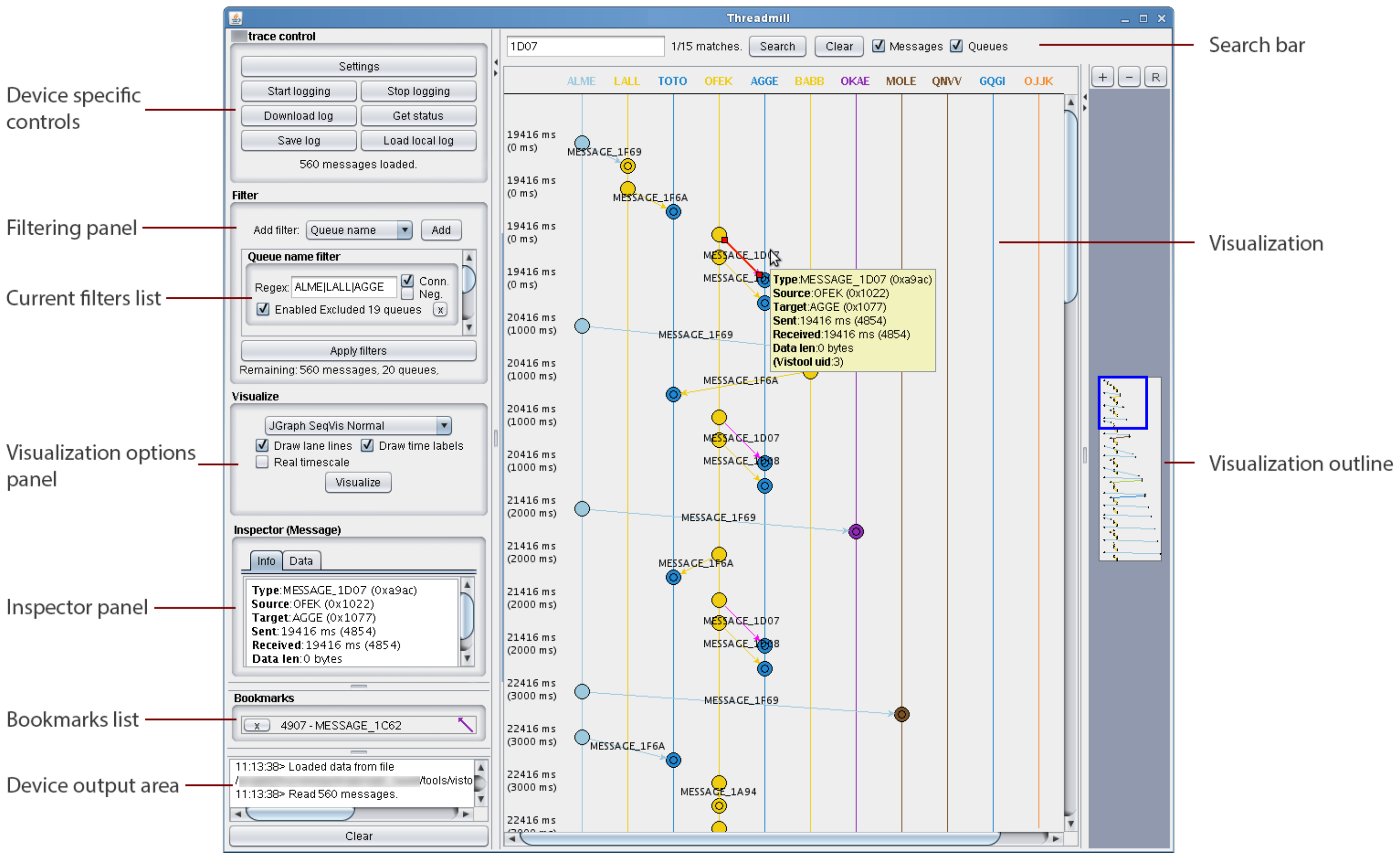
The different parts of the front end are described in more detail below.

User interface

The graphical user interface (GUI) for the tool uses the Swing component framework and can be seen in Figure 5.13. The JGraphX framework is used to render the visualization. On the left hand side we have a side panel, where the child panels are laid out from top to bottom in the order they are intended to be used; capturing, filtering and visualization.

In the top left we have the **device control panel** that lets the user communicate with the back end system to start and stop capturing, download the data or change capturing settings. This panel is provided by the adapter component, and is thus specific to the particular system used at Ericsson. It would be replaced for another system.

The second panel is the **filtering panel**, allowing the user to add different types of filters to filter out the data of interest.



Type:MESSAGE_1D07 (0xa9ac)
Source:OFEK (0x1022)
Target:AGGE (0x1077)
Sent:19416 ms (4854)
Received:19416 ms (4854)
Data len:0 bytes
(Vistool uid:3)

```
11:13:38> Loaded data from file
/                                /tools/visto
11:13:38> Read 560 messages.
```

Figure 5.13: Overview of the graphical user interface

The third panel is the **visualization options panel**, where the user can choose between different visualizations and change options for them. The fourth panel, the **inspector panel**, displays detailed information about messages or queues that are selected in the visualization.

The **bookmarks panel** displays a list of messages and queues that the user has bookmarked for easy access. The list items show the time and name of the item and a small icon displaying the type of item (message or queue event). By clicking on one of the items the user can quickly find the corresponding item in the visualization.

The sixth and last side panel is the console, which shows output from the back end.

In the center we have the actual **visualization panel**. At the top there is a search bar that can be used for free text search for items. On the right we have a navigation panel that functions as an overview of the visualization, and that provides quick navigation by clicking an area using the mouse.

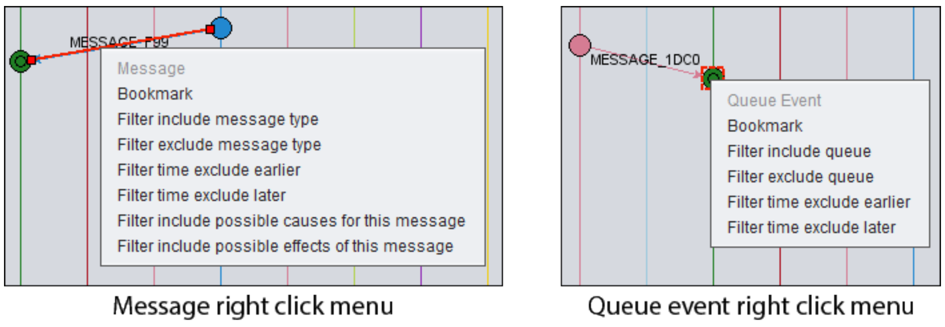

MESSAGE_F99
Message
Bookmark
Filter include message type
Filter exclude message type
Filter time exclude earlier
Filter time exclude later
Filter include possible causes for this message
Filter include possible effects of this message
Message right click menu
MESSAGE_1DC0
Queue Event
Bookmark
Filter include queue
Filter exclude queue
Filter time exclude earlier
Filter time exclude later
Queue event right click menu

Figure 5.14: Right click menu options for items in the visualization.

User interaction

Most actions are quickly accessible using keyboard shortcuts. For example *Ctrl-S* starts capturing, *Ctrl-Shift-S* stops capturing and *Ctrl-D* downloads the captured data.

To quickly browse events in the visualization the user can use the arrow keys to go to the next item of the same type and follow message arrows. If a message of type *HANDSHAKE_MSG* is selected for example, pressing up or down will move to the next message of that type, while left or right will go to the sender or receiver queue events. This allows for very quick navigation between events.

By right clicking on either a queue or a message in the visualization a popup menu is shown, see Figure 5.14. From this menu a user can access options for filtering or bookmarking for the item.

Filtering

The tool supports filtering on a number of different aspects such as queue names, message names, event time and message data size shown in Figure 5.15.

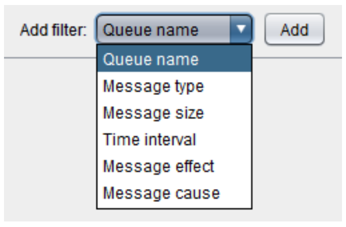


Figure 5.15: The available filters in the tool.

In addition to these fairly standard filters it also allows filtering on message cause or effect. This feature is inspired by the cause-and-effect trees that were considered in the research phase but implemented as a filter instead of a separate visualization. It supports similar functionality although slightly more limited.

By filtering on the cause of a message, the user can focus on all the previous events

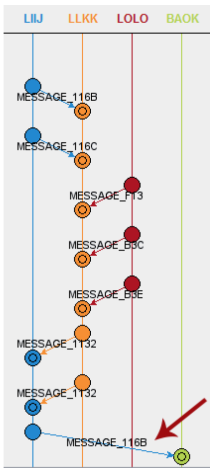


Figure 5.16: Example of message cause filtering on the message marked with a red arrow.

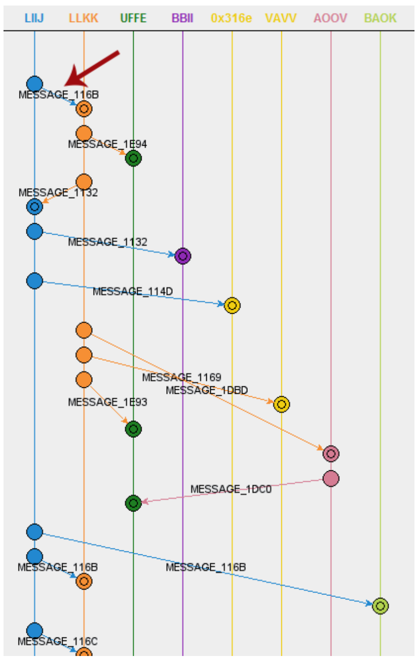


Figure 5.17: Example of message effect filtering on the message marked with a red arrow.

that are possible causes in the message chain. Figure 5.16 shows an example result after applying a cause filter to the highlighted message. The tool also supports the opposite, filtering to show the possible effects of a particular message, see Figure 5.17.

These filters leave a superset of the actual cause-and-effect tree, since we so not have complete process scheduling information.

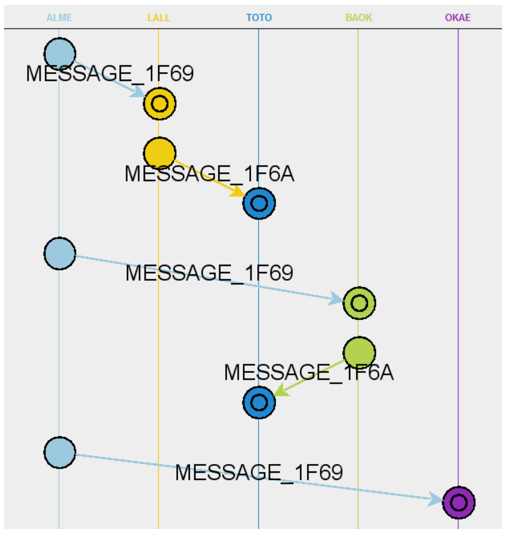


Figure 5.18: Event node and arrow glyphs and colors.

5.4.5 Final visualization design

The basic design of the visualization in the first functional prototype was described in the prototyping section. In this section we describe the changes and refinements made in the development phase, as well as more detailed motivations for our design choices. The final visualization design can be seen in Figure 5.13.

Glyphs

For the lane style visualization we originally had boxes with text, such as the queue name, along the lanes to indicate queue events, and arrows between them to indicate messages, see Figure 5.9. These were big and there were a lot of them making the visualization cluttered. It was also not obvious from their shape if they represented message receive events or message send events. The user had to determine whether the attached arrow was pointing at or away from the box.

We decided to remove the text from the queue event glyphs, and move the queue names to a header for each row that stays in place when the user scrolls vertically, removing the duplicated information and clearing up much of the clutter in the visualization. Further information about the queue events can be seen by selecting them and looking in the inspector panel, or by hovering them with the mouse cursor until a tooltip with more detailed info pops up.

In order to make distinguishing the types of queue events easier, we added an extra circle inside the message receive events. As mentioned in the ”Theory” chapter, added shapes are preattentive cues and make it very easy to see which types events have [10]. The circle is meant to make the glyph slightly resemble a target symbol, indicating that the queue received a message.

Color choices

The colors of the glyphs in the visualization were chosen with care to make them easy to distinguish, but also to make them pleasant to look at (to the taste of the authors). They have a medium brightness to ensure enough contrast against the white background aswell as against the black outlines.

Queue lanes have colors based on a hash value of their name, which makes queue colors always consistent between runs. Compared to simply using an ordering of the colors for consecutive queues, this has the disadvantage that two neighboring queues can have the same color. However, we found that the advantage of being able to recognize queues based on color outweighs this.

Message arrows get the color of the source queue to indicate which queue they were sent from. This can be useful if there are multiple arrows going to a queue lane from off screen, to quickly see if they come from different queues. However this is just a hint to the user, and if they have the same colors the user would have to follow the arrows to the sources to see if they are indeed the same or just share the same color, see Figure 5.18. The arrow coloring is also useful when trying to get an overview of which queues are the most active. The queues that are the most active during a time interval will have many arrows going out, making their color dominant in the navigation panel, see Figure 5.21 for an example.

Time scale

By default the events in the visualization are laid out vertically with equal distance between consecutive queue events in time order. There is also the option to lay them out in a time-proportional fashion. This can be useful when trying to understand how events relate in time. Figure 5.19 shows an example of the insight this feature can give. In the left hand figure the red and orange messages seem to take up a majority of the time, while the real time scale in the right picure reveals that they actually happen in a short burst in the middle of the time interval.

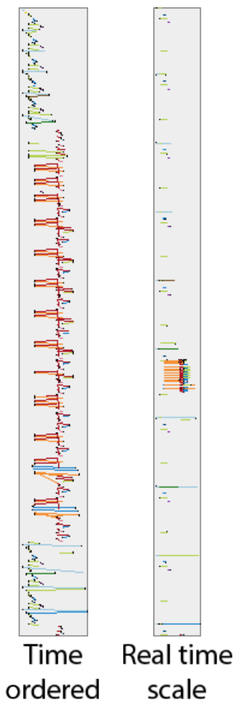


Figure 5.19: Time ordered events versus real time scale.

Handling incomplete data and filtering

There were some special cases we had to consider when creating the layout of the items in the visualization. For example there might be send events without corresponding receive events, or one of the two queues for a certain message might be filtered out.

If a message has been sent but not received, this can be an important piece of information. Therefore we chose to display this in the visualization with a short gray arrow without any target.

When only one of either the sender or receiver queue of a message is filtered out, we chose to not display the message at all since we want the user to be able to focus on only the interactions between the remaining queues. Otherwise there would have been too many incomplete event nodes cluttering the visualization.

5.4.6 Visualization lane layout

In the first iterations we found that if the visualization contains data from many queues it can become somewhat hard to overview. The queues were simply placed from left to right in the order they appear in the captured data, in what we will refer to as a *simple forward layout*. This meant that the horizontal arrows between queues sometimes became very long and the user had to scroll since the sender and receiver queue did not fit on one screen. It also meant that arrows are likely to overlap labels and other elements of the visualization, making it messy and hard to read. We found many possible solutions to the problem of ordering the queues in a more intelligent way, but none of them fully solved the problem.

One heuristic that might be useful is to consider the total horizontal edge length in the graph. Assuming that horizontally shorter arrows would result in a better and more compact visualization, it might be of interest to try to minimize the total sum of these lengths. We have considered a number of approaches for doing this.

Brute force optimization

The first approach to minimize edge lengths would be to test all possible permutations of lane orderings, but this quickly becomes too computationally heavy. The number of orderings are on the order of $n!$, and the edge length computations are not insignificant. The complexity to calculate the total edge lengths for all permutations thus becomes $n! \cdot m$, where n is the number of queues and m is the number of messages to display. With over a hundred queues in many data sets visualizations we have tried, we reached the conclusion that the problem can not be brute forced in reasonable time.

Disjoint graph identification

The second idea to come to our minds was to identify disjoint groups of queues that only communicate within the group, and then ordering them so that they don't overlap. After implementing this we saw that this solves part of the problem for certain cases, but if the groups are big there can still be a considerable amount of long horizontal arrows within groups. It is possible, and from our experience rather common, that the queues and messages in the visualization all belong to a single connected graph, in which case there is only one disjoint graph and the method does not make a difference to the result. This makes the approach less useful, and motivates the need for other ways to lay out the connected graph lanes more intelligently.

Force-directed layout

A third approach is to use methods from force-directed graph drawing, where the edges of a graph work like springs that act upon the nodes in order to try to find a visually pleasing placing of nodes. In our case we are only interested in the horizontal positioning of the lanes, so we have a mass-spring system in one dimension.

We decided to try and implement this approach in the visualization tool. The algorithm we used is based on the one by Eades[21], based on Hookes law for the force exerted by springs, but modified to use logarithmic attractive force scaling instead of linear to avoid excessive forces for nodes far apart. We also modified the algorithm by adding weighted edge springs with different stiffness. Our algorithm is outlined as follows:

1. Each queue is represented by a node (mass) and given a (semi-random) starting position.

2. Between each pair of nodes whose queues exchanged a message we add an edge (spring) with a stiffness equal to the number of messages passed.
3. The mass-spring system is simulated for a number of iterations (we used 1000).
4. The final positions of the nodes are used to order the lanes for the queues.

The semi-random start position was chosen to be identical between runs so that the user doesn't get disoriented by different orderings for identical data. The force equations used in the simulation were the ones used by Eades [21] with the addition of a spring stiffness factor that aims to make well connected queues attract each other more. The attractive force equation used was

$$F_a(n_1,n_2) = (c_1 + C(n_1,n_2) * c_e) \cdot \log d / c_2 \tag{5.1}$$

where n_1 and n_2 are two nodes, d is the distance between the nodes, C is the edge weight (spring stiffness) between the nodes and c_e is a edge weight contribution factor. The repulsive force equation used was

$$F_r(n_1,n_2) = c_3 / \sqrt{d} \tag{5.2}$$

where d is the distance between nodes n_1 and n_2 like before.

We found the constants suggested by Eades, $c_1 = 2, c_2 = 1, c_3 = 1, c_4 = 0.1$, worked well. A large c_e has the possibility of making the simulation unstable because of excessively large forces for pairs of nodes that are very tightly connected. Empirically we found that an c_e of around 0.05 gave the most pleasing results for our test data, but this might need to be tweaked based on the number of queues and messages.

We plotted the positions of the nodes over time to verify that the simulation behaved as expected, and became reasonably stabilized within the given number of iterations, see Figure 5.20. In the top of the figure nodes are placed semi-randomly along the x-axis. The simulation then starts and each node is attracted to the nodes it is connected to. About halfway down the plot the nodes have settled in their final ordering. It can be seen how assumably well connected nodes cluster in the middle while less well connected nodes are pushed to the sides.

However we found that the results were not much better than with the simple forward layout, see Figure 5.21 for an example. While the total edge length becomes smaller, as appearent from the lower edge coverage in the figure, the resulting ordering is in our subjective opinion overall less structured and harder to find patterns in, especially for small visualizations. It is also a lot less predictable. Compare for instance the very top part of the visualization, where the simple forward layout is much more structured. In the bottom half of the visualization however, the force-directed layout is arguably easier to interpret since the edges are generally shorter.

We also believe that force-directed layout does not handle 1D layout problems very well since the repulsive force which is proportional to the inverse distance gets very large when nodes are close, meaning they can not easily pass over each other in a 1D space

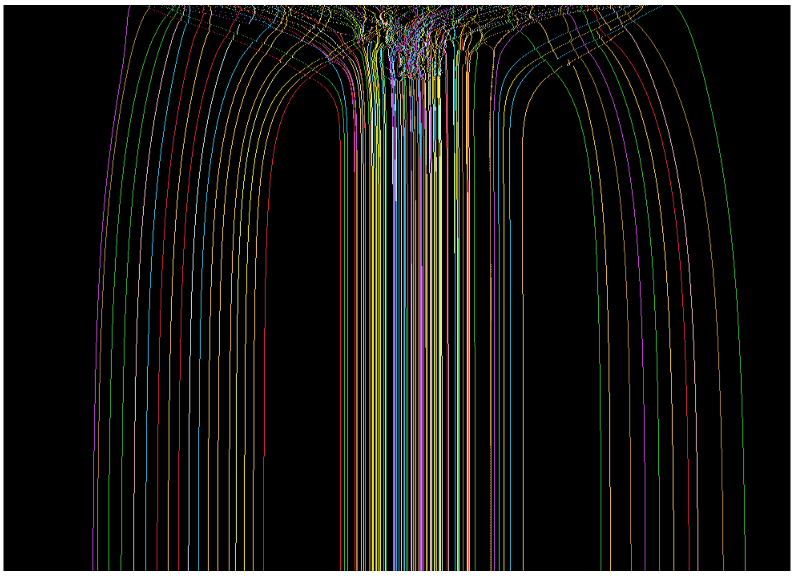


Figure 5.20: 1D plot of the mass positions over time, from top to bottom.

to explore different permutations easily. In practice they can occasionally pass because of the discrete simulation steps used (Euler integration), but in a perfect simulation the repulsive force would grow towards infinity as they approached each other. In 2D on the other hand, they can use the extra dimension to find a path around each other.

5.5 Evaluation

5.5.1 Message data integrity

After implementing the functionality to log message data, we wanted to make sure it was copied correctly. One fear was that there would be problems regarding the use of little and big endian since the simulator and the MPES node might not necessarily use the same variant.

Because our knowledge of the system as a whole is rather limited, just looking at some reoccurring messages to determine that the message data had been consistently copied was not sufficient. After researching some other programs in the system we found a program that would let the user define a message on the command line and send it to any given queue. We used it to ensure that data was intact when being inspected in the visualization.

5.5.2 Performance testing

It was important that the tool did not claim too much resources from the system, since then it might interfere with the system's normal behavior which could result in misleading trace results. Since the tool leverages the existing logging system, which uses a

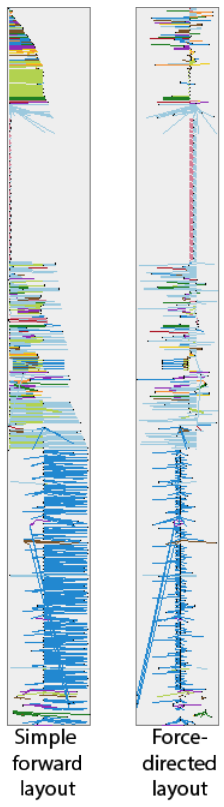


Figure 5.21: Comparison of visualization results from simple forward layout and force-directed layout

pre-allocated circular buffer, we can be confident that we do not use too much memory. However, the CPU performance is a potential issue. The logging of message data could use up too much processing time and thus interfere with other processes. Therefore it was important to perform some performance testing to be sure that the tool is lightweight enough to give a correct picture of the messages passed in the system.

	Min	Max	Average
With tool	101	102	101.3
Without tool	100	102	101.3

Table 5.3: Boot times in seconds. Min, max and average time for 3 measurements.

These tests were performed in a lab environment on a hardware node. The simulator was not taken into account since its performance doesn't equal that of the real node.

	Min	Max	Average
No logging	141.96	142.26	142.11
Logging without tool	143.28	143.49	143.39
Message logging with data	141.84	142.17	142.01
Message logging without data	141.86	142.27	142.07

Table 5.4: Message sending stress test. Min, max and average elapsed time in seconds for 2 measurements of sending 1000 messages containing 400 byte of body data each. *Logging without tool* refers to the default Btrace logging in the system.

None of these tests attempts to mimic realistic working conditions for a node , rather they are meant to justify our assertion that our modifications do not seriously hamper the performance of the system.

In Table 5.3 the time it took to reboot the system compiled with and without Threadmill can be seen. The boot times were retrieved from the system start up logs. The start up times we retrieved from the system are only accurate to the second, so we can not draw any strong conclusions, but for all practical reasons the difference in start up time is negligible.

To perform the message sending stress test, a tool to send mock-up messages was used. A thousand messages were generated, each 400 bytes large. This is a quite extreme case that never happens under normal conditions. Figure 5.4 shows the results of running a stress test in some carefully chosen scenarios. We compare results when no logging is being done, when data is included in the logging, when logging is done without including message data and finally logging using the default logging behaviour of the unmodified Btrace program.

Results in both tests show no significant differences between the scenarios. The measured differences are insignificant, so we conclude that the capturing of messages and message data has no significant impact on system performance.

User testing

We did not perform any structured user testing. The target group was rather narrow, developers of a specific system at Ericsson, who were most often busy with their own work. During the project we understood that the tool would not be used very frequently in daily work, rather it would be very valuable for some more uncommon tasks, so people were not that eager to test it. Due to restrictions on what information we could bring outside the company user testing with non-employees was also out of the question.

We did however conduct short informal user testing sessions during our weekly meetings with the supervisors at Ericsson, as described in the ”Work process” section, which resulted in valuable feedback on functionality and usability. We also performed a short demo session at the company during which we got some feedback that helped us. In particular we found out that certain message types were often not important, and that we might want to add a feature to filter them out already in the back end.

Chapter 6

Result

THIS MASTER THESIS has resulted in the design of a visualization of message passing, and in a tool application that can be used to control capturing of messages and help developers analyze and debug message passing. The ordering of queue lanes in the visualization was investigated, and a few different approaches to optimizing the ordering to minimize edge lengths were attempted.

6.1 Capturing messages

We found the binary logging using a circular buffer, which already existed in the MPES system, to be adequate for capturing the message data required for the visualization. The circular buffer uses a constant amount of memory and the binary logging minimizes log entry sizes, both desirable properties in an embedded systems with limited resources.

The time constraints did not allow us to try other approaches, thus we have no hard data on the effectiveness of this method compared to others.

6.2 Message passing visualization

From informal user tests with supervisors and employees, the visualization based on sequence diagrams was found to best fit the purpose. It clearly illustrates the message flow and aids debugging and analysis of message passing applications. It supports the tasks given by the use cases: general communication overview, debugging of timing issues and debugging of message data flow issues.

A minor contribution is the exploration of different ways to order the queue lanes in the visualization. The simple forward layout resulted in some very long message lines for large data sets, but we found that it was the most satisfactory for moderately sized data sets because it was more predictable. Our force-directed layout was flawed due to being simulated in 1D instead of 2D. Still it resulted in a lower total horizontal edge length compared to the simple forward layout, but a visually more disorganized result.

6.3 The tool

All the available operations when capturing and visualizing messages in the system can be performed using Threadmill. It allows the user to start capturing, stop capturing and download the captured data. It also allows the user to set options on the device before starting capturing. Once captured data is downloaded it is converted to a device independent format, which can be visualized.

The device independent data format ensures that the back end implementation is decoupled from the front end, allowing the tool to be reused more easily.

The tool provides filtering options such as filtering on queue name, message type, or message causal dependencies, and is easily extended with new filters.

While not the main focus, the user interface was found in informal user tests to be easy to understand and use, with shortcut keys and tool tips that gives it a fast and intuitive work flow. It has convenience features such as free text searching, stepping through the message chain using the arrow keys, and bookmarking of individual events or messages.

6.4 Performance impact

The performance impact of the tool was evaluated by running the MPES system with and without different features of the tool activated, measuring the time elapsed for certain operations.

We found that the differences in time between using and not using the tool in these scenarios were insignificant, see Table 5.3 and Table 5.4 for measurement details.

Chapter 7

Discussion

THE PROJECT went for the most part according to plan. There is a learning curve when working in a new environment which somewhat slows down progress. We didn't have any detailed requirements on the tool or the visualization, so we have been free to take the project in the direction we have seen fit. Our work has been very dependent on feedback from our supervisors but ultimately it has been up to us to decide which features should be included. We feel that we have reached a result that is complete enough to be used with the MPES system at Ericsson, but that is also general enough to be used for other systems.

7.1 Process discussion

Our focus in this project has been to find a way to effectively visualize message passing. We have also aimed to leave Ericsson satisfied with a working tool that could be used by the engineers in their everyday work. To achieve these goals our development has been very open to feedback from our supervisors and other staff at Ericsson. Many of the features implemented were done by requests from the employees. We feel that in this aspect our work has been very successful although there is still room for improvement.

7.2 User testing

There were plans for formal user testing sessions that could have provided more concrete metrics and justifications for the benefits of the tool and visualization. We wanted to let users perform certain debugging tasks with and without the tool, or with different visualizations, and measure the difference in time or in number of operations required.

However, we did not have the time or resources for formal user tests. Our supervisors were very helpful giving us a lot of their time, but the other employees were understandably busy with other tasks. Another complication was that we were not allowed test with non-employees because of company policy.

For the same reason focus groups, workshops and other such methods were out of the question.

7.3 The tool

The limited time and relatively large extent of the project meant that we were not able to implement many different visualizations in order to compare their qualities. However, the use cases we had laid down guided our choice between the alternatives we had come up with and made us more confident in our choice of which visualization to start implementing. With more time we could have explored a larger number of design approaches, for example visualizations that are more specialized for certain use cases, such as when debugging synchronization issues.

From our supervisor meetings and testing, we have concluded that the final visualization fulfills the use cases we had identified in the section ”Tool use cases”:

1. The visualization provides an overview of the messages passed during the capture session. Filtering and searching makes it easy to find data points of interest.
2. It is straightforward to investigate timing issues using the visualization. Events are shown in ascending order by time. By default they are positioned with equal vertical spacing, but there is a check box that activates the real time scale mode, which is preferable when trying to see the actual time relations between the events.
3. The visualization makes it easy to follow the flow of data. The user can select an event and then step through all events of that type or follow message arrows by using the arrow keys. At each step the data can be inspected in the inspector panel.

7.4 Lane layout

The implemented visualization had the problem that message arrows sometimes became very long when messages were sent between queues whose lanes were positioned far apart, making the visualization hard to navigate.

We tried to solve this problem using force-based layout, but did not provide any hard data on its effectiveness in this report. We measured the total edge lengths in the visualization for the different lane layout algorithms during development, but did not save any numbers due to time pressure and that we felt this aspect was somewhat of a tangent. In hindsight it might have been a good idea to document and motivate our choices more rigorously. Also, the heuristic of minimizing the total edge length was chosen without much backing, but we believe it is a reasonable assumption that a lower total edge length results in a clearer visualization.

The force-directed layout might be improved by moving the simulation into two dimensions to allow nodes to pass around each other, and then projecting the resulting positions down to one dimension before getting the final lane ordering. To avoid clusters of nodes overlapping when projected, one could apply a slight force towards the line of projection to keep the graph stretched out along the 1D-line. One might also add some randomness to avoid getting the simulation stuck in local minima.

After implementing the force-directed layout we made some more research into the problem, and realized the problem of minimizing the edge lengths was in fact an instance of a standard problem called the *Minimum Linear Arrangement problem*, which is NP-hard for the general case [22]. Thus there is no known polynomial time algorithm, but there are efficient evolutionary algorithms for the problem to find approximate solutions. For example Rodriguez-Tello describe a memetic algorithm called MAMP that finds near optimal solutions [23]. However we found out about these too late in the project to have time to implement them, and the algorithms would need to be modified slightly to take into account the weighted edges in our graph.

In the end none of the approaches gave perfectly satisfactory results, and the initial simple forward layout was most often preferable since it was more consistent between runs.

7.5 Challenges

A big challenge of this project has been learning how the existing system works, and finding a way of making our modifications without affecting the system behavior. The system had many parts and not all of them were thoroughly documented. Furthermore, a not so obvious time sink has been the slow build times for the back end system, often taking at least 5 minutes to finish, which slowed down iteration times considerably.

We also faced the challenge of how to make the tool sufficiently decoupled to allow it to be used for another system. We believe this was accomplished, but testing the tool for another system was out of scope for this project.

7.6 Future Work

7.6.1 Visualization improvements

It is possible to encode more information into the glyphs of the visualization. For example it might be useful to see the amount of data in a certain message from a glance by connecting it to the thickness of the arrow, or to distinguish certain types of messages such as sync or timer messages by color or shape. Furthermore, the number of messages waiting in queues (the queue utilization) could be visualized. The usefulness of such indications depends a lot on the area of use for the visualization, but might be worth pursuing.

7.6.2 Swimlane diagram

At the moment the visualization only displays queues and messages, and does not concern itself with processes or programs. This is reasonable if the number of queues is not much larger than the number of processes, but can get hard to overview when processes have many queues. A message received on one queue of a process can result in a message sent from a different queue in the same process. The swimlane diagram visualization briefly discussed in the ”Prototyping” section could be a good alternative when the focus is more on the communication between processes than message flow between queues, since it allows collapsing all queues belonging to a process into one lane.

7.6.3 Cause-and-effect trees

The idea behind cause-and-effect trees was to make it possible to follow the chain of sent messages and process activations to see the chain of events that caused a specific process activation, see Figure 5.3. We initially thought this would be very valuable in tracking down the cause of errors, and the supervisors at Ericsson also thought the idea had potential. However, due to technical reasons in the way the system at Ericsson works this was a lot harder to implement than we initially thought.

Specifically, it would be necessary to log all process activations. This was not feasible as it would require modifications to the kernel that were out of scope for the project, and might also affect the performance of the system. Cause-and-effect trees could be investigated further.

7.6.4 Indications of bad system performance

An indication of performance issues in the system could be to observe that messages are being discarded. The reason for this could possibly be due to message queues overflowing. Message queues overflowing could in turn be caused by starvation as a result of faulty message or process priorities. By attaching the state of queues to the captured data, this information could also be visualized to aid the user in detecting when there is an imbalance in the system.

7.6.5 Distributed systems

Our tool only concerns messages in a single embedded system. A future modification could be to make the tool work with distributed systems, where nodes communicate over a network. In the system we have worked with it’s unusual that messages never arrive at their destination, unless there is a bug in the system. In a distributed system however, this would be far more common, and it would be of greater interest to show dropped messages in the visualization.

7.6.6 Structured message data inspection

The message data in Threadmill is displayed raw as hexadecimal bytes together with an ASCII representation. It would be preferable to be able to see the names of the fields tied to their data to eliminate the need to manually look up message structures to figure out which data belongs to which field.

Chapter 8

Conclusion

THIS PROJECT investigated ways to visualize message passing in an embedded system. We described the value of visualizations for complex systems and message passing in particular. Several viable visualization designs were thought out on paper, and we made the decision to focus our attention on one of these. This decision was made to ensure that we could produce a visualization of high quality that could be included in a working application within our limited time frame.

A tool meant to help developers understand and debug complex multithreaded software was developed. The tool displays the aforementioned visualization, and has functionality to manipulate the visualization and quickly find the information of interest. A priority in the design of this tool was to ensure that the capturing of messages had negligible impact on system performance, something our tests showed was most likely accomplished. Another goal was to keep modifications needed to capture messages in the system as non-intrusive as possible. By slight configurations the user is able to prevent all program code related to the tool from being built together with the system.

Although user testing has been fairly minimal, it suggests that the final visualization is very useful to aid the development of multithreaded message passing systems.

Glossary

Bash Bash is a Unix shell that supports a basic language that allows the user to write simple but efficient programs. Often these programs tackle everyday problems in computing. 31

cause-and-effect tree this refers to a tree formed by messages sent, the resulting process activations, and the messages sent in turn by the activations. 21

Endianness Two ways of ordering data in memory is big-endian and little-endian. In Big-endian the most significant byte is stored in the smallest memory address and the least significant byte is stored in the largest memory address. In little-endian on the other hand the most significant byte is stored in the largest memory address and the least significant byte is stored in the smallest memory address. 42

Hard real-time system is a real-time system where breaking timing constraints will mean the result is useless, and could possibly result in serious consequences. In an aircraft, malfunctions of this kind could possibly jeopardize the safety of the flight. 15

Legacy code Legacy code is dated code that is not well documented and which few know how it works. If you ask a programmer to explain this type of code he will very possibly reply ”don’t worry, it just works”. 2

MPES an acronym for Message Passing Embedded System, the term we use for the system at Ericsson that we worked with during the project. 2

SCP Secure copy is a way of transferring files through an SSH tunnel. 31

Soft embedded real-time system is a real-time system where timing constraints can be broken without dire consequences. Most often the system will take a performance hit when these constraints are broken. 15

SSH Secure shell is a network protocol which is used to create an encrypted connection between computers. 31

Threadmill Threadmill is the visualization tool described in this paper. 5

Bibliography

[1] L. Lamport, Time, clocks, and the ordering of events in a distributed system, Commun. ACM 21 (7) (1978) 558–565.
URL `http://doi.acm.org/10.1145/359545.359563`

[2] R. H. Netzer, B. P. Miller, Optimal tracing and replay for debugging message-passing parallel programs (1992).

[3] B. A. Myers, Taxonomies of Visual Programming and Program Visualization, 1990.

[4] T. Bemmerl, P. Braun, Visualization of message passing parallel programs, in: Parallel Processing: CONPAR 92—VAPP V, Springer, 1992, pp. 79–90.

[5] J. J. Van Wijk, The value of visualization, in: Visualization, 2005. VIS 05. IEEE, IEEE, 2005, pp. 79–86.

[6] J.-D. Fekete, J. J. Van Wijk, J. T. Stasko, C. North, The value of information visualization, in: Information visualization, Springer, 2008, pp. 1–18.

[7] J. Chassing de Kergommeaux, B. Stein, P. Bernard, Pajé, an interactive visualization tool for tuning multi-threaded parallel applications, Elsevier Science B.V., 2000.

[8] H. Kopetz, Real-time systems: design principles for distributed embedded applications, 2011.
URL `http://books.google.com/books?hl=en&lr=&id=oJZsvEawlAMC&oi=fnd&pg=PR5&dq=REAL-TIME+SYSTEMS+Design+principles+for+distributed+embedded+applications&ots=nJzPn4Rzfy&sig=z88SwRLfi0LyUAIyP90s1U1HnNohttp://books.google.com/books?hl=en&lr=&id=oJZsvEawlAMC&oi=fnd&pg=PR5&dq=Real-time+systems:+design+principles+for+distributed+embedded+applications&ots=nJzPp6Sw9A&sig=_9XnpYjvyZ18Pmf4L21agtY03WMhttp://books.google.com/books?hl=en&lr=&id=oJZsvEawlAMC&oi=fnd&pg=PR5&dq=Real-time+systems:+design+principles+f`

[9] POSIX introduction.
URL `https://computing.llnl.gov/tutorials/pthreads/`

[10] C. Ware, Information visualization: perception for design, Elsevier, 2012.

[11] A. Treisman, Preattentive processing in vision, Computer vision, graphics, and image processing 177 (1985) 156–177.
URL `http://www.sciencedirect.com/science/article/pii/S0734189X85800049`

[12] C. Artho, K. Havelund, S. Honiden, Visualization of Concurrent Program Executions, 31st Annual International Computer Software and Applications Conference - Vol. 2 - (COMPSAC 2007) (Compsac) (2007) 541–546.
URL `http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=4291176`

[13] B. P. Douglass, Real-time UML : developing efficient objects for embedded systems, Addison-Wesley, 1999.

[14] B. Selic, Turning clockwise: using UML in the real-time domain, Communications of the ACM 42 (10).
URL `http://dl.acm.org/citation.cfm?id=317675`

[15] V. O. Jeff Heer, Mike Bostock, D3.js data-driven documents (2014).
URL `http://d3js.org/`

[16] J. Rumbaugh, I. Jacobson, G. Booch, Unified Modeling Language Reference Manual, The (2Nd Edition), Pearson Higher Education, 2004.

[17] C. R. Ben Fry, Processing programming in a visual context (2014).
URL `https://www.processing.org/`

[18] BiD, Prefuse official webpage (2012).
URL `http://prefuse.org/`

[19] T. J. F. D. Team, JUNG java universal network/graph framework (2010).
URL `http://web.archive.org/web/20140601012027/http://jung.sourceforge.net`

[20] J. Ltd, JGraphX official webpage (2014).
URL `http://www.jgraph.com/`

[21] P. Eades, A heuristic for graph drawing, Congressus Numerantium, 1984.

[22] M. Garey, D. Johnson, Computers and Intractability: A guide to the Theory of NP-Completeness, W.H. Freeman and Company, 1979.

[23] E. Rodriguez-Tello, Memetic algorithms for the MinLA problem, 2006.

Appendix A

Tool features

Below is a more detailed list of the features in the tool.

1. **Filters**
   Free text filters follows the rules of regular expressions.
   (a) *size* – lets the user specify in what range message size is allowed in the visualization
   (b) *time* – lets the user specify in what range in time messages are allowed in the visualization
   (c) *message type* – lets the user only include messages of a certain type
   (d) *queue name* – lets the user only include messages sent from a certain queue
   (e) *cause-and-effect* – lets the user only include queues and messages that was caused by some message.
2. **Bookmarks**
   With a lot of events in the visualization it's easy to lose track of a specific event when going back and forth between different events. To avoid this issue the user can bookmark both queues and messages to easily jump back to a previous point in the visualization.
3. **Searching**
   With a basic knowledge of regular expressions the user can search for the litteral names or hexadecimal id's of either queues or message types.
4. **Detailed message inspection**
   The user can inspect a message and find out information such as the queue id of the sender, how many bytes is in the message data and so on. Most importantly the user can view the raw data of the message in hexadecimal, where the system also has the ability to convert the raw data into ASCII if recognized as valid characters. This is mostly useful if any of the fields contain string or character values otherwise the tool will recognize nonsense sentences.

5. **Real time scale**
   Many events can occur at the same recorded time because of the low resolution of the system clock. To make it easier to judge how much time has passed between events in the system the user can switch on *real time scale* to clamp events together with the result that events further away from each other in time will appear further away in the visualization.

6. **Exclude message data**
   If message data is simply not interesting or depending on system load, the system cannot spare the resources to capture the message data of all messages, the user can optionally choose to exclude it.

7. **Pre-filter on device**
   If it's the case that the message capture buffer is overflowing the user has the option to let some filtering occur on the device.

8. **Save and load logs locally**
   The tool allows the user to save or load the captured data as a file locally.