universität
wien

DISSERTATION

Titel der Dissertation

Improving reliability and performance of telecommunications systems by using autonomic, self-learning and self-adaptive systems

Verfasser

Mag. Michael Nussbaumer

angestrebter akademischer Grad

Doktor der technischen Wissenschaften (Dr. techn.)

Wien, 2011

Studienkennzahl lt. Studienblatt:	A 786 175
Dissertationsgebiet lt. Studienblatt:	Wirtschaftsinformatik
Betreuer:	Ao.Univ.Prof. Dr. Helmut Hlavacs

Michael Nussbaumer:
Improving reliability and performance of telecommunications systems by using autonomic, self-learning and self-adaptive systems

Department of Distributed and Multimedia Systems
Faculty of Computer Science
University of Vienna, Austria

This research has been supported by Softnet Austria, Kapsch CarrierCom

Abstract

My dissertation will be about autonomic, self-learning and self-adaptive systems. Usually an autonomic and self-learning system must be able to know its own status and the external operations, must be able to monitor system changes and must be able to self-adapt to them. Within this area my dissertation will present two case studies of autonomic and self-learning systems.

Improving reliability of multimedia communication:
While testing a commercial VoIP server it became obvious that the SIP protocol, used to initiate VoIP calls, is defined in a very open standard. That fact results in a great number of different SIP dialects, leading to the problem that some VoIP devices (hard and soft phones) may not be able to communicate with each other, even though they use the same protocol. Therefore an autonomic, self-learning SIP translator will be presented, that will decrease the rate of rejected SIP messages.

Automatic adaptation of system parameters to improve system performance:
The performance of a commercial system that collects data from various mobile devices is critical, because of the high amount of incoming data. Therefore performance tests will be initiated and automatically evaluated. Through self-learning techniques the system will self-adapt to the environment and the hardware on which the system is currently running, with the goal to improve the systems performance.

Acknowledgement

Many people deserve my deepest gratitude.

First of all I would like to thank my thesis advisor, Helmut Hlavacs, for his continued support throughout my work at the University. He always helped and encouraged me to pursue the goal of writing this thesis.

Also I would like to thank another advisor at the University, Karin Anna Hummel, for her support, especially during the first part of my work.

Then I would like to thank Christof Puntigam, Andrea Hess and Roman Weidlich, my colleagues in certain parts of the projects. It has been great working with them.

I developed the basic idea of my thesis within a cooperation of my university with the Softnet Austria Competence Network and the company Kapsch CarrierCom. The people there were willing to help with their hints and background knowledge: Werner Weissenbacher (Kapsch Mississippi), Gerhard Gruber (Kapsch Mississippi), Dieter Melnizky (Kapsch DataXtender), Nicolas Damour (Kapsch DataXtender), Wolfgang Scherer (Kapsch DataXtender), Karl-Heinz Driza (Kapsch DataXtender), Auguste Stastny (Kapsch Mississippi, DataXtender).

Most of all, I owe many thanks to my family and friends, who supported me throughout my whole professional career. None of my achievements would have been possible without their patience, understanding and advice. Thank you very much!

This work is especially dedicated to my parents, who had a terrible car accident while I was working on this thesis and are slowly recovering from it.

Contents

Chapter 1

Introduction

Especially in the last two decades telecommunication systems have become more and more important. [Wik09b] shows that 1997 just nearly 20 percent of people owned a cell phone, while today almost everybody owns a cell phone.

There are two aspects that are crucial for telecommunication systems today:

- Today everybody who uses telecommunication systems expects them to function at all times. Therefore telecommunication systems have to be highly available, but also have to guarantee that the system performs according to its specification. A reliable system should be able to correctly respond to proper requests.
- Due to the huge amount of requests sent minutes after midnight on January 1st, the New Year's Day traffic is a good indicator for the maximum workload of a telecommunication system. Thus, telecommunication systems must be built to handle New Year's Day traffic. Of course, there or other events (e.g.: human crises) which could lead to an extremely increased number of requests. If a telecommunication system is able to handle these kinds of workload bursts, it will very likely be able to handle all other workloads as well.

In my dissertation I worked on two different approaches to improve the reliability and the performance of telecommunication systems. In both cases I used autonomic, self-learning and self-adaptive systems or techniques to achieve an improvement.

Problem Definition I

The development of new Internet protocols can lead to a number of problems. Standards may be formulated in a very open way; software developers may only implement a subset of the protocol in early software versions, while other software developers are using the entire standard right from the start; there could be code faults and errors; and so forth. All these problems could lead to the fact that software implementing these new protocols may reject correct protocol messages resulting in a situation where devices using the same protocol may not be able to communicate with each other.

Problem Definition II

Server software that is constructed and designed to handle an enormous amount of load, might be overloaded at certain peak moments, which could lead to the fact that requests may be lost. Software that handles incoming data and splits its work into atomic tasks is often designed as a data-flow graph, where every incoming request passes a number of nodes where certain tasks are executed. State-of-the-art multi-core machines are used to improve the performance of the server software, by delegating certain tasks to individual cores. To improve the performance of the system even further, the tasks of over-utilized nodes can be allocated to idle cores. A number of cores can then in parallel handle a greater amount of requests in the same amount of time. Finding an algorithm on how to optimally delegate which task to which core can have enormous effects on this kind of multi-thread data-flow server software.

Improving voice over IP related systems is the main focus of my work, but finding solutions to the mentioned problems, could also lead to benefits in other information technology areas.

Operational Areas I

New ideas and technologies often result in the development of new Internet protocols. All these newly developed protocols share the same problems in the early stages of development, where communication between certain software products or devices fails. In my dissertation I focused on the Session Initiation Protocol (SIP), a text-based protocol that is used for creating, managing and

terminating multimedia sessions, but finding a solution to improve the reliability of SIP-based communication could also have an effect on other operational areas. The Universal Plug and Play Audio/Video (UPnP AV) protocol for instance, is used for communication between certain (in most cases) audio/video multimedia devices. These devices could use the same protocol, but often software developers are using their own communication protocols for setting up multimedia sessions or communications. The communication between two devices using a slightly different protocol, but with identical purposes and features might fail. The development of self-adaptive network protocols, which identify the purpose of certain messages and then alter incoming messages in a way to make two devices conform and therefore enable communication between them, could be a huge scientific progress in all information technology areas.

Operational Areas II

Optimizing server software is an important task. In my dissertation I worked with a multi-thread data-flow software used by a telecommunication system in the area of Voice over IP. Besides telephone servers, web servers and database servers, banks could use that kind of software for Internet online banking requests. For all these operational areas it is crucial that the software uses the hardware of a certain server in the best possible way to obtain the best possible performance.

Self-Learning Approach I

In my dissertation I try to solve the two mentioned problems by using autonomic, self-learning and self-adaptive techniques. The basic idea behind solving the problem of communication problems between two devices using the same protocol, is to identify the problematic parts of the messages used to set-up the sessions between them. These problematic message parts can then be altered to make a communication possible. The solution is to continuously learn which messages are rejected by the server, compare them to other rejected messages and by doing that, identify the aforementioned problematic message parts. This solution approach includes a way to classify incoming messages and a way to find a message which is as similar to the incoming, rejected message as possible and is known to be accepted by the server. By comparing

the two messages, differences can then be eliminated and the possibility that the incoming message will be rejected, should be reduced. Reaching this goal could be the first step to develop self-adaptive network protocols, which could have a benefit for all the previously mentioned operational areas.

Self-Adaptive Approach II

As mentioned before, multi-thread data-flow software is used in a great number of operational areas. Finding the optimal configuration for that kind of software for given hosts should be possible easily and quickly. The solution proposed in my dissertation presents an analytical approach, a measurement approach and a simulation approach. The idea behind all these approaches is to recursively add threads to over-utilized tasks. The multi-thread data-flow software uses multi-core servers to take advantage of the software structure by assigning tasks to threads which are executed on individual cores. The solution to finding the optimal number of threads per task is using an autonomic, self-adaptive system that is testing the used tasks and is finding out which task and thread is over-utilized. By recursively adding threads to over-utilized tasks, and thereby parallelizing the work, the system stabilizes itself, finding the optimal configuration of the system that can then handle the highest possible external request arrival rate.

1.1 Publications

The content of my dissertation was already published in four publications. These publications present the ideas and results of the work also discussed in this dissertation, but focus on specific parts of my work, whereas my dissertation presents my work in every detail. This section presents the abstracts of these publications.

Software implementing open standards like SIP evolves over time, and often during the first years of deployment, products are either immature or do not implement the whole standard but rather only a subset. As a result, standard compliant messages are sometimes wrongly rejected and communication fails. In this paper we describe a novel approach called Babel-SIP for increasing the rate of acceptance for SIP messages. Babel-SIP is a filter that can be put in

front of the actual SIP parser of a SIP proxy. By training a C4.5 decision tree, it gradually learns, which SIP messages are accepted by the parser, and which are not. The same tree can then be used for classifying incoming SIP messages. Those classified as not accepted can then be pro-actively changed into the most similar message that is known to be accepted from the past. By running experiments using a commercial SIP proxy, we demonstrate that Babel-SIP can drastically increase the message acceptance rate ([HAAM08]).

Software implementing open standards like SIP evolves over time, and often during the first years of deployment, products are either immature or do not implement the whole standard but rather only a subset. As a result, messages compliant to the standard are sometimes wrongly rejected and communication fails. In this paper we describe a novel approach called Babel-SIP for increasing the rate of acceptance for SIP messages.

Babel-SIP is a filter that is put in front of a SIP parser and analyzes incoming SIP messages. It gradually learns which messages are likely to be accepted by the parser, and which are not. Those classified as probably rejected are then adapted such that the probability for acceptance is increased. In a number of experiments we demonstrate that our filter is able to drastically increase the acceptance rate of problematic SIP REGISTER and INVITE messages. Additionally we show that our approach can be used to analyze the faulty behavior of a SIP parser by using the generated decision trees ([HNHH08]).

[HAAM08] presents Babel-SIP, a tool to improve the rate of accepted SIP messages and focuses on the results conducted with SIP REGISTER messages only, whereas [HNHH08] presents a more detailed description of Babel-SIP and the used decision trees and also includes experiments conducted with SIP INVITE messages.

This work presents a special class of a data flow oriented optimization tool that finds the optimal number of threads for multi-thread software. Threads are assumed to encapsulate concurrent executable key functionalities, are connected through finite capacity queues, and require certain hardware resources. We show how a combination of measurement and calculation, based on Queueing Theory, leads to an algorithm which recursively determines the best com-

bination of threads, i.e. the best configuration of the multi-thread software on a specific host. The algorithm proceeds on the directed graph of a queueing network which models this software. Optimization towards hardware consolidation, where CPU cores, memory, disk space and speed, and network bandwidth are constraints, but also towards throughput is described. Two experiments on different SUN machines verify our optimization approach ([WNH10]).

This work presents an optimization tool that finds the optimal number of threads for multi-thread data-flow software. Threads are assumed to encapsulate parallel executable key functionalities, are connected through finite capacity queues, and require certain hardware resources. We show how a combination of measurement and calculation, based on queueing theory, leads to an algorithm that recursively determines the best combination of threads, i.e. the best configuration of the multi-thread data-flow software on a given host. The algorithm proceeds on the directed graph of a queueing network that models this software. Experiments on different machines verify our optimization approach ([NH11]).

[WNH10] presents the idea of the calculation and measurement approach on how to find the optimal configuration for multi-thread data-flow software on a specific host, whereas [NH11] focuses more on validating the mentioned approaches by conducting extensive experiments.

1.2 Synopsis

The following section gives a detailed content overview of my dissertation to present a synopsis and an introduction to my work.

Chapter 2 presents the idea behind autonomic, self-learning and self-adaptive systems. It mainly presents work related to the topic of self-adaption and the idea behind creating these types of software.

Chapter 3 and Chapter 4 both deal with improving Voice over IP telecommunications systems, by using self-learning or self-adaption techniques. On the one hand, my work tries to make telecommunications systems more re-

liable, by finding a way to accept more correct incoming messages. On the other hand, my work tries to improve the performance of telecommunications systems by using the hardware of a specific host in the best possible way.

Chapter 3 presents my approach to create self-adaptive network protocols. The chapter deals with my work on a commercial VoIP server, which uses the SIP protocol to handle multimedia sessions. The SIP protocol uses text messages with a number of mandatory and optional headers to create, modify and terminate multimedia calls. Each header consists of certain mandatory and optional parameters. These parameter-combinations lead to different dialects that may cause problems for the VoIP server.

The chapter picks up these problems and uses self-learning techniques to improve the acceptance rate of the VoIP server, when handling incoming, RFC-correct, SIP messages.

At first, the basic idea and the motivation behind my work is presented (see Section 3.1), followed by work related to improving the reliability of VoIP systems and testing these systems (see Section 3.2).

Section 3.3 gives a background on Voice over IP and especially shows scenarios on how to set up a VoIP environment within companies.

Section 3.4 presents a background on the Session Initiation Protocol (SIP) and focuses mainly on different SIP dialects.

Different SIP dialects resulting in problems that two SIP devices may not be able to communicate with each other, even though they are using the same protocol, became the main focus of the first part of my work. Therefore, Section 3.4.1 presents different SIP messages and SIP dialects and also gives an overview of which headers and parameters are used by which VoIP hard and soft phone.

The goal of the first part of my dissertation was to make VoIP servers more reliable, therefore, it was necessary to test these systems. For every software developer, testing their product is crucial. For my work it was a necessary

step to get a feeling of the SIP protocol as well as understand different VoIP call scenarios. Of course, in the second step, testing a VoIP server with different SIP dialects showed which of these dialects were causing the VoIP server problems. Section 3.5 presents VoIP call scenarios and a way to test VoIP systems with a commercial test tool used to implement these call scenarios. Furthermore, the section presents two Java test tools I developed, which are used for further experiments.

Section 3.6 presents Babel-SIP, a self-learning tool using C4.5 decision trees to classify incoming SIP messages, extract the parameters of the incoming message, identify the problematic parameters of the incoming message and suggest to alter these problematic parameters. The section also presents an introduction and a qualitative analysis of decision trees, the extensive experiments conducted with Babel-SIP and discusses the results.

Like Chapter 3, Chapter 4 also deals with improving the work of VoIP-related telecommunications systems. The first part of my work tries to improve the availability and reliability of a VoIP server, while the second part of my work tries to improve the performance of software used for handling and storing telephone call data.

Telephone devices send Diameter tickets to the server software and the data is stored for (e.g.) the billing of calls. Of course, relative to the time of the day and the number of costumers, the amount of incoming tickets can be huge. The goal of my work is to automatically adapt the software in a way that it uses the hardware of the server in the best possible way.

Again, the motivation behind my work is presented (see Section 4.1), followed by related work and other approaches where a system tries to automatically find an optimal configuration for specific software on a given host (see Section 4.2).

The telecommunications software used in this second part of my work is structured like a queueing network. Therefore, Section 4.3 gives an overview on queueing networks and explains the difference between open and closed queue-

ing networks.

Section 4.4 then presents software using queueing network, which is used in my entire dissertation. The section focuses on the structure and the usage of the software and also presents the goals for the optimization of the software.

Section 4.5 presents the different approaches that can be used to find the optimal configuration for a given host. This work not only describes one single approach, but covers all bases by presenting an analytical approach, a measurement approach and a simulation approach. Finally, experiments will confirm results from the presented optimization approaches.

The developed ideas and tools presented in my thesis were developed in a working relationship with a global telecommunication company and are therefore tested with real software used in the telecommunication industry.

Finally, Chapter 5 presents a conclusion and sums up the ideas presented in my dissertation.

Chapter 2

Automated Systems and Related Work

Self-adaptive, self-learning and automated systems are (in most cases) used to handle the complexity of software system, which are getting bigger and more complex due to the development of new hardware technologies such as multi-core machines. Self-adaptive systems are furthermore used to minimize the involvement of human administrators. In my dissertation self-adaptive and self-learning techniques and systems are used to improve the reliability and performance of telecommunications systems without constant human intervention.

> *"Self-adaptive software evaluates its own behavior and changes behavior when the evaluation indicates that it is not accomplishing what the software is intended to do, or when better functionality or performance is possible."* (see [HT03])

The Defense Advanced Research Projects Agency (DARPA) provided this definition of self-adaptive software in 1997 and is quoted in numerous works, including [HT03].

In [KC03] IBM introduced their vision on automated systems. According to IBM a self-managing software system has to be *self-configuring*, *self-optimizing*, *self-healing* and *self-protecting*. Because software systems become bigger and

more interconnected, the challenge of the configuration and maintenance of these systems will be too complex for human administrators. Automated, self-managing systems should therefore free the system administrators from the details of maintenance and system operation. Another goal for automated systems is to guarantee the best possible system performance.

A big part of self-managing systems is monitoring. An autonomic element must be able to monitor external conditions and its own status.

Furthermore, two or more autonomic elements must be able to communicate with each other. Therefore such an autonomic element must *specify* input and output services; *locate* input services of other elements; *negotiate* with other elements to use services; *provision* their internal resources; *operate* with each other and *terminate* the connection.

[GVAGM08] takes IBMs approach on autonomic computing and adds the pervasive computing paradigm. [GVAGM08] takes each of the four attributes (*self-configuring*, *self-optimizing*, *self-healing* and *self-protecting*) described by IBM and adds the pervasive aspect to create smart spaces.

[KM07] defines a self-managed software architecture as one in which components automatically configure their interaction. This should be done to achieve the goals of the system, but by meeting an overall architectural specification. Within these specifications, the goal is to not just guarantee functional behavior, but self-managed systems also have to concentrate on performance, reliability and security issues. Also, such a self-managed system not only should meet the requirements, but the goal should be to optimize a system to a given specification.

[HL08] concentrates on security issues of autonomic systems. The component-based software paradigm is used to realize self-protected systems, that are running with almost no human administrator intervention.

In [HM08] the idea of autonomic computing is discussed. Like the autonomic nervous system in the human body takes care of unconscious reflexes that do

not require our attention (e.g.: bodily adjustments such as the size of the pupil), the goal of autonomic systems is to decrease the human involvement. Also, [HM08] discusses the Autonomic Computing Adoption Model Levels introduces by IBM. These levels basically classify software on a scale from *Basic* to *Autonomic*:

- *Level 1, Basic:* software is managed by administrators who perform changes manually.
- *Level 2, Managed:* through intelligent monitoring, the administrators work is reduced.
- *Level 3, Predictive:* system monitoring recognizes system behavior patterns and suggests changes.
- *Level 4, Adaptive:* software is more capable of not only suggest changes, but take action.
- *Level 5, Autonomic:* self-managed (see [KC03]) components that are driven by policies.

[ST09] states that autonomic software must be able to adapt itself triggered by either *"internal causes"* or *"external events"* (e.g.: changes within the entire body of the software or the entire external operating environment). Furthermore [ST09] states that

> *"such a system is required to monitor itself and its context, detect significant changes, decide how to react, and act to execute such decisions."*

[ST09] also defines that self-adaptive software must be able to do so *"at a reasonable cost and in a timely matter"*. Furthermore the following reasons for creating and using self-adaptive software are mentioned:

- managing the complexity of a software system,
- decreasing the costs for monitoring, managing and adapting software,
- guaranteeing the reliability of a software system in case of unexpected conditions and

- changing the software at runtime to achieve the desired goals.

According to [ST09], the requirements of self-adaptive software can be best explained by answering the questions *where*, *when*, *what*, *why*, *who* and *how*:

- By answering the question *"Where?"*, the system states which layer of the software needs to be changed.
- By answering the question *"When?"*, the system states when changing the software can be done or has to be done.
- By answering the question *"What?"*, the system states what parameters or components of the software have to be changed.
- By answering the question *"Why?"*, the system states the reasons why the software has to be adapted.
- By answering the question *"Who?"*, the system states if changes are done automatically or require human intervention.
- By answering the question *"How?"*, the system states the process on how to adapt the behavior of the software.

[NRS10] states that self-adaptive systems usually use FCLs (Feedback Control Loops) to achieve the mentioned requirement goals. [HGB10] notes that

> *"available techniques to describe the software architecture of such systems do not support to make the control loops explicit."*

Therefore, [HGB10] tries to extend UML modeling concepts to guarantee that control loops are treated as *"first class entities"* in such architectures.

[DDKM08] states that

> *"by observing its internal state and surrounding context continuously using feedback loops, an adaptive system is able to analyze its effectiveness by evaluating quality criteria and then self-tune to improve its operations."*

[DDKM08] uses the reflection mechanisms of programming languages, like Java, to monitor itself and the external environment.

As mentioned before, control loops are often used for achieving some sort of self-adaptive behavior. [SILM07] adds another control loop that manages the autonomic system, *"to improve the ability of an autonomic control loop to modify itself with changes in its environment"*.

[Pin08] introduces a three-step concept to automatically create a web performance simulation and also provide a trend analysis. In the first step performance parameters of a system are monitored and recorded. In the second step the monitored performance parameters are used to automatically create a simulation model. And in the third step, results of the first two components are used to predict possible scenarios and longterm trend analysis.
In [BPA+08] adaptive techniques based on machine learning are used to avoid distributed attacks and flooding. Every element in a network learns about the behavior of the network. A training set, including samples of attacks and abuses, is used to train each node within the network. Then, a Naive Bayes method is used to predict if an attack will happen. In a local model or classifier, information about the local traffic patterns of a node is collected and shared with all other nodes.

The basic goal of autonomic computing, self-managing or self-organizing (see [HdM08]) systems is to decrease the complexity and work for human administrators. With this as a goal, self-adaptive systems are becoming more and more important. [KRG+10] introduces a set of criteria, which can be used to evaluate the quality of the design of self-adaptive systems. Case study results helped to group the criteria into the categories *"methodological, architectural, intrinsic, and runtime evaluation"*.

As mentioned before, monitoring external and internal states and conditions is an important factor of self-adaptive software. Of course, the bigger and more complex the software and the external environment gets, the harder it becomes to monitor the entire system. [WSF10] discusses the role of a human administrator in monitoring such a system, where human commentary should help adapt the software. Decisions based on information about the software or the environment that cannot be monitored automatically, are then possible,

because of human input.

[DMSFR10] uses metadata to improve self-organising systems. The idea of the proposed approach is that

> *"at run-time, both the components and the run-time infrastructure exploit metadata to support decision-making and adaptation based on the dynamic enforcement of instantiated policies."*

Policies are also used in [KKSJ10] to *"control and adapt the system behavior"*.

This Chapter presented the basic idea of self-adaptive, self-learning and automated systems. The most important part for my dissertation is that self-adaptive and self-learning systems should be able to know the system that they are used for (e.g.: software and hardware) and the environment (e.g.: incoming data). To improve the reliability and performance of telecommunications systems, the developed tools have to adapt themselves or the incoming data automatically without constant human intervention.

In my dissertation I use self-learning and self-adaptive techniques and systems in the area of telecommunications systems to improve these software systems without increasing the cost for administrative manpower.

Work that is even closer related to my dissertation topic will be presented in Section 3.2 and Section 4.2.

Chapter 3

Self-Adaptive Network Protocols

Network protocols, like the SIP protocol, used to create Voice over IP sessions, are defined in published open standards. Software implementing these standards often evolve over time and not necessarily handle all possible features. Therefore new products using these protocols are often immature at the beginning or do not implement the whole standard but rather only a subset.
In the recent years voice over IP became very popular. In the early stages, products using the SIP protocol to create and manage VoIP sessions showed the mentioned symptoms and problems. As a result, SIP messages compliant to the standard are sometimes wrongly rejected by VoIP systems and communication fails.

This chapter introduces a novel approach called Babel-SIP for increasing the rate of acceptance for SIP messages. Section 3.1 deals with the basic idea and the motivation behind my work. Section 3.2 describes related work. Section 3.3 provides an overview of voice over IP (VoIP) in general. Section 3.4 specializes on the most popular used protocol with VoIP, the Session Initiation Protocol (SIP). Section 3.5 presents my work while testing VoIP systems. And finally Section 3.6 shifts the focus to autonomic and self learning systems and presents an approach on how to automatically and autonomically adapt SIP messages.

3.1 Basic Idea and Motivation

In recent years voice over IP (VoIP) became very popular. Telecommunication providers are now offering products using VoIP to their costumers. Companies are changing from their old telephone network to a VoIP network to reduce costs. Through that hype more and more VoIP hard and soft phones have been produced and commercial and open source VoIP systems have been developed.

The initial idea behind my work was to test VoIP systems and additionally to improve their performance. Through researches it became obvious, that VoIP system developers often demand special types of VoIP phones to be used with their product. The main reason is that SIP, which is defined in RFC 3261 [RSC+02], is a very open protocol and leaves room for different interpretations. That fact leads to the problem that VoIP phones are sending messages in their own SIP dialect (see Section 3.4.1). For instance, for a commercial VoIP server, during recent versions, several hard and soft phones were known that would not be able to register themselves to the proxy, due to yet RFC conform, but still problematic SIP messages.

During my work I found that valid SIP messages may not work with certain VoIP servers. Therefore as my work emerged, the focus and motivation changed from simply testing the VoIP system, to dealing with different SIP dialects. The main goal was to develop independent software that would improve this situation, making it possible for a wider range of VoIP phones to function with different VoIP servers.

3.2 Related Work

In the application area of VoIP and SIP, authors both investigate traffic behavior and failures in particular software implementations. In [KZRN07] the authors describe the need and their solution for profiling SIP-based VoIP traffic (protocol behavior) to automatically detect anomalies. They demonstrate that SIP traffic can be modeled well with their profiling and that anomalies could be detected.

In [APWW07] it is argued that based on the SIP specification, a formal testing

of an open source and a commercial SIP proxy leads to errors with the SIP registrar. Both findings are encouraging to propose a method for not only detecting incompatibilities and testing SIP proxies, but further to provide a solution for messages rejected due to slightly different interpretations of the standard or software faults [HNHH08].

In [ASF07] a stateful fuzzer was used to test the SIP compatibility of User Agents and proxies by sending different (faulty) messages both in terms of syntax and in terms of protocol behavior. The idea here is only to find weaknesses in the parser implementation, without trying to adapt messages online.

In [AWW+07a] and [AWW07b], incoming and outgoing SIP messages of a proxy are analyzed by an in-kernel Linux classification engine. Hereby, a rule-based approach is proposed, where the rules are pre-defined (static) [HNHH08].

In [RN09] a self-healing approach is introduced to recognize and restart failed SIP servers. [RN09] uses the Windows filtering Platform (WFP) to monitor outgoing SIP traffic on a VoIP system. When the monitor detects no outgoing traffic for a pre-defined time period it injects a SIP INVITE message without a *Call-ID* header. If the VoIP system works it should respond with a *400 Bad Request* message. If there is no such respond from the VoIP system, the monitor will declare the system as crashed. In the second step the VoIP server is restarted. During that time all incoming SIP requests will be stored and handled after the restart.

Decision trees, and in particular the used C4.5 tree, allow classifying arbitrary entities or objects that can be used, for instance for computer vision (applied to robotics) [WR05] or characterization of computer resource usage [HM04]. In [WR05] decision trees were used for learning about the visual environment that was modeled in terms of simple and complex attributes and successfully implemented for improving recognition possibilities of Sony Aibo robots (e.g., the surface area or angles). Decision trees that use further linear regression have been proposed for the characterization of computer resource usage in [HM04]. Parameters like the CPU, I/O, and memory were used as attributes and the classification tree was finally used to successfully determine

anomalies of the system's parameters. The authors claim that the configuration of the learning process was time consuming (e.g., finding the trade-off between accurate history knowledge and time-consuming training).
In [ABR04] intrusion detection was introduced based on a combination of pattern matching and decision tree-based protocol analysis. This tree-based approach allows adapting to new attack types and forms while the traditional patterns are integrated into the tree and benefit from refinement of crucial parameters [HNHH08].

[Sub09] presents an architecture (KitCAT) for testing converged applications. The system under test is a Web/VoIP server and includes a web portal, a VoIP server and a media server. The goal is to test converged applications which include multiple sources, multiple user interfaces and use a number of different protocols for communication purposes. Therefore the proposed system not only tests SIP-based traffic, but also uses HtmlUnit to test the web-based applications within the SUT.

[FNKC07] tries to improve the reliability of SIP telecommunications systems by using a combination of a centralized infrastructure and a Peer-to-Peer (P2P) approach, where besides using a centralized VoIP server, user agents form a P2P network. By using a P2P approach, the proposed system CoSIP (Cooperative SIP), can replace VoIP server in case they are not responding and thereby improving the systems reliability.

3.3 Background: Voice over IP

Phone service over the Internet was first introduced in 1995 by an Israeli company called VocalTec. Their *'Internet Phone'* used a half-duplex communication system that only allowed alternate talking. The real breakthrough for VoIP came 2004 with Skype [Lim09]. Skype uses a proprietary protocol for Internet telephone calls. The reasons for using VoIP are mainly lower costs and increased functionality. For example, VoIP provides features that would be very difficult or even impossible to provide with traditional phone services. Furthermore VoIP calls are automatically routed to the current position within a network [VI09a].

The most important part of a VoIP system is the proxy. The proxy handles the routing of Internet telephone calls and is responsible for the call setup, call management and call tear down. Also the proxy provides advanced features like conference calls, presence indication and the possibility to forward or redirect calls.

Most of the times VoIP developers use media servers for managing voice recordings, voice messages and early media functions.

A VoIP system usually also contains a gateway that connects the VoIP telephone network with the traditional phone network. The gateway can be part of the own network architecture, but is usually provided by a telephone company.

To use a VoIP system with a traditional phone network an ADC (Analog-to-Digital-Converter) must also be used.

VoIP end users then use either VoIP hard or soft phones for communication. VoIP hard phones are digital stand-alone phones with, most of the time, two Ethernet ports (LAN, PC), so they can be installed between the Local Area Network and the user's computer. A VoIP soft phone is installed on a computer and needs a microphone and a headset or speakers. A soft phone is typically cheaper than a hard phone (e.g.: open source soft phones) and usually provides advanced features and services like video communication or online messaging.

Table 3.1 presents a list of commonly used VoIP codecs. Codecs are used to convert an analog voice signal to a digitally encoded version [VI09a]. Usually each VoIP soft or hard phone supports several codecs.

Codec	**kbit/s**
G.711	64
G.722	48/56/64
G.723	5.3/6.3
G.726	16/24/32/40
G.728	16
G.729	8
GSM	13
iLBC	15
Speex	2.15 - 44.2

Table 3.1: VoIP codecs.

The mentioned codecs usually differ in bandwidth requirements (e.g.: voice payload size, packets per second) and the voice quality (e.g.: codec bit rate), to achieve a balance between the quality of a call and the bandwidth efficiency.

3.3.1 Setup scenarios

Usually there are two possibilities for companies to set up a VoIP solution: Either a phone provider offers a VoIP solution for a company and only the end devices (hard and soft phones) are set up within the company (see Figure 3.1), or the company sets up the entire VoIP system within the company. In the latter case the VoIP phones, the proxy and the media server are located within the company (see Figure 3.2).

Provider Solution

Figure 3.1 shows a VoIP solution offered by a phone provider. Within a company hard and soft phones are located on the desks of the employees. These phones are then connected through a switch. Often this system is then protected by a firewall and connected to the provider backbone through a modem. The VoIP system within the provider backbone usually consists of a VoIP server, handling the VoIP call scenarios, and a media server, for e.g. storing voice messages. The system is then connected to the PSTN network through a gateway.

Definition: The Public Switched Telephone Network (PSTN) is the name of the connected global telephone network. It was usually invented as an analog all fixed-line telephone system, but does now include a digital system as well as mobile devices.

Definition: A gateway connects internet devices and is used for the communication between devices using different network protocols.

Of course the configuration and maintenance of the VoIP system itself is handled by the provider and most of the time it means that users are able to talk to other costumers within the provider's backbone with no additional costs. On the other hand such a solution mostly comes with monthly costs.
Advantages of a VoIP provider solution are

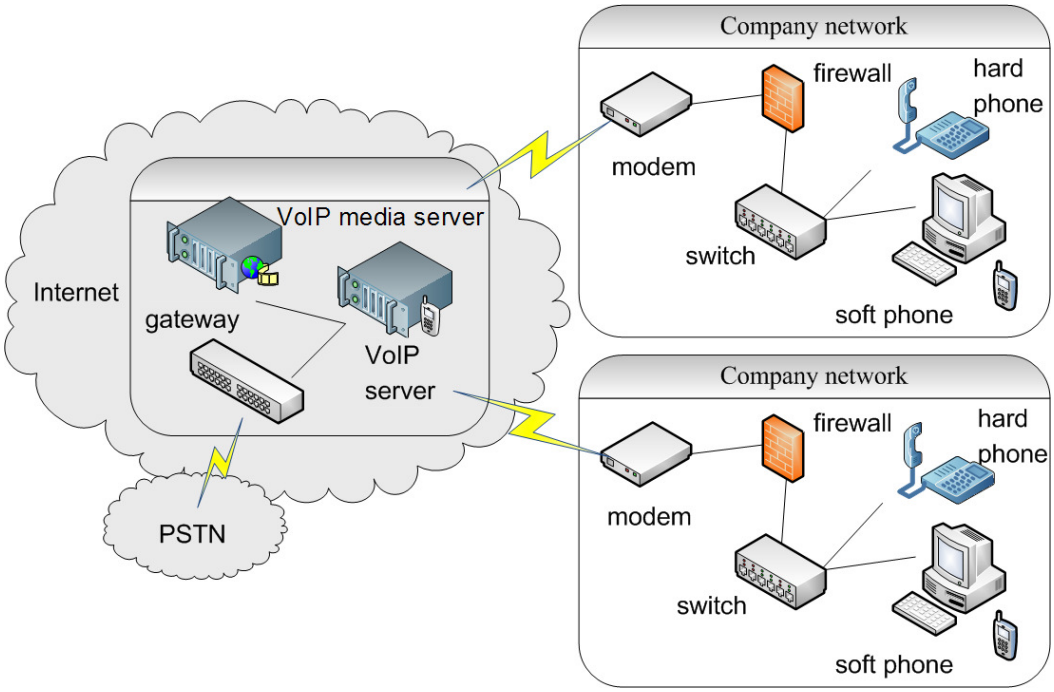


Figure 3.1: VoIP system, provider solution.

- no VoIP knowledge needed by the company,
- configuration of the VoIP system is done by the provider,
- maintenance of the VoIP system is done by the provider, and
- no additional costs for communication between companies within the provider backbone.

Possible disadvantages include

- expanding has to be done by the provider,
- the company has to contact the provider with every problem/request, or
- monthly costs.

Company Solution

Figure 3.2 shows an in-house VoIP solution. Usually this means that VoIP hard and soft phones, analog phones and fax, and the company server are all connected through a switch.

The VoIP server and the media server are also set up within the company and connected through the switch. In most cases the switch is protected through a firewall and connected to the Internet through a modem.
Within the company an ISDN system handles incoming calls from the PSTN network and forwards them to an analog phone or an ISDN phone. The VoIP system handles PSTN calls that are intended for VoIP recipients.

Definition: The Integrated Services Digital Network (ISDN) is a standard for an international telecommunications network.

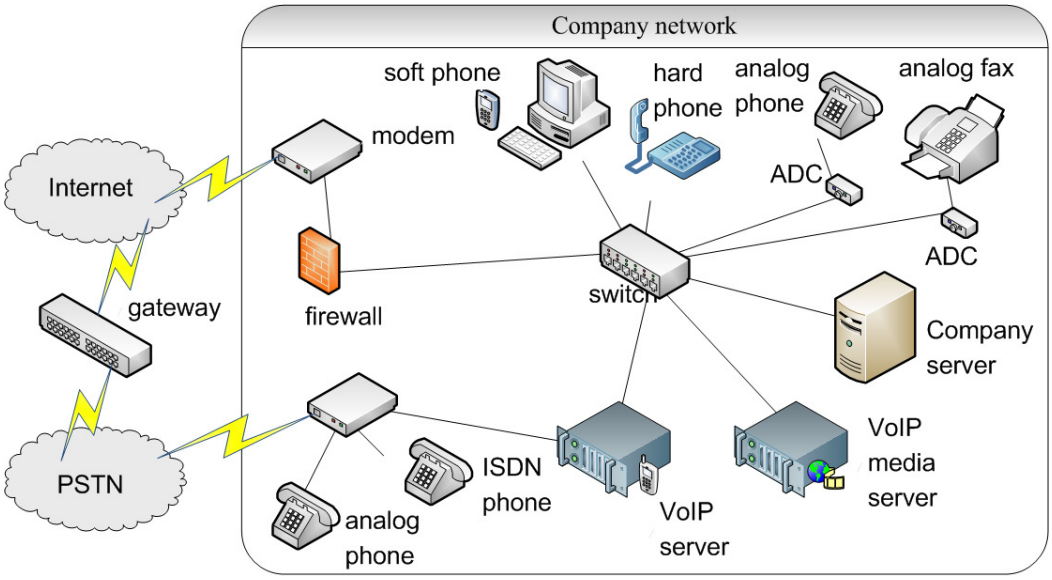


Figure 3.2: VoIP system, company solution.

Of course purchasing and setting up an entire VoIP system may be very expensive. And seeing as the company is responsible for configuration and maintenance, an administrator with knowledge on VoIP is necessary. On the other hand the company can decide on which hard and soft phones to use and on when and how to expand the VoIP system.
Advantages of a VoIP company solution are

- managing the VoIP system is done within the company,
- the VoIP system is easy to expand, and
- the company can specify hard and software components.

Disadvantages include

- an administrator with knowledge on VoIP is necessary,
- costs for setting up an entire VoIP system, or
- configuration and maintenance can be expensive and time-consuming.

The following section shifts the focus to the Session Initiation Protocol, which is used to set up voice over IP sessions.

3.4 Background: The Session Initiation Protocol

The Session Initiation Protocol (SIP) has become the main protocol when dealing with VoIP. SIP basically works as *"a signaling protocol and is used to create, modify and terminate sessions"* (see [RSC+02]). A session would mainly be an Internet telephone call, but it can also be a multimedia distribution, a multimedia conference and so on.

As mentioned in [RSC+02] there are five facets of establishing and terminating sessions that SIP deals with. First: the location of an end point (*user location*). Second: determine if the called party is willing to engage in a started session (*user availability*). Third: determine the media and media parameters that should be used within a session (*user capabilities*). Fourth: setting up the session and establishing the session parameters at both parties (*session setup*). Fifth: modifying established session, invoking services and terminating sessions (*session management*).

Usually within Internet telephone calls the Session Initiation Protocol (SIP) is used to establish, modify and terminate sessions. Furthermore the Session Description Protocol (SDP) is used to describe the actual session. Between two or more parties it is used to agree on transport protocols and addresses as well as multimedia codecs [HJP06]. Finally the Real-Time Transport Protocol (RTP) is used for continuous transmission of, usually audio, data streams [SCFJ03].

With SIP being a text-based protocol, SIP messages can either be requests (messages from a client to a server) or responses (messages from a server to a

client). SIP requests start with a request line, while SIP responses start with a status line. The following shows a typical SIP request line which starts with the method used (REGISTER, INVITE, ACK, CANCEL, BYE, OPTIONS), followed by a request URI and the SIP version.

```
INVITE sip:1001@myDomain SIP/2.0
```

The following shows a typical SIP response line. Such a response line starts with the SIP version, followed by a status code and a reason phrase.

```
SIP/2.0 200 OK
```

The status code is a 3-digit number, where the first digit defines the class of response. There are six possible response classes [RSC+02]:

- *1xx:* Provisional Response. The request was received and the VoIP server is continuing to process the request. Example: 100 Trying.
- *2xx:* Success. The request was successfully received and accepted. Example: 200 OK.
- *3xx:* Redirection. Further action is needed to complete the request. Example: 302 Moved Temporarily.
- *4xx:* Client Error. The request contains bad syntax. Example: 407 Proxy Authentication Required.
- *5xx:* Server Error. A valid request could not be completed. Example: 503 Service Unavailable.
- *6xx:* Global Failure. Example: 600 Busy Everywhere.

An extended list of SIP response codes can be found at [RSC+02] and [Wik09a].

Usually the first thing a user agent has to do is to register at the VoIP server. Most VoIP systems are using a registrar and a location server for handling the *user location*.
A user agent sends REGISTER messages, containing the user's ID and the IP address of the user agent, to a local registrar server. The registrar server then

updates the received locality information in the location server (see Figure 3.3). From this time the user can be reached at the sent address [HAAM08]. The 200 OK response of the VoIP server signalizes that the register process was successful.

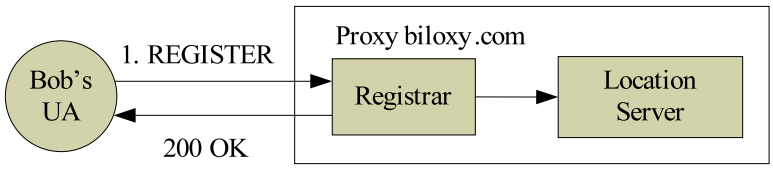


Figure 3.3: SIP registration of a user agent.

After a user agent is successfully registered at a local server it can send or receive call invitations. Handling calls is then the task of a so called proxy server. A SIP call scenario starts by a user (*Alice*) picking up either a VoIP hard or soft phone and calling another user by dialing the user's SIP URI (like *bob@biloxy.com*). The user agent (*Alice*) then sends a SIP INVITE message to the proxy it is registered at (*atlanta.com*). The local proxy then forwards the INVITE message to the proxy server of the called party (*biloxy.com*), which asks the local location server where to find the called party's user agent. If Bob's user agent has already been registered, the proxy forwards the INVITE message to Bob's user agent (see Figures 3.4 and 3.5) [HNHH08].

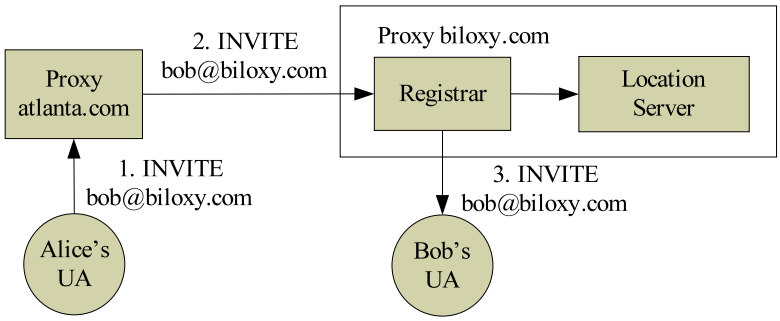


Figure 3.4: SIP call setup.

The initial SIP INVITE message also contains a Session Description Protocol (SDP) part. Table 3.2 shows a typical SIP INVITE message, including the in-

formation regarding the session description. After the initial INVITE message has been delivered to the called user agent, that user agent usually responds with a 180 Ringing message. At the moment when the called party has accepted the incoming call and picks up the phone, the user agent sends a 200 OK message back to the caller's user agent. After the caller's user agent has acknowledged the incoming 200 OK message by sending an acknowledgement (ACK) message, the call is successfully set up.

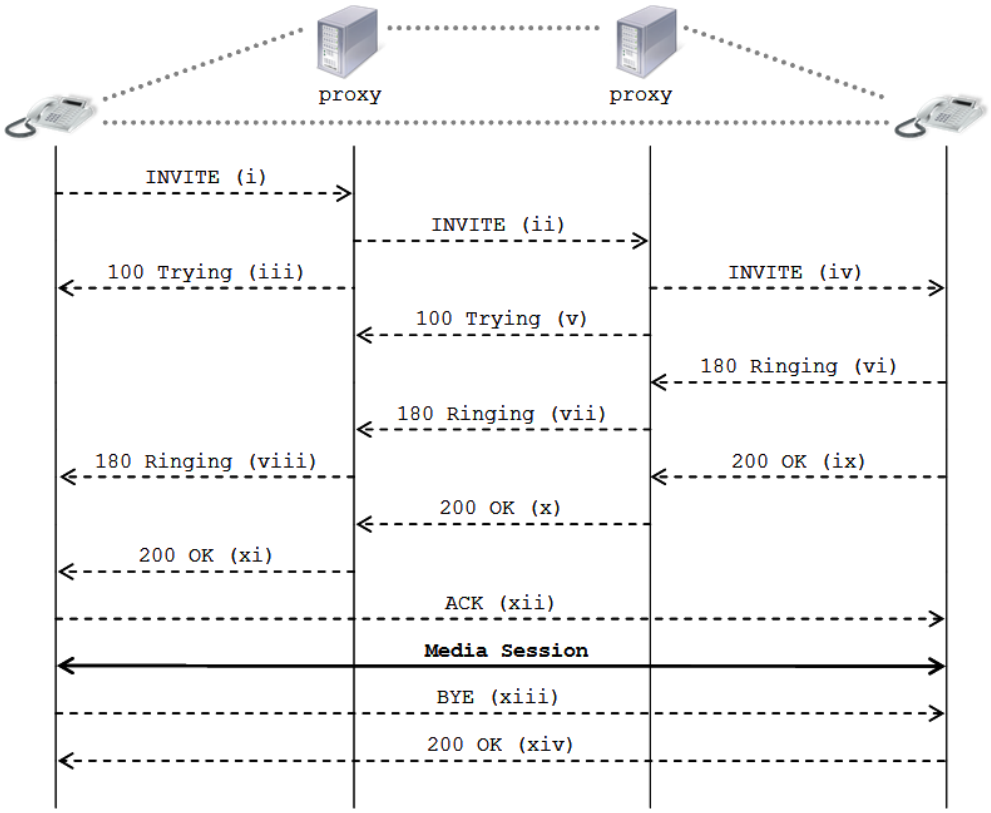


Figure 3.5: SIP trapezoid.

From that moment on the audio stream is transmitted via the Real-Time Transport Protocol (RTP) directly between the users. A basic call usually ends by a user hanging up the phone. At that moment the user agent sends a BYE message to the second party, which has to acknowledge the incoming BYE message by answering with a 200 OK message (see Figure 3.5). Table 3.2 shows an example of a basic SIP INVITE message [RSC+02].

Of course commercial VoIP systems offer a lot more features then just basic calls between two parties.

```
INVITE sip:bob@biloxy.com SIP/2.0
Via:  SIP/2.0/UDP IPAddress:Port
From:  <sip:a lice@atlanta.com>
To:  <sip:bob@biloxy.com>
Call-ID: 972238000055610
CSeq:  1 INVITE
Max-Forwards:  70
Contact:  <sip:alice@IPAddress:Port>
Content-Type:  application/sdp
Content-Length:  185

v=0
o=root 431065466 431065466 IN IP4 IPAddress
s=call
c=IN IP4 IPAddress
t=0 0
m=audio 40000 RTP/AVP 8 101
a=rtpmap:8 pcma/8000
```

Table 3.2: SIP basic INVITE message.

The following section presents different SIP dialects and Section 3.5 describes common call scenarios.

3.4.1 SIP dialects

One strong motivation behind my work was that some VoIP servers only allow certain SIP hard and soft phones. The reason is that the VoIP servers may not work with other VoIP devices because they send different, problematic messages. In most cases these problematic messages are correct SIP messages according to RFC 3261, which nevertheless may cause problems. One of the reasons is that RFC 3261 defines a very open standard and leaves room for different interpretations. Another reason is that VoIP servers may not include all SIP features in early stages of the development phase.

RFC 3261 defines that only certain header fields are necessary for a SIP message to work properly, but of course there are many optional header fields that can be used by a VoIP phone within a SIP message as well [HNHH08].
As seen in Table 3.3 a SIP REGISTER message according to RFC 3261 has to contain a request line and the headers To, From, Call-ID and CSeq.

```
REGISTER sip:Domain SIP/2.0
To:  <sip:UserID@Domain>
From:  <sip:UserID@Domain>
Call-ID: NDYzYzMwNjJhMDRjYTFj
CSeq:  1 REGISTER
```

Table 3.3: SIP REGISTER message, RFC minimum.

Additional to the request line and the headers To, From, Call-ID and CSeq, SIP requests like the INVITE message, according to RFC 3261, have to contain the headers Via and Max-Forwards, as seen in Table 3.4.
The fact that SIP devices apparently send different messages made it necessary to examine some VoIP phones to get a sense what messages where sent and what headers and attributes were used.

```
INVITE sip:CalleeUserID@CalleeDomain SIP/2.0
To:  <sip:CalleeUserID@CalleeDomain>
From:  <sip:CallerUserID@CallerDomain>
Via:  SIP/2.0/UDP IPAdress:Port
Call-ID: NDYzYzMwNjJhMDRjYTFj
CSeq:  1 INVITE
Max-Forwards:  70
```

Table 3.4: SIP INVITE message, RFC minimum.

It was important to get a set of test phones that included VoIP phones used by the commercial VoIP server I worked with, but also get other popular and often used VoIP phones. After my research I gathered nine VoIP hard phones. Also I chose five popular VoIP soft phones and added them to my list.
Table 3.5 shows a list of the VoIP hard and soft phones that I used to identify different SIP dialects. The first step was to set up a test environment. I therefore connected each phone to the commercial VoIP server. In the next step I monitored the messages sent from each VoIP phone during the REGISTER process. Furthermore to see the differences within INVITE messages I initiated various basic call scenarios. To get a full overview I made calls with every phone being the calling party and the called party once and again monitored the sent SIP messages.

An extended list of VoIP hard and soft phones can be found at [VI09b].

Hard Phones	Snom 300 Polycom SoundPoint IP 330 Linksys SPA IP Phone SPA 941 DLink VoIP IP-Phone DPH-120S Thomson ST2030 Allnet 7950 Grandstream GXP2000 Enterprise IP Phone Elmeg IP 290 Siemens
Soft Phones	X-Lite PortSIP BOL Express Talk 3CX

Table 3.5: Tested VoIP hard and soft phones.

Figure 3.6 shows the basic testing environment used to monitor the SIP traffic of the tested VoIP hard and soft phones. The VoIP server consists of the VoIP proxy, the location server and the media server. SIP hard phones, as well as the soft phones installed on a personal computer, are connected to the VoIP server via a switch.

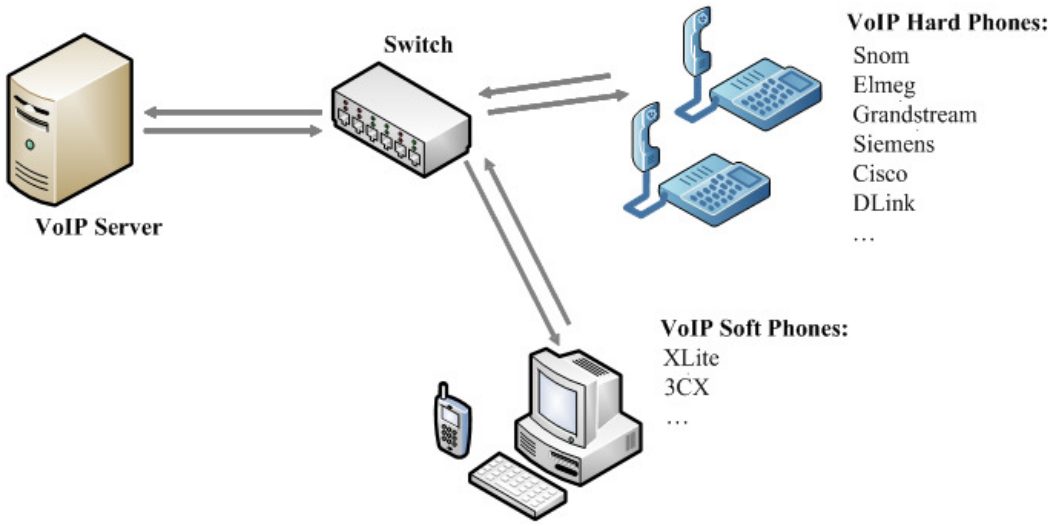


Figure 3.6: VoIP test environment, monitoring the SIP traffic.

After monitoring the traffic and gathering the various SIP messages from the different VoIP phones it became obvious that every phone sends different mes-

sages.
Table 3.6 shows the SIP REGISTER message of the DLink VoIP IP-Phone. The differences between the RFC minimum (see Table 3.3) and the message sent from the DLink VoIP phone (see Table 3.6) are apparent.

```
REGISTER sip:Domain:Port SIP/2.0
To:  <sip:UserID@Domain:Port>
From:  <sip:UserID@Domain:Port>;tag=1f54431a
Via:  SIP/2.0/UDP IPAdress:Port;branch=z9hG4bK-
Call-ID: NRjYTFjMmI5NTE3NDRkOGFkYzY3OTc.
CSeq:  1 REGISTER
Contact:  <sip:UserID@IPAdress:Port>
Max-Forwards:  70
Expires:  3600
User-Agent:  DLink
Content-Length:  0
```

Table 3.6: DLink VoIP IP-Phone DPH-120S REGISTER message.

Additional to the request line and the headers To, From, Call-ID and CSeq, the DPH-120S also uses the headers Via, Contact, Max-Forwards, Expires, User-Agent and Content-Length within its REGISTER message. The DPH-120S also inserts the used *Port* into the request line and the To and From headers.

To compare the different SIP dialects, Table 3.7 shows the SIP REGISTER message sent by the Elmeg VoIP phone. Additional to the headers defined in RFC 3261 (To, From, Call-ID and CSeq) and the headers used by the DLink VoIP phone (Via, Contact, Max-Forwards, Expires, User-Agent and Content-Length), the Elmeg IP 290 VoIP phone uses the headers Allow-Events, X-Real-IP, WWW-Contact and Supported in its REGISTER message. Furthermore the Elmeg IP 290 uses the *rport* parameter within the Via header and inserts the *Display Name* into the headers To and From. Also the Elmeg IP 290 uses a greater set of parameters within the Contact header.

The mentioned SIP REGISTER messages show that there are vast differences between the tested phones. Every tested phone uses a different set of SIP headers, so the open SIP standard could lead to the problem that some VoIP phones are unable to register at a VoIP server. Of course it is a big problem

```
REGISTER sip:Domain SIP/2.0
Via:  SIP/2.0/UDP IPAdress:Port;branch=z9hG4bK-;rport
From:  "Name" <sip:UserID@Domain>;tag=1f54431a
To:  "Name" <sip:UserID@Domain>
Call-ID: NDYzYzMwNjJhMDRjYTFjMmI5NTE3NDRkOGFkYzY3OTc.
CSeq:  1 REGISTER
Max-Forwards:  70
Contact:  <sip:UserID@IPAdress:Port;line=abUserID>;
  q=1.0;audio;mobility='fixed';duplex='full';
  +sip.instance='<urn:uuid:767e30a9-0c85-4238-ab02>';
  methods='INVITE,ACK,CANCEL,BYE,NOTIFY,REFER,OPTIONS'
User-Agent:  Elmeg
Allow-Events:  dialog
X-Real-IP: ServerIP
WWW-Contact:  <http://IPAdress:Port>
WWW-Contact:  <https://IPAdress:Port>
Expires:  3600
Content-Length:  0
Supported:  gruu
```

Table 3.7: Elmeg IP 290 REGISTER message.

if a phone is unable to register itself, because that means that the phone will not work with the particular VoIP server. Although the experiments showed that the register process has not been a problem for most of the tested phones, the different SIP dialects would probably be more critical within more complex call scenarios. Therefore it was necessary to examine the different SIP INVITE messages.
Table 3.8 shows an INVITE message created by the VoIP soft phone XLite. It can be noted that within SIP REGISTER messages as well as within SIP INVITE messages real VoIP phones use more than the mandatory SIP headers prescribed by RFC 3261. SIP requests, like INVITE messages, have to contain a request line and the SIP headers To, From, Via, Call-ID, CSeq and Max-Forwards according to RFC 3261.

Table 3.8 shows that INVITE messages created by the soft phone XLite additionally contain the headers Contact, User-Agent, Allow, Content-Type and Content-Length. Also the From header includes a *From-tag* and the *Via header* includes the Via-branch and the *rport* parameter.

Table 3.9 shows a SIP INVITE message created by the Thomson ST2030 VoIP

```
INVITE sip:CalleeUserID@Domain SIP/2.0
To:  <sip:CalleeUserID@Domain>
From:  <sip:CallerUserID@Domain>;tag=1f54431a
Via:  SIP/2.0/UDP SenderIP:Port;branch=z9hG4bK;rport
Call-ID: NDYzYz.
CSeq:  1 INVITE
Contact:  <sip:CallerUserID@SenderIP:Port>
Max-Forwards:  70
User-Agent:  XLite
Allow:  INVITE,ACK,CANCEL,BYE,REFER,OPTIONS,NOTIFY,
        SUBSCRIBE,MESSAGE,INFO
Content-Type:  application/sdp
Content-Length:  303

v=0
o=- 3 2 IN IP4 SenderIP
s=CounterPath X-Lite 3.0
c=IN IP4 SenderIP
t=0 0
m=audio 5062 RTP/AVP 119 6 0 8 102 3 5 101
a=alt:1 1 :  nnXz/c07 dA+YPE1r SenderIP 5062
a=fmtp:101 0-15
a=rtpmap:119 BV32-FEC/16000
a=rtpmap:102 L16/16000
a=rtpmap:101 telephone-event/8000
a=sendrecv
```

Table 3.8: XLite INVITE message.

phone. In addition to the headers used by the XLite soft phone, the ST2030 uses the headers Supported and Session-Expires. Also the ST2030 inserts the *Port* and the *user=phone* parameter in the request line and the headers To, From and Contact.

Table 3.10 shows an INVITE message created by the VoIP phone Snom 300 that adds the headers P-Key-Flags, Accept, Allow-Events and Min-SE to the already mentioned SIP headers.

Analyzing the message flow of all fourteen tested VoIP hard and soft phones, in addition to a request line and the headers To, From, Via, Call-ID, CSeq and Max-Forwards (according to RFC 3261), the phones also used the headers Contact, P-Key-Flags, User-Agent, Accept, Allow-Events, Allow, Supported, Session-Expires, Min-SE, Content-Type, Content-Length, Proxy-Authorization, Authorization, Event, X-Real-IP, WWW-Contact, Expires, Route, Record-

```
INVITE sip:CalleeUserID@Domain:Port;user=phone SIP/2.0
To:  <sip:CalleeUserID@Domain:Port;user=phone>
From:  <sip:CallerUserID@Domain:Port;user=phone>;tag=1f54431a
Via:  SIP/2.0/UDP SenderIP:Port;branch=z9hG4bK
Call-ID: NDYzYz.
CSeq:  1 INVITE
Contact:  <sip:CallerUserID@SenderIP:Port;user=phone>
Max-Forwards:  70
User-Agent:  Thomson
Allow:  INVITE,ACK,CANCEL,BYE,REFER,OPTIONS,NOTIFY,SUBSCRIBE,
        PRACK,UPDATE,INFO,REGISTER
Supported:  timer, replaces
Session-Expires:  1800
Content-Type:  application/sdp
Content-Length:  268

v=0
o=CallerUserID 431065466 431065466 IN IP4 SenderIP
s=-
c=IN IP4 SenderIP
t=0 0
m=audio 41000 RTP/AVP 8 0 18 4 97
a=rtpmap:8 PCMA/8000
a=rtpmap:0 PCMU/8000
a=rtpmap:18 G729/8000
a=rtpmap:4 G723/8000
a=rtpmap:97 telephone-event/8000
a=fmtp:97 0-15
a=sendrecv
```

Table 3.9: Thomson ST2030 INVITE message.

Route, Require, RSeq and Server.

Of course there are further additional headers that can be used by SIP VoIP phones. A full list of SIP headers and the according parameters can be found at [Aut09].

Tables 3.11 and 3.12 show a complete list of all the headers, header parameters and attributes used by each tested phone. Table 3.11 provides an overview of the different SIP dialects used within the SIP REGISTER messages. Table 3.12 concentrates on the SIP INVITE messages and presents the content of the different messages.
The following list will describe and explain most of the headers used by the

tested hard and soft phones (see [RSC+02]).

```
INVITE sip:CalleeUserID@Domain;user=phone SIP/2.0
To:  <sip:CalleeUserID@Domain;user=phone>
From:  <sip:CallerUserID@Domain>;tag=1f54431a
Via:  SIP/2.0/UDP SenderIP:Port;branch=z9hG4bK;rport
Call-ID: NDYzYz.
CSeq:  1 INVITE
Contact:  <sip:CallerUserID@SenderIP:5060;line=ab1001>;
          flow-id=1
Max-Forwards:  70
P-Key-Flags:  keys='3'
User-Agent:  Snom
Accept:  application/sdp
Allow-Events:  talk, hold, refer
Allow:  INVITE,ACK,CANCEL,BYE,REFER,OPTIONS,NOTIFY,
        SUBSCRIBE,PRACK,MESSAGE,INFO
Supported:  timer, 100rel, replaces, callerid
Session-Expires:  3600;refresher=uas
Min-SE: 90
Content-Type:  application/sdp
Content-Length:  368

v=0
o=root 431065466 431065466 IN IP4 SenderIP
s=call
c=IN IP4 SenderIP
t=0 0
m=audio 40000 RTP/AVP 0 8 9 2 3 18 4 101
a=rtpmap:0 pcmu/8000
a=rtpmap:8 pcma/8000
a=rtpmap:9 g722/8000
a=rtpmap:2 g726-32/8000
a=rtpmap:3 gsm/8000
a=rtpmap:18 g729/8000
a=rtpmap:4 g723/8000
a=fmtp:101 0-16
a=ptime:20
a=sendrecv
```

Table 3.10: Snom 300 INVITE message.

- The *request line* of a SIP message has to contain the method, a Request-URI and the SIP version. It is used to declare the message type. The Request-URI names the domain of the location service and it is used to locate the VoIP system.

Within a SIP REGISTER message the Request-URI is just the name of the domain of the location service, whereas within other SIP requests, the Request-URI must contain the user info as well. Also, the Request-URI of a SIP message should be set to the value of the URI in the *To* header.

- The *Via* header is on the one hand used to indicate the used transport protocol and on the other hand to specify the location where the response is to be sent.

  Besides the protocol name and version, the *Via* header must include a branch parameter. This parameter is used to identify the call created by a request and is used by both the client and the server. Table 3.11 shows that the BOL soft phone does not use a branch parameter in its *Via* header.

  The *rport* paramter is introduced in [RS03] and is used for symmetric response routing. By using the *rport* paramter the client requests that the server sends the response back to the source IP address and port where the request came from.

- The *From* header is used to indicate the identity of the initiator of a request. The *Display Name* is used to display the name of the initiator in a human user interface.

  Additionally to the URI a *From* header has to contain a *from tag* parameter. The *from tag*, along with the *to tag* and the *Call-ID*, is used to identify a dialog. The *epid* parameter (endpoint ID) is used by some hard phones to uniquely identify a SIP device.

  Routing, authentication and registration can benefit from the use of an *epid* parameter. The parameter *user=phone* indicates that the User-ID of the SIP URI should be treated as a telephone URI (a regular global telephone number).

- The *To* header is used to specify the recipient of a request. It usually just contains the SIP URI of the called party.

- The *Call-ID* header identifies a series of messages that form one call scenario. Usually the *Call-ID* should be the same in each registration process from a user agent.

- The *CSeq* header consists of a sequence number and the used method (e.g. REGISTER, INVITE). The sequence number orders a set of messages and the method must be the same as the method used in the *request line* header.
- The *Max-Forwards* header indicates the number of hops a message can transit to the destination. Usually the value should be decremented by one at each hop. Most of the phones set the *Max-Forwards* value to the suggested value 70 (see [RSC+02]).
- The *Contact* header is used to provide a SIP URI where the user agent can be reached for further requests. Tables 3.11 and 3.12 show that there are many parameters that can be used within the *Contact* header.

  For example the *flow-id* parameter is used to tell the SIP proxy the difference between a re-register request and registering an additional connection.
- The *User-Agent* header contains information on the used hard or soft phone.
- The *Allow* and *Allow-Events* headers are used to specify supported methods and features.
- The *Supported* header contains a list of option tags supported by the user agent.
- The *Authorization* and the *Proxy-Authorization* headers contain authentication credentials of a user agent.
- The *Route* header is used to force the message to be routed to specified proxies.
- The *Content-Length* header specifies the size of the message body (SDP).
- The *Content-Type* header specifies the media type of the message body (SDP).
- The *Expires* header specifies the time after which the message expires.

Header	Param.	Snom	Grands.	Elmeg	Siemens	Thomson	Linksys	Polycom	DLink	Allnet	X-Lite	BOL	PortSIP	ExpressT.	3CX
Request L.	*Domain*	x	x	x	x	x	x	x	x	x	x	x	x	x	x
	transport				x										
	Port								x	x					
Via	*IP Address*	x	x	x	x	x	x	x	x	x	x	x	x	x	x
	Port	x	x	x	x	x	x		x	x	x	x	x	x	
	branch	x	x	x	x	x	x	x	x	x	x		x	x	x
	rport	x		x							x		x	x	
From	Disp.Name	x	x	x	x		x	x							x
	User ID	x	x	x	x	x	x	x	x	x	x	x	x	x	x
	Domain	x	x	x	x	x	x	x	x	x	x	x	x	x	x
	tag	x	x	x	x	x	x	x	x	x	x	x	x	x	x
	epid				x							x			
	Port								x	x					
To	Disp.Name	x		x	x		x								
	User ID	x	x	x	x	x	x	x	x	x	x	x	x	x	x
	Domain	x	x	x	x	x	x	x	x	x	x	x	x	x	x
	Port								x	x					
Call-ID		x	x	x	x	x	x	x	x	x	x	x	x	x	x
CSeq		x	x	x	x	x	x	x	x	x	x	x	x	x	x
Max-Forw.		x	x	x	x	x	x	x	x	x	x	x	x	x	x
Contact	User-ID	x	x	x	x	x	x	x	x	x	x		x	x	x
	IP Address	x	x	x	x	x	x	x	x	x	x	x	x	x	x
	Port	x	x	x	x	x	x		x	x	x	x	x	x	
	Disp.Name				x		x								
	line	x		x											
	flow-id	x													
	q	x		x										x	
	audio			x											
	transport				x										
	expires				x		x							x	
	user=phone					x									
	*					x									
	methods							x				x			
User-Agent		x	x	x	x	x	x	x	x	x	x	x	x	x	x
Allow-Ev.	dialog	x		x											
	presence											x			
X-Real-IP		x		x											
www-Cont.		x		x											
Authorizat.		x	x	x	x	x	x	x	x	x	x	x	x	x	x
Expires		x	x	x		x		x	x	x	x		x		x
Cont.-L.		x	x	x	x	x	x	x	x	x	x	x	x	x	x
Allow			x				x				x				
Supported				x											
Route					x										
Event												x			

Table 3.11: SIP messages: differences between the tested phones. REGISTER messages.

Header	Param.	Snom	Grands.	Elmeg	Thomson	Linksys	Polycom	DLink	Allnet	X-Lite
Request Line	*UserID*	x	x	x	x	x	x	x	x	x
	Domain	x	x	x	x	x	x	x	x	x
	user=phone	x		x	x		x			
	Port				x			x	x	
Via	*IP Address*	x	x	x	x	x	x	x	x	x
	Port	x	x	x	x	x		x	x	x
	branch	x	x	x	x	x	x	x	x	x
	rport	x		x						x
From	*User ID*	x	x	x	x	x	x	x	x	x
	Domain	x	x	x	x	x	x	x	x	x
	tag	x	x	x	x	x	x	x	x	x
	user=phone				x					
	Port				x				x	
To	*User ID*	x	x	x	x	x	x	x	x	x
	Domain	x	x	x	x	x	x	x	x	x
	user=phone	x		x	x		x			
	Port				x			x	x	
Call-ID		x	x	x	x	x	x	x	x	x
CSeq		x	x	x	x	x	x	x	x	x
Max-Forwards		x	x	x	x	x	x	x	x	x
Contact	*User-ID*	x	x	x	x	x	x	x	x	x
	IP Address	x	x	x	x	x	x	x	x	x
	Port	x	x	x	x	x		x	x	x
	line	x		x						
	flow-id	x								
	user=phone				x					
User-Agent		x	x	x	x	x	x	x	x	x
Allow-Events	talk	x		x			x			
	hold	x		x			x			
	refer	x		x						
	conference						x			
Expires						x				
Content-Length		x	x	x	x	x	x	x	x	x
Allow		x	x	x	x	x	x	x	x	x
Supported	timer	x	x	x	x			x	x	
	100rel	x		x			x	x	x	
	replaces	x	x	x	x		x	x	x	
	callerid	x								
P-Key-Flags		x		x						
Accept	application/sdp	x		x						
Session-Expires	Seconds	x		x	x					
	refresher	x								
Min-SE		x								
Proxy-Authorization		x	x	x	x	x	x	x	x	x
Content-Type		x	x	x	x	x	x	x	x	x

Table 3.12: SIP messages: differences between the tested phones. INVITE messages.

3.5 Testing SIP VoIP systems

The demanded reliability of a VoIP system has to be as high as the reliability of a regular telephone system, thus testing the reliability of software products is crucial. Therefore a big part of the motivation behind my work was to test the reliability of state-of-the-art VoIP systems. Section 3.5.1 presents the basic motivation for testing VoIP servers. Section 3.5.2 focuses on the different call scenarios that can occur within a VoIP system and which therefore have to be tested. Section 3.5.3 presents the test cases I created with a commercial SIP test tool. These test cases enable companies to automatically test their VoIP system. Sections 3.5.4 and 3.5.5 present the two SIP test tools I developed and explain the basic idea behind these test tools.

3.5.1 Motivation

For VoIP to replace regular telephone systems it is necessary to guarantee high reliability. For VoIP system vendors testing their systems by writing and modeling test cases and testing software is a very time consuming task. Therefore software developers have to find a way to automatically test their software.
For instance, our industrial partner used a commercial test tool, the Spirent Protocol Tester (SPT) [Spi06], to create test cases and automatically run these test cases against their VoIP server.
Of course many different call scenarios can occur within VoIP systems (see Section 3.5.2). The optimal situation for a VoIP system developer is to have test cases for each scenario and then run these tests randomly and automatically. Of course, with a greater set of test cases, companies can achieve broader test coverage. Therefore my first task was to create such test cases with the Spirent Protocol Tester to get a feeling of the SIP protocol (see Section 3.5.3).

As the focus of my work shifted to different SIP dialects and the problems resulting from them, the need arose to not only work with the commercial test tool, but to develop individual test tools that are designed for my special needs.
Therefore simultaneously with creating test cases with the Spirent Protocol Tester, I developed two different SIP Test tools that focused on the problem with different SIP dialects (see Chapters 3.5.4 and 3.5.5).

As test environment I used the same environment I used to monitor the SIP traffic of the VoIP hard and soft phones and added the two created test tools, which were simulating yet another VoIP device (see Figure 3.7).

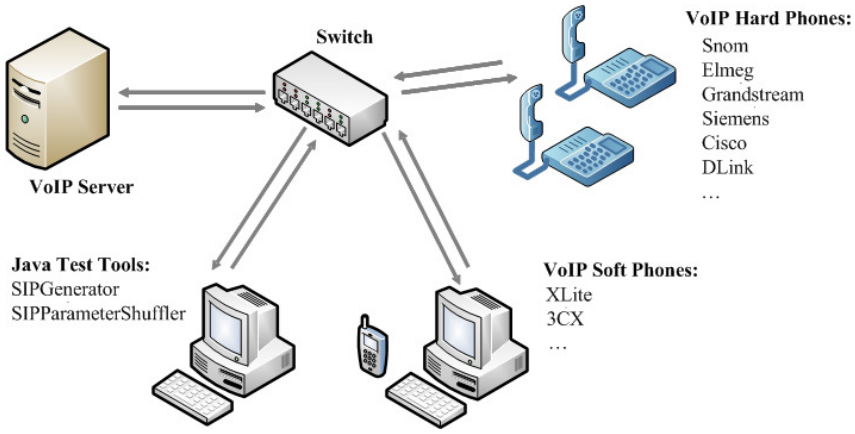


Figure 3.7: VoIP test environment.

Again, the tested hard and soft phones, along with the two created soft phone imitating test tools where connected to the VoIP proxy through a switch.
The soft phones and the developed Java test tools were installed on a standard personal computer. All including elements (VoIP server, hard phones, soft phones and test tools) were then passing SIP messages via the UDP protocol through the switch to each other.

3.5.2 Call Scenarios

This section describes various call scenarios that occur within a VoIP system. For the two most important and most often used scenarios, the register process and the basic call, the entire message flow will be discussed. The message flow for complex scenarios can be pretty extensive, so all the other scenarios will be discussed in general.

Register

As mentioned in Section 3.4 every phone within a VoIP network has to first register itself at the VoIP system before it is possible to establish or receive calls. So even though no call will be created, there is still the need to mention the register process because of its importance for all future call scenarios. Figure 3.8 shows the message flow within the register process of a typical SIP

VoIP phone. When the phone is first connected to the VoIP system, and continuously repeatedly after that when it is connected, it sends a REGISTER message to the VoIP proxy. The VoIP proxy usually responds with a 401 Unauthorized message, signaling the phone that it needs to authorize itself. The phone then re-sends the REGISTER message with an additional Authorization header.

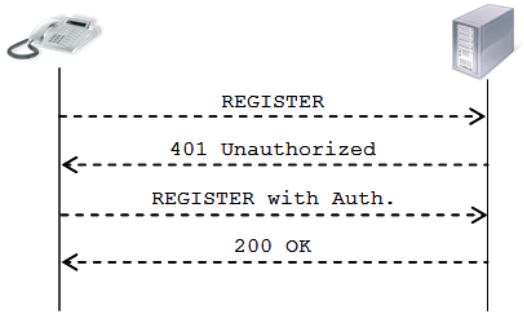


Figure 3.8: Call Flow, Register.

If the authorization process was correct, the VoIP proxy responds with a 200 OK message and the phone is successfully registered at the VoIP system. Section 3.4 gives additional information on the register process and also on the headers used within a SIP REGISTER message.

Basic Call

The simplest and most often used scenario is a basic call between two parties. Figure 3.9 shows the message flow of a simple basic call scenario.
To initiate a call, user A dials the number of user B and the SIP VoIP phone then sends an INVITE message to the VoIP proxy. Usually the SIP VoIP proxy responds with a 100 Trying and 407 Proxy Authorization Required message telling the phone to re-send the INVITE message with an additional Proxy-Authorization header. If the authorization process was correct, the proxy sends the INVITE message to user B. The SIP VoIP phone of user B responds with a 180 Ringing message and when the phone is picked up with a 200 OK message. User A receives these messages and then responds with an ACK (acknowledgement) message. After that the call is established. The call ends when one party hangs up the phone, sending a BYE message, followed by a 200 OK response. After the 200 OK message was received the call is successfully

terminated.

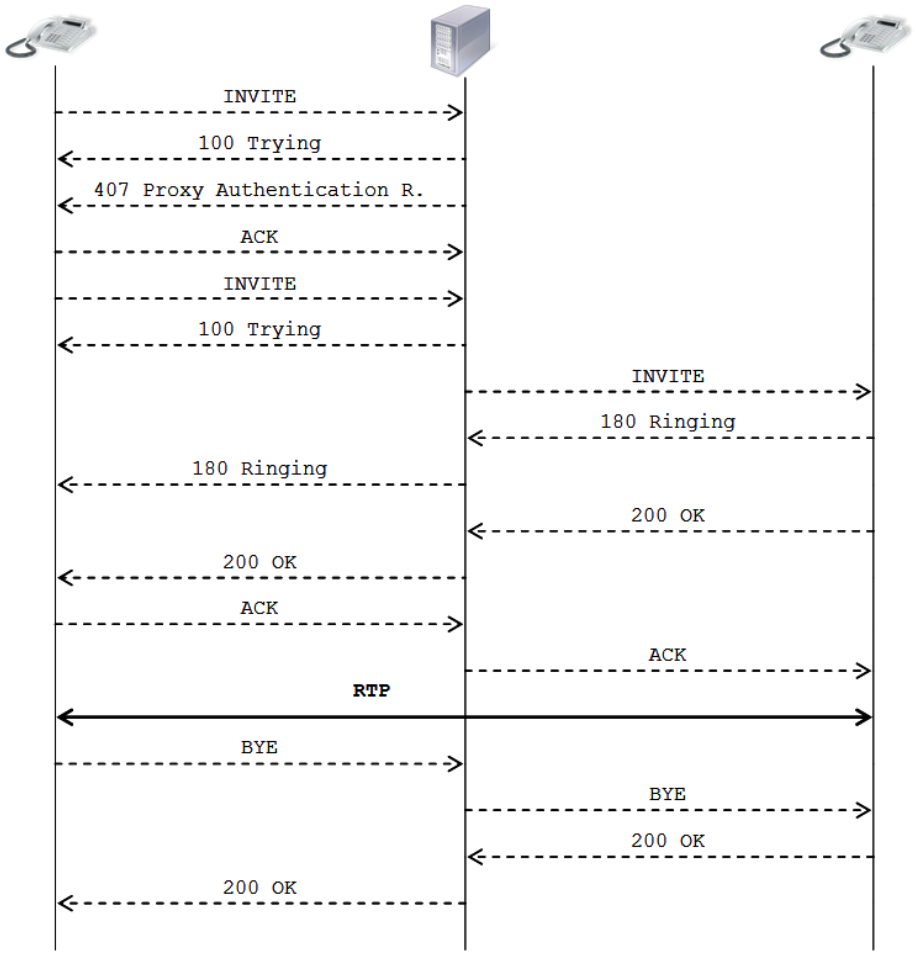


Figure 3.9: Call Flow, Basic Call.

Basic Call, Cancel

Of course it is possible that the called party is not available at the moment or cannot take the call. This scenario shows what happens if the caller does not want to wait any longer for the called party to pick up the phone.
Figure 3.10 shows the basic idea of a user calling another user who does not pick up the phone, either because he is busy at the moment or not available at the moment. User *A* then hangs up the phone sending a CANCEL message and gets a 487 Request Terminated response, successfully terminating the scenario.

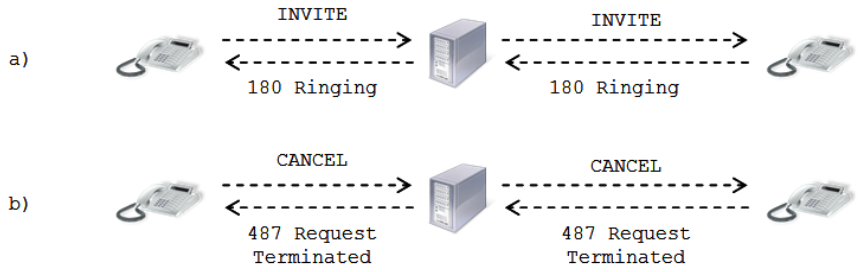


Figure 3.10: Call Flow (extract), Basic Call (no answer, cancel).

Basic Call, Deny

Another possibility, if the called party is busy at the moment the call is received, is to manually deny the incoming call. Figure 3.11 shows the idea that the called party denies an incoming call by sending a 486 Busy Here message terminating the scenario.

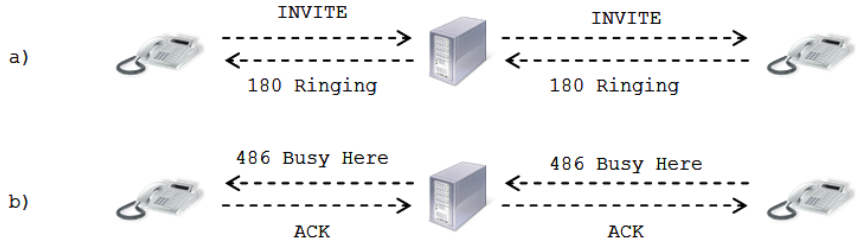


Figure 3.11: Call Flow (extract), Basic Call (busy, deny).

Parallel Ringing

Most VoIP systems allow users to activate certain features, like in this case parallel ringing. The user can specify one or more additional parties who will receive every incoming call that was intended for this user as well.

Figure 3.12 shows the basic message flow within a parallel ringing scenario. User A initiates the call by sending an INVITE message to user B. User B has activated the parallel ringing feature with (in this case) two additional receiving parties.

The proxy therefore forwards the initial INVITE message to user B, user C

and user *D*. Every party sends a 180 Ringing response back to the caller. One party then picks up the phone sending a 200 OK message to the caller and the call is successfully set up.

The proxy then sends CANCEL messages to the two other receiving parties stopping the ringing at the two phones.

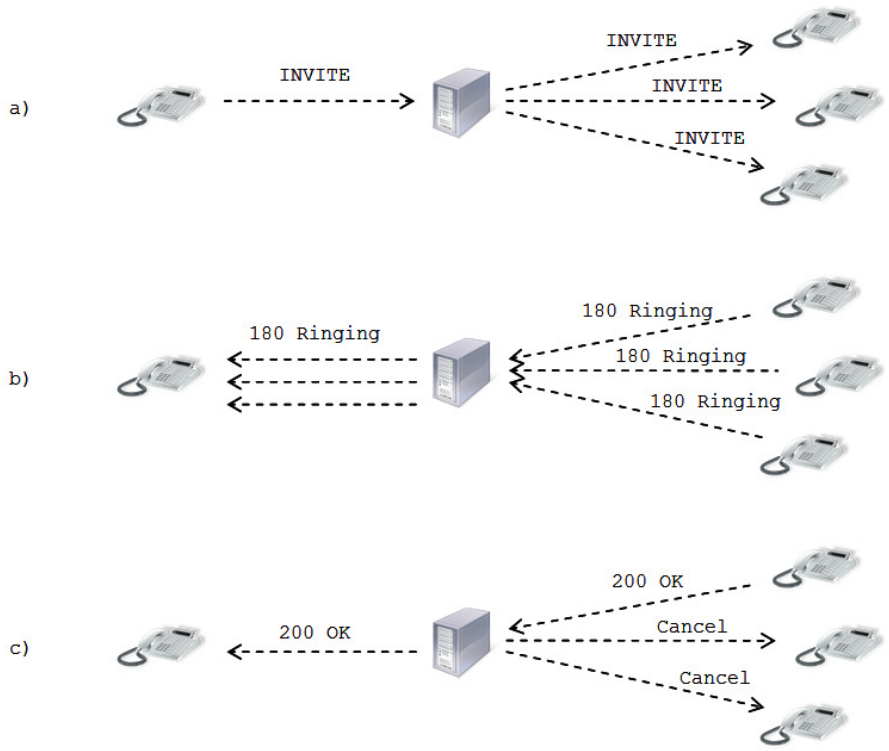


Figure 3.12: Call Flow (extract), Parallel Ringing.

Call Pickup

Another feature that can be activated is call-pickup. The basic idea is that one additional party gets notified on a specific call and is then able to pick up the call initially intended for someone else.
Figure 3.13 shows the basic idea that user *A* calls user *B*, who has the call-pickup feature activated with (in this case) one additional party, user *C*, who receives a NOTIFY message on the call. User *C* can then pickup the call sending a new INVITE message to the initial caller user *A*. After that, a

CANCEL message is sent to user B to successfully terminate the first initiated call and the call between user A and user C is successfully connected.

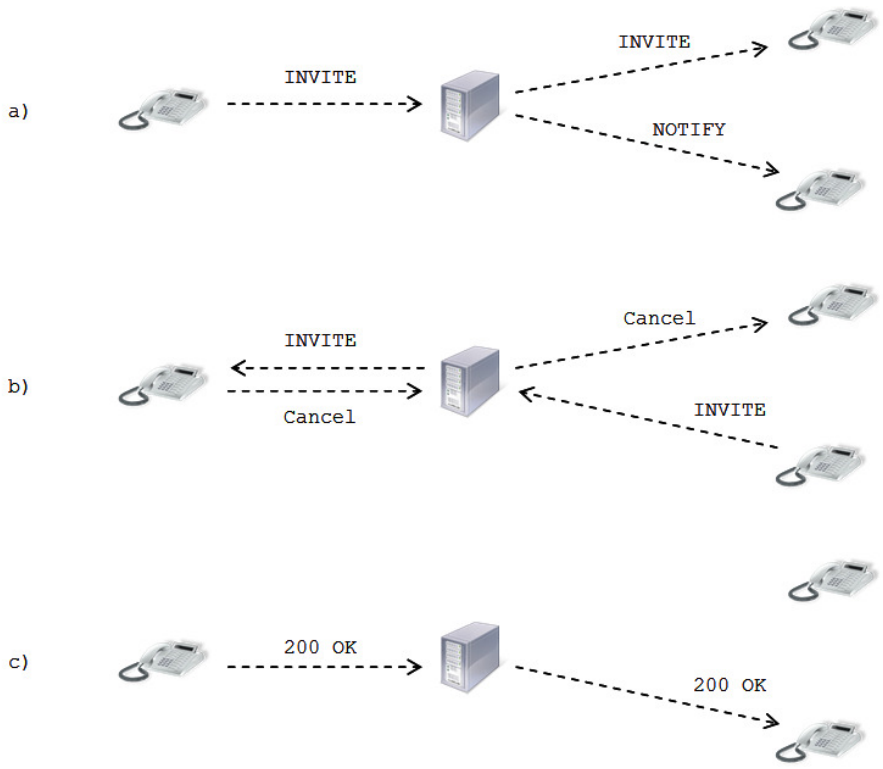


Figure 3.13: Call Flow (extract), Call Pickup.

Call Forwarding, Busy here

Another important feature is the call forwarding to a third party. There are different possibilities for a user to use this feature.

Figure 3.14 shows the first example of call forwarding to a third party. User A calls user B who is busy at the moment and denies the call by sending a 486 Busy Here message.

User B has the call forwarding on deny feature activated so after the 486 Busy Here message is received by the proxy, it sends the INVITE message to the specified third party, user C, who picks up the phone sending a 200 OK message to user A and the call is successfully established.

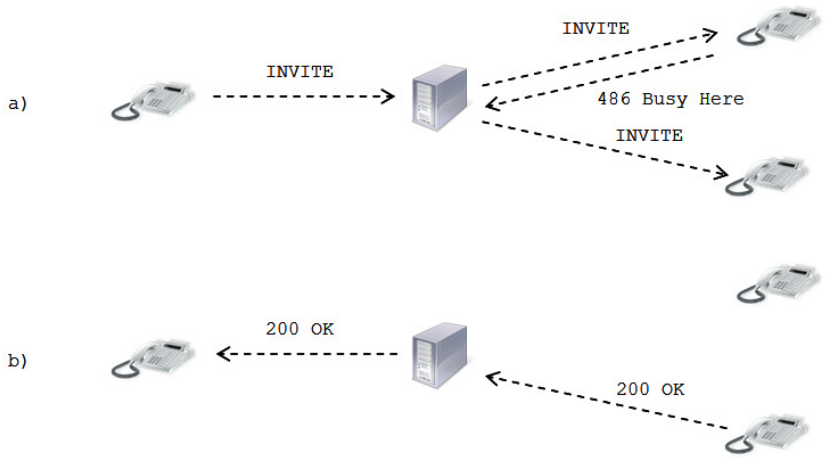


Figure 3.14: Call Flow (extract), Call Forwarding (busy here).

Call Forwarding, No response

Another form of call forwarding to a third party is if there is no response from the called party, the call gets automatically forwarded to a third party. Figure 3.15 shows that user A calls user B and after a certain time of no response from user B the proxy sends a CANCEL message to user B, who responds with a 486 Request Terminated message. After that the proxy sends the INVITE message to a specified third party, user C. User C picks up the phone sending a 200 OK message to successfully establish the call.

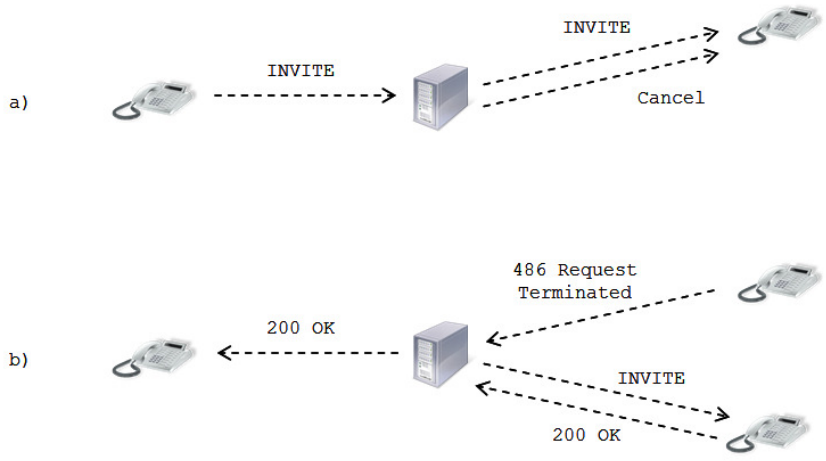


Figure 3.15: Call Flow (extract), Call Forwarding (no response).

Unattended Call Transfer

Sometimes during a call it is necessary to transfer the call to a third party, so two different calls have to be established. Figure 3.16 shows the basic idea of an unattended call transfer. User A calls user B who picks up the phone and at step a) the call between user A and user B is established.
User B then wants to transfer user A to a third party, user C, and therefore sends an INVITE message back to user A and also sends a REFER message to user A, containing all the information on user C necessary to establish a second call.
User A responds with a 202 Accepted message and sends a new INVITE message to user C, creating a new call. After the new call between user A and user C was successfully set up, user B is no longer involved in the call.

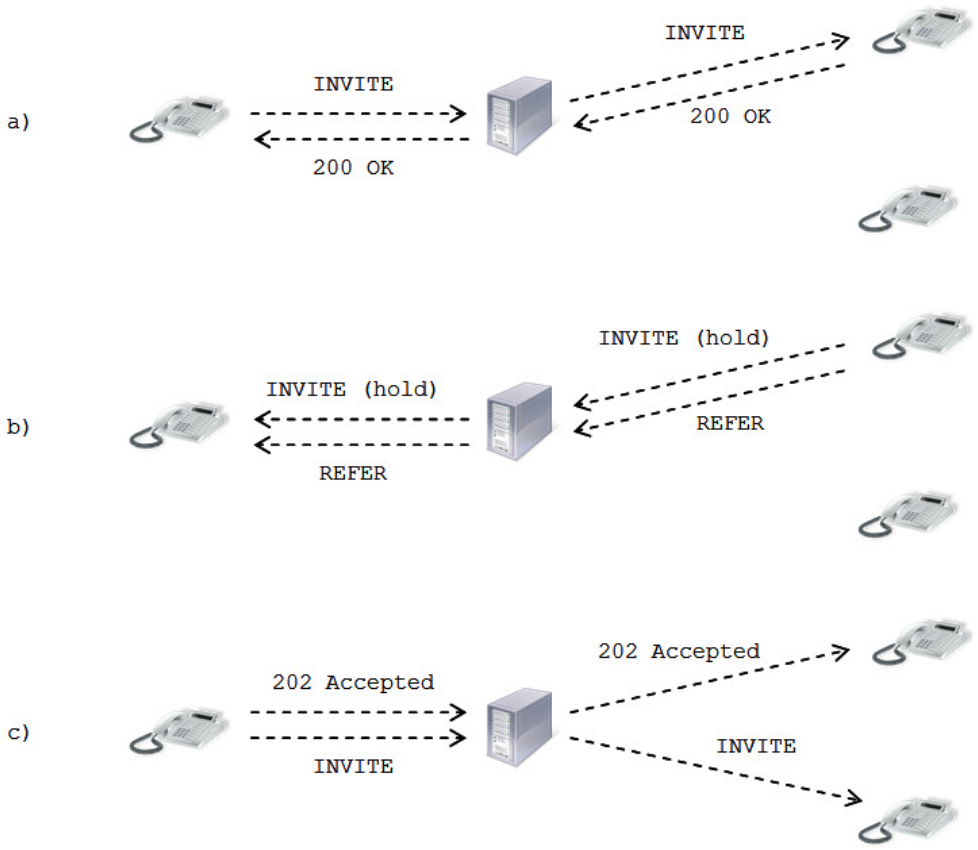


Figure 3.16: Call Flow (extract), Unattended Call Transfer.

During the conversation user B receives NOTIFY messages, keeping him informed on the status of the call between user A and user C.

Attended Call Transfer

A slightly different scenario is the attended call transfer. Sometimes, before transferring a call to a third party, it is necessary to call and inform the third party about the transfer. Figure 3.17 shows the basic idea of an attended call transfer. User A calls user B who picks up the phone and at step a) the call between those two parties is successfully established.

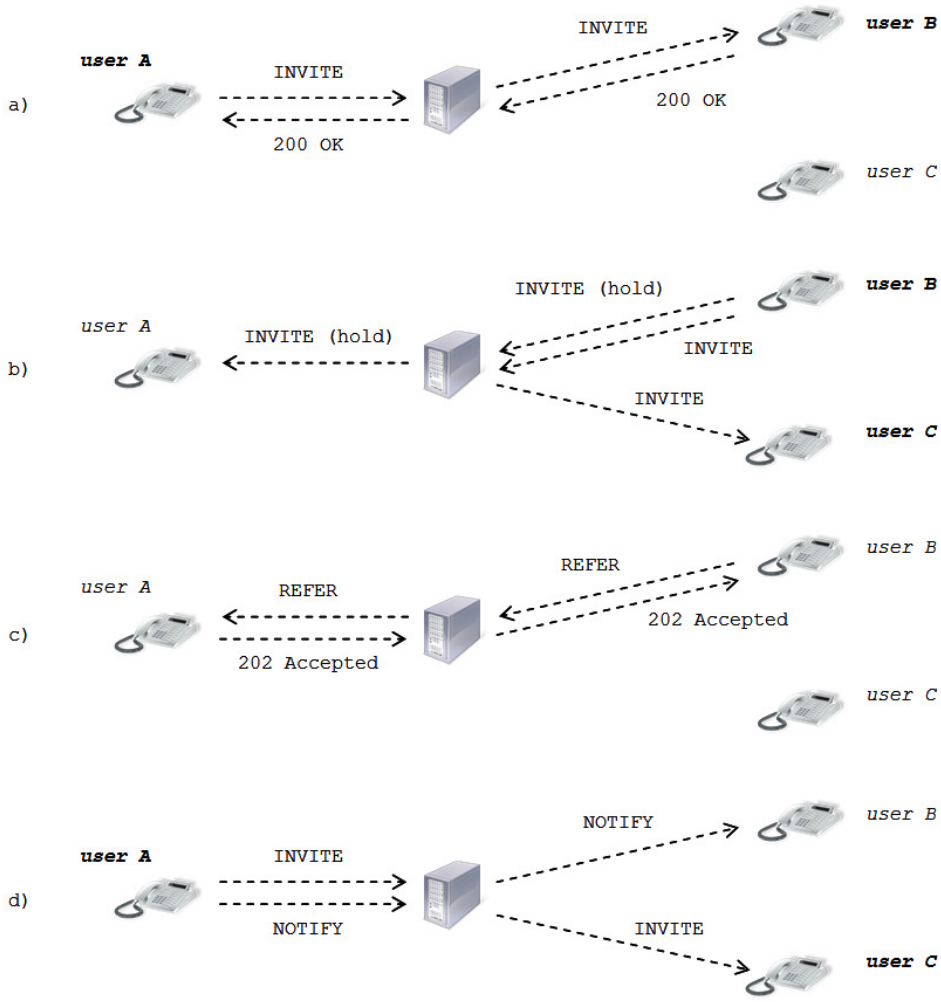


Figure 3.17: Call Flow (extract), Attended Call Transfer.

Then user B wants to transfer the call to user C, but first wants to talk to user C about referring the call. Therefore user B sends an INVITE (hold) message to user A, putting user A on hold, and an INVITE message to user C. User C picks up the phone and at step b) the call between user B and user C is established. If user C agreed to the call transfer, user B sends the REFER message, containing the important information on user C, to user A, who responds with a 202 Accepted message. After that user A sends a new INVITE message to user C and sends NOTIFY messages to user B, keeping user B informed on the status of the third call. So at step d) the call between user A and user C is successfully established.

ChefSec Call Transfer

An often used scenario is the so called Chef/Secretary (ChefSec) feature. The basic idea is that every incoming call that is intended for the chef is automatically forwarded to the secretary.

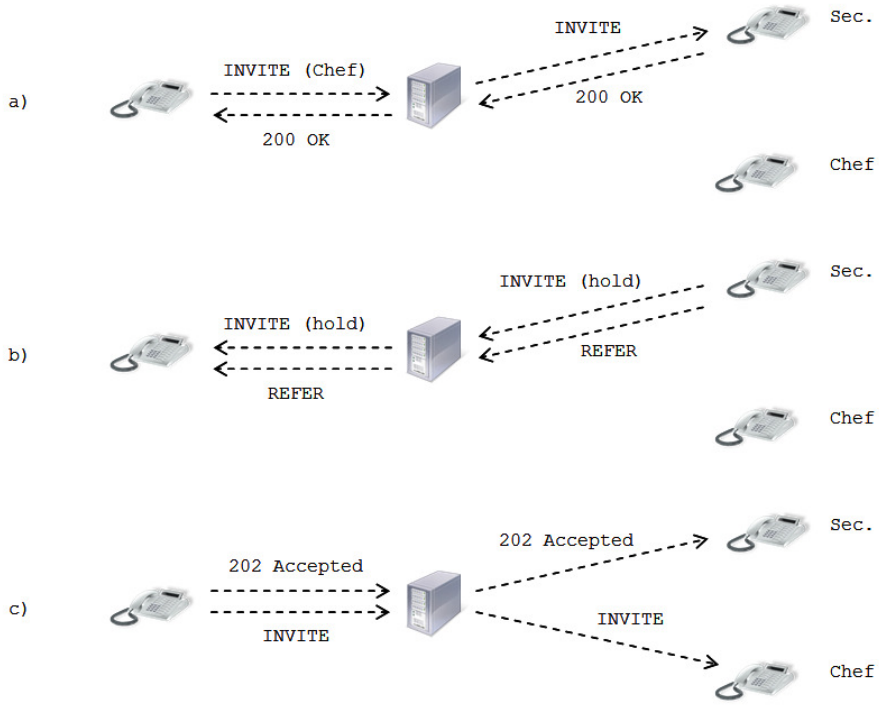


Figure 3.18: Call Flow (extract), ChefSec.

Figure 3.18 shows the basic idea of the Chef/Secretary (ChefSec) scenario. User A calls the chef who has the ChefSec feature activated, therefore the INVITE message is automatically forwarded to the secretary. Then after talking to user A, the secretary can transfer the call to the chef.
Again there are basically two possibilities for the secretary to do that, either an unattended call transfer where the call is automatically transferred to the chef (see Figure 3.18), or an attended call transfer where the secretary first establishes a call to the chef and then transfers the call.

Early Media

A user can also activate the early media feature so the caller not only hears a ringing noise, but a music or sound file played while waiting for the called user to pick up the phone and answer the call. Figure 3.19 shows the basic idea of the early media feature. User A calls user B and receives a 180 Ringing with Early Media response.
Usually the media files used in this type of scenarios are stored on the media server. The early media feature does not say anything about the scenario that follows after the 180 Ringing with Early Media message was received, but VoIP developers still need to test this scenario to guarantee that the feature was properly developed.

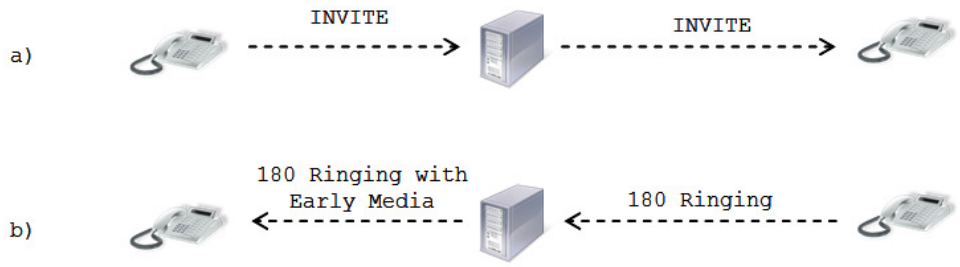


Figure 3.19: Call Flow (extract), Early Media.

More details on the exact SIP message flow within certain call scenarios can be found at [TI09].

3.5.3 Testing different call scenarios with a commercial test tool

The testing of VoIP systems requires test tools that offer high levels of scalability, re-usability and most important high levels of testing automation. Of course there are many SIP test tools available, so the first step was to evaluate the differences between these test tools. The following list presents some SIP test tools.
SIPsak [SIP02] is a command line tool for simple tests on SIP applications. It sends SIP requests via text files and, among other things, can be used for flooding tests and random character trashed tests. [HCL06] offers a SIP test tool for test automation. *ProLab SIP test Solution* [Rad06] offers stress and performance testing as well as media testing. The *Codenomicon SIP Test Tool* [Cod06] uses multiple carefully crafted messages to stress nodes in the SIP network. Also to capture and display SIP traffic the *Sipient* SIPFlow Standard [Sip06a] can be used.
The research of SIP test tools showed that SIPp [SIP06b] and the Spirent Protocol Tester [Spi06] offered the best features.

SIPp

SIPp is an open source SIP test tool with integrated scenarios that can be used for performance testing. Additionally to the included test cases, other test cases can be created in XML files to use them to test VoIP systems. SIPp offers a command line monitoring that allows the tester to dynamically adjust the used call rates. To analyze completed test cases SIPp offers a set of statistics including the total number of created calls, the call rate (calls per second), the number of successful and failed calls, the total time of one call and the response time of the VoIP system. Advantages of SIPp are

- dynamical display during the testing period,
- the possibility to dynamically increase/decrease the call rate,
- user agent can be used both as client and server,
- a great set of counters to analyze test scenarios,
- offers TCP and UDP over multiple sockets,
- available Third Party Call Control (3PCC),

- RTP echo feature to listen for RTP media, and
- the possibility to create and send XML call scenarios.

Spirent Protocol Tester (SPT)

With SIPp being a good open source test tool, I still chose the Spirent Protocol Tester to use as a test tool, because it offered a broader range of features. Advantages of SPT include

- a GUI (graphical user interface) that offers an easy way to create new call flows and call scenarios,
- the possibility to send up to 42.000 simultaneous calls,
- offers extensive diagnostics including an graphical call flow that can also be used for troubleshooting,
- includes a test suite manager that offers fully automated testing,
- includes high performance media generation and analysis for voice and video (RTP), and
- offers an easy way to manipulate (use, add, remove) any parameter of a SIP message.

Test case creation process with the SPT

The basic idea of the SPT is to create call scenarios recreating the message flow for all involved parties. For each party (SIP hard phone, SIP soft phone) at least one state machine has to be created simulating the message flow of (for example) a SIP hard phone. Multiple state machines are then combined into a test case that can then be executed.
So for each call scenario a single test case has to be created including state machines for the involved clients. To automatically test more than one test case at a time, one or more test cases can be combined into a test suite that can then be executed. SPT also contains a scheduler that allows the test case designer to start test cases or test suites at specified times. Test suites and the included scheduler provide great possibilities for test automation. Other features of the SPT include a database editor (to store user data like SIP URI, password, etc.), a load profile editor (to determine the outgoing call rate

for each state machine), a message editor (to create user defined message templates) and a report viewer (to read and verify call execution results, to display call flows and call sequence details).
To illustrate the test case creation process with the SPT, Figure 3.20 again shows the message flow of a basic call scenario. To get an executable basic call test scenario, two SPT state machines have to be created. The first step is to observe the outgoing and incoming messages within the message flow of a basic call for the caller side A. Figure 3.20 shows every message user A sends

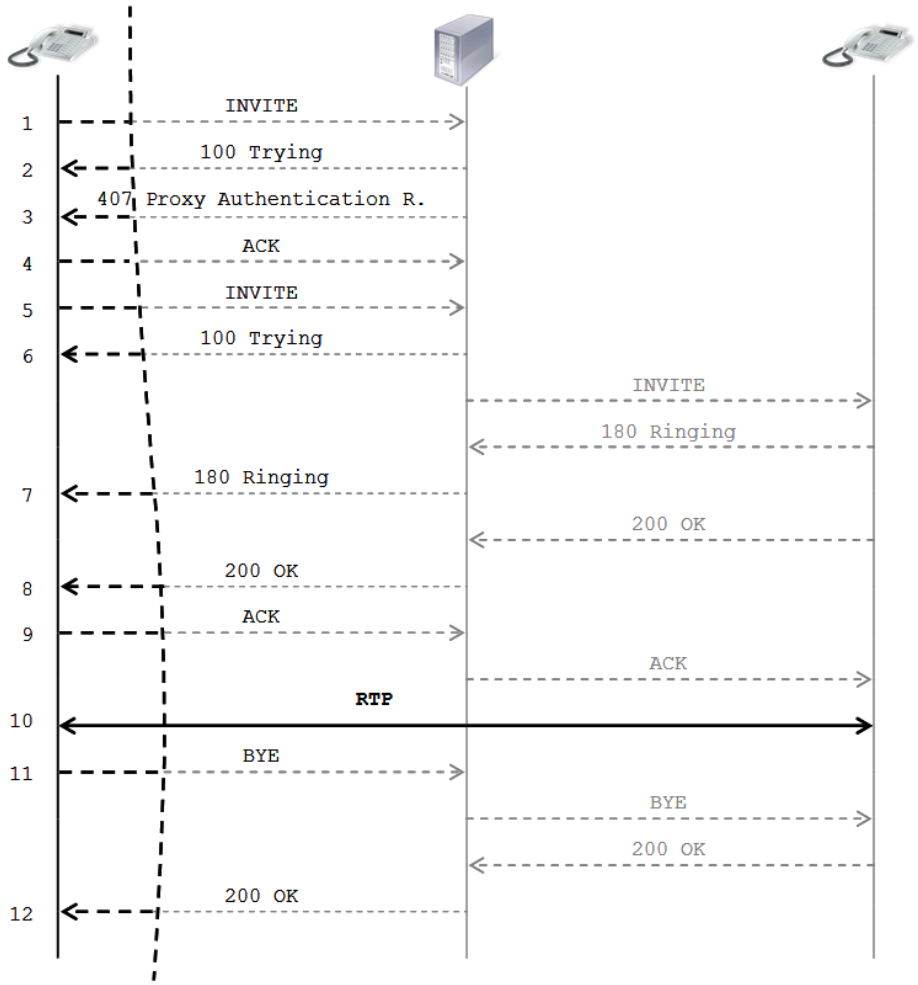


Figure 3.20: Call Flow, Basic Call, Side A.

and receives during a basic call. At (1) user A sends an INVITE message and then receives a 100 Trying (2) and a 407 Proxy Authorization Required (3)

message. Then user *A* sends an ACK (4) and again an INVITE (5) message and receives a 100 Trying (6), 180 Ringing (7) and 200 OK (8) message. After sending an ACK (9) message, the call is set up (10). After a certain time user *A* sends a BYE (11) message and receives a 200 OK (12) message terminating the scenario.

The next step is to create an SPT state machine that reproduces the mentioned message flow. The SPT state machine is a call flow design diagram which is composed of a set of states and transitions between states. All state machines begin with a Start state and end with a Stop state. In every state there are procedures that can be executed (sending messages, manipulating data, database operations, etc.).

The transition is a trigger event that initiates a change from one state to another. There are a few different condition types that trigger the transition from one state to the next state:
Unconditional Transition (Auto Transition):
The call flow diagram will automatically change from one state to the next state.
Receive Message:
A transition to next state will happen when a message is received.
Timer Expired:
The transition to the next state happens after a predefined period of time.
Criteria:
A transition to the next state occurs when a specific criteria is satisfied.

SPT also provides the possibility to extract certain parts of incoming messages and save the data into buffers enabling the call flow designer to use the data for creating other messages.

Figure 3.21 shows the state machine for user *A* during a basic call, recreating the message flow shown in Figure 3.20. Within the SPT state machine, boxes are used to display states and the arrows between the states show transitions. The state machine starts in the start state and follows the transitions to other states. Every state is used to execute certain tasks (e.g. writing data

into buffers).

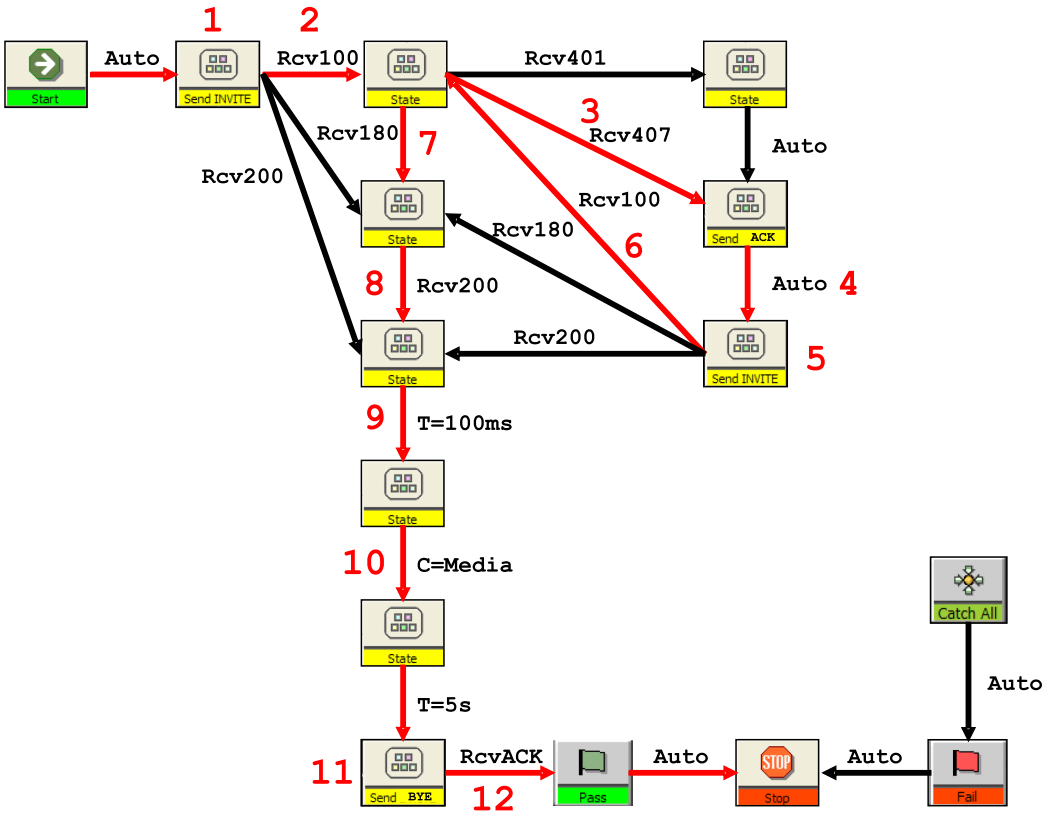


Figure 3.21: SPT State Machine, Basic Call, Side A.

At (1) all the initial buffers needed to run this scenario are set and the initial INVITE message is sent. Then there are the possibilities of an incoming 100 Trying, an incoming 180 Ringing or an incoming 200 OK message, represented by the three transitions. The state machine shown in Figure 3.21 therefore remains in the current state and waits for incoming messages.

Figure 3.21 shows that after a certain time the tested VoIP server responded to the sent INVITE message and answered with a 100 Trying message. So at (2) a 100 Trying message is received, followed by a 407 Proxy Authentication Required (3) message, that was also sent by the VoIP server.

At (4) the state machine has reached a new state. Within that state the ACK message is sent. Through an unconditional transition the state machine then jumps to (5). Within this new state the INVITE message will be re-

sent, this time including an Authorization header. After the second INVITE message was sent, the state machine remains in that state and again waits for incoming messages.

The VoIP server received the new INVITE message including the Authorization header and answers with an 100 Trying message that is then received by the state machine (6). Now the message flow brings the state machine back to the state in which a 100 Trying message was received. This time the Authorization header was included within the INVITE message and the VoIP server sends an 180 Ringing message which is then received by the state machine (7).

Now the state machine waits for an incoming 200 OK message from the tested VoIP server. After the 200 OK message was sent by the VoIP server and received by the state machine (8), an ACK message is sent by the state machine in state (9).

After the call is set up (10), the transition to end the call with sending a BYE message (11) happens after a predefined 5 second time period (timer expired transition).

The state machine then remains in that state and waits for the closing 200 OK message from the VoIP server. After the 200 OK message is received by the state machine in (12), the message flow ends up in the stop state, successfully ending the created test case.

Figure 3.21 also shows on the bottom right corner the Catch All state, which is a special state that catches all messages that are not defined within the regular call flow diagram. It will therefore be used especially for failure handling.

To finish the entire basic call scenario, the user B side of the call has to be recreated as well. Figure 3.22 again shows the message flow shown in Figure 3.20, but with the incoming and outgoing messages of user B highlighted.

Figure 3.22 shows that the scenario for user B starts by receiving the INVITE message from user A. User B responds with sending a 180 Ringing and

a 200 OK message. The call is set up after receiving the ACK message from user A. The receiving of the BYE message indicates the end of the call and user B responds with a 200 OK message terminating the call. Figure 3.23 shows the SPT call flow diagram for user B and the mentioned message flow. During

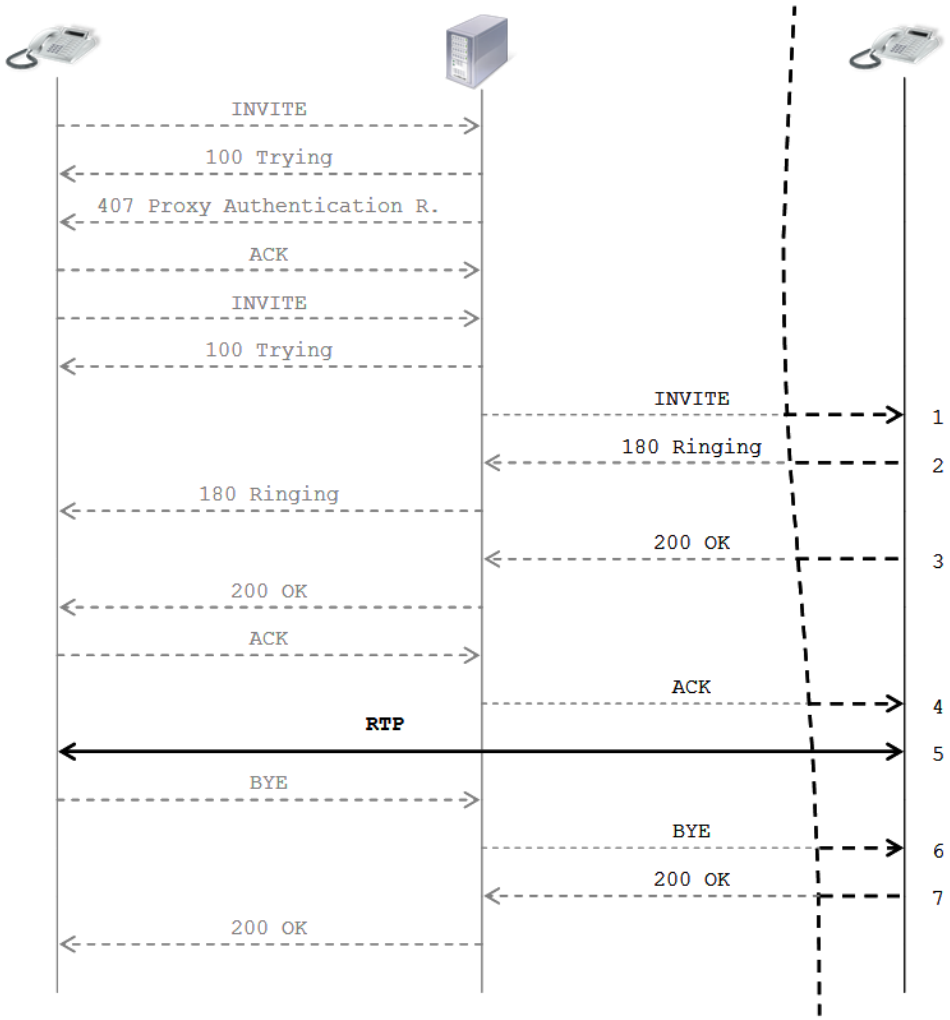


Figure 3.22: Call Flow, Basic Call, Side B.

my work I used the SPT to create 60 different test cases covering all the call scenarios mentioned in 3.5.2. Working with the SPT it became obvious that it was pretty easy to create new test scenarios by the simple drag and drop GUI.

Given a specific call flow the outgoing messages were simply represented by boxes and the incoming messages where represented by the transitions. Of

course the complexity grew with the difficulty of a test scenario. Especially with more then two involved parties it was a challenge to manage incoming messages, selecting the important parameters from that message, writing them into buffers and re-using them in newly created messages. These problems arose just out of the complexity of the call scenarios and, of course, would have been even bigger with other SIP test tools, like SIPp.

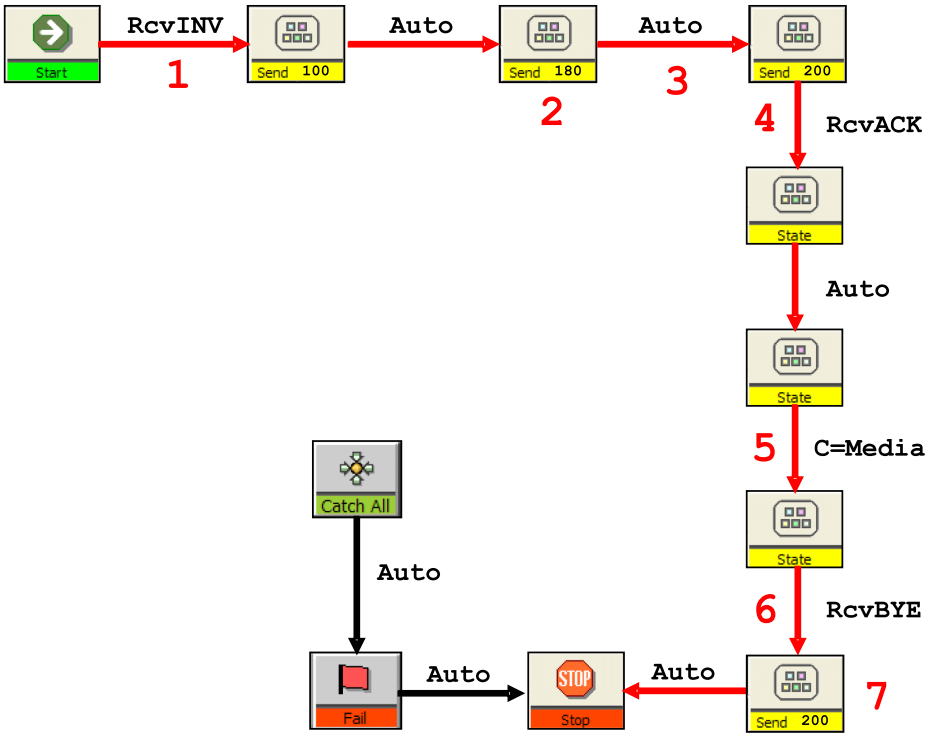


Figure 3.23: SPT State Machine, Basic Call, Side B.

The most important feature of the SPT for companies is the easy and perfect way to automatically test their developed VoIP system. The test suite manager along with the possibility to select different call rate distributions allows companies to run test cases in a highly automated way. With a great set of test cases the SPT is the right tool for companies to test their SIP product.

However as my work evolved, it became clear that test automation and just running the same test cases that are known to work with the VoIP server, would not be my main focus.
My focus shifted to exploring the different types of SIP dialects and therefore

it was necessary to have a test tool that allows you to simply and quickly change values of SIP messages and test them against a VoIP server. Of course that would have been possible with the SPT, but changing buffer values and message values cannot be done in a fast manner.

The idea behind the SPT is to take your time and create a message or a test case once, and have then the opportunity to test the test case at every time. For my purposes I thought there could be a better and easier way to quickly change values of SIP message and send them to a VoIP system. Therefore I decided to develop my own SIP test tool to meet my needs.

3.5.4 SIPGenerator

As I looked at different SIP test tools and created a great amount of test scenarios and test cases with the SPT, I got a good feeling about what my test tool should be focused on.

The main goal was to have a simple tool to send REGISTER and INVITE messages and the possibility to quickly change values of headers, header parameters and header values of a single SIP message. I therefore created a Java SIP test tool that met the following requirements:

- The possibility to send REGISTER and INVITE messages.
- The possibility to check if the call flow was successful and the SIP message created was accepted by the VoIP server.
- A possibility to choose from a set of predefined SIP messages, recreating already existing phones.
- A possibility to change certain details within the SIP message to check how false header details affect the acceptance of SIP messages.
- A simple GUI that shows the differences between certain SIP dialects, with the possibility to change SIP message details manually.

Appendix A shows the GUI of the Java program SIPGenerator. On the upper left corner Appendix A shows the SIP message that will be sent to the VoIP

server, where almost all the details of the SIP message can be changed manually. On the upper right corner you can choose between predefined hard and soft phones. And at the bottom the message flow will be displayed.

As mentioned before, the SIPGenerator offered the possibility to change values manually and very quickly; send the message to the VoIP system and check the result; change the values again and offered the possibility to directly compare the results or test new ideas.

The basic idea was to use the SIPGenerator to send SIP messages including faulty values to check the response from the VoIP system. The objective was to answer the following questions:

- What is the response from the VoIP server to a faulty message, header or value?
- Is there a response from the VoIP server?
- Does the VoIP server return the right error responses?
- Does the VoIP server crash, because of the faulty message?

After creating a REGISTER or INVITE message, the Java program creates a DatagramPacket with the SIP text message. This DatagramPacket will then be sent via a UDP socket to the VoIP server. Then the responses will be processed and new messages created and sent, to successfully terminate the scenario.

Listing 3.1 shows the important parts of the Java code that is necessary to send and receive SIP messages. The string variable SIPResponse is used to extract details from incoming messages that are needed to create additional SIP messages.

A challenge there is to create the REGISTER or INVITE message with the Authorization or Proxy-Authorization header, because the MD5 algorithm has to be used to generate all the details needed for these headers. Listing 3.2 shows

the parts usually used within an Authorization header.

```
[..]
//Datagram Details
public DatagramPacket pack=null;
public String MississippiIP = "XXX.XXX.XXX.XXX";
public final int MaxSize = 66000;
public byte[] pbuf = new byte[MaxSize];
//DatagramSocket
public DatagramSocket socket=null;
//UDP Socket
socket = new DatagramSocket(5060);
socket.setReceiveBufferSize(1000000);
socket.setSoTimeout(5000);
[..]
//SIP message
SIPMessage = "REGISTER sip: [..]";
//Sending SIP message
pack = new DatagramPacket(SIPMessage.getBytes(),SIPMessage.length(),new
    InetSocketAddress(InetAddress.getByName(MississippiIP),5060));
socket.send(pack);
//Receiving Response
pack = new DatagramPacket(pbuf,MaxSize);
socket.receive(pack);
SIPResponse = new String(pack.getData(),0,pack.getLength());
[..]
```

Listing 3.1: SIPGenerator Java code: sending and receiving SIP UDP messages.

The important thing creating the Authorization or Proxy-Authorization header is to generate the response header detail.

```
Authorization: Digest username="",realm="",nonce="",uri="",qop=auth,nc
    =00000001,cnonce="",response="",opaque="",algorithm=md5
```

Listing 3.2: SIPGenerator Java code: Authorization header.

Listing 3.3 shows the Java code using the MD5 algorithm especially for creating the response header detail for the Authorization header.

```
1  //MD5 for response value
   A1 = ToUserID + ":" + ToDomain + ":" + Password;
3  A2 = "REGISTER:sip:" + ToDomain;
   MessageDigest md = MessageDigest.getInstance("MD5");
5  md.update(A1.getBytes());
   MD5A1 = HexString.bufferToHex(md.digest());
7  md.update(A2.getBytes());
   MD5A2 = HexString.bufferToHex(md.digest());
9  MD5A1 = MD5A1.toLowerCase();
   MD5A2 = MD5A2.toLowerCase();
11 RespClear = MD5A1 + ":" + Nonce + ":00000001:" + Cnonce + ":auth:" +
       MD5A2;
   md.update(RespClear.getBytes());
13 Response = HexString.bufferToHex(md.digest());
   Response = Response.toLowerCase();
```

Listing 3.3: SIPGenerator Java code: MD5 algorithm for authorization.

3.5.5 SIPParameterShuffler

The basic idea behind the SIPGenerator test tool was to be able to manually change details and entries within a SIP message. The first approach was to generate SIP messages that would probably not be accepted by the VoIP server. The created message tried to create buffer overloads or used not expected data types to simulate SIP messages that a possible attacker would send to attack or trouble the VoIP server.

During the tests the focus shifted to the question: which SIP messages would not be accepted by a certain VoIP server, even though they follow the rules of RFC 3261? As mentioned in Section 3.4.1 different VoIP phones send different SIP messages in different dialects and therefore some VoIP proxy vendors only accept a small set of VoIP devices to be used with their system.

So in addition to the monitored hard and soft phones mentioned in Table 3.5 (see Section 3.4.1) there was a need to generate different SIP dialects that follow the rules of RFC 3261. For that purpose I had to create a test tool that met the following requirements:

- The possibility to send REGISTER and INVITE messages.

- The possibility to check if the call flow was successful and the SIP message created was accepted by the VoIP server.
- A possibility to choose from a set of predefined SIP messages, recreating already existing hard and soft phones.
- A possibility to manually choose a subset of SIP headers, header details and attributes from a predefined set.
- A possibility to randomly choose such a subset.
- A possibility to automatically create a SIP message using these chosen SIP headers, header details and attributes according to RFC 3261.
- A possibility to create a large amount of different SIP dialects and parameter combinations.
- A simple GUI that shows what attributes and header details are chosen to form the SIP message.

The reason I changed the testing process quickly, was to answer a more important question than with the first SIP test tool. The fundamental idea was the same as before: To get the VoIP server to reject messages or function incorrectly and thereby explain possible weaknesses or failures within the VoIP server.

With the SIPGenerator I tried to answer that question by mostly using faulty values within SIP messages. The objective with the SIPParameterShuffler was to answer the following questions:

- Can correct SIP message create problems for the VoIP server?
- Can certain headers or header combinations create problems for the VoIP server?
- Can certain parameters or parameter combinations create problems for the VoIP server?

Given these new requirements it did not make sense to just change the SIP-Generator test tool and therefore I created an additional Java test tool, the SIPParameterShuffler. Prior to that, a set of SIP headers, header details and

attributes had to be created. With the information found in [RSC+02] and the observations of the different VoIP phones mentioned in Table 3.5 (see Section 3.4.1), a set of the 53 most often used SIP headers, with 145 attributes, was created.

Appendix B shows the GUI of the SIPParameterShuffler test tool where at the upper right corner the user can choose the predefined SIP phones and on the bottom right corner the SIP message flow will be displayed.

On the left side the 53 SIP headers are shown, which are used to generate the different SIP messages. Furthermore each of the 145 attributes is represented by a checkbox, so when a new SIP message is created, every checkbox will be examined and the new message will be created.

Random selection of SIP headers and attributes

The most important requirement for the SIPParameterShuffler test tool was to create a set of very different SIP messages. So instead of manually creating every single message, a set of 145 SIP attributes was generated and the idea was to randomly choose a subset of attributes.

Even though the SIP messages should be a random combination of headers and attributes, the messages created should still look like they could have been sent from a real VoIP phone. So the idea was to create an intelligent fuzzing testing tool. Usually a fuzzing tool creates random and unexpected data [Wik10a].

My goal was to create random and unexpected data, but use correct and SIP-typical content. Therefore it was necessary to randomly choose a set of attributes but with consideration of the importance of every single header and attribute.
So with the knowledge of the messages sent from the tested VoIP phones shown in Table 3.5 and the messages created with the SPT test tool (see Section 3.5.3) a possibility was set for every attribute. The following three examples show the knowledge-based random selection of attributes:

Random selection, high probability, e.g.: Content-Length Header

Table 3.13 shows all header parameters that were used in more than 85% of all SIP REGISTER and INVITE messages of the tested real VoIP hard and soft phones (see Table 3.5). The third column presents how often a header parameter was used within the real phones, while the fourth column gives the probability the SIPParamterShuffler used to decide whether a header parameter should be included in a new message.

Header	Parameter	% Phones	% Shuffler
Request Line	UserID	100%	97%
Via	IP Address	100%	97%
	Port	86.96%	97%
	branch	95.65%	97%
From	User-ID	100%	97%
	Domain	100%	97%
	tag	95.65%	97%
To	User-ID	100%	97%
	Domain	100%	97%
Call-ID		100%	97%
CSeq		100%	97%
Max-Forwards		100%	95%
Contact	User-ID	95.65%	97%
	IP Address	100%	97%
	Port	86.96%	85%
User-Agent		100%	75%
Content-Length		100%	90%

Table 3.13: SIP messages: high probability header parameters.

Based on the observed knowledge of SIP messages, the Content-Length header appears in every real SIP message. So for the Content-Length attribute a random number between 1 and 100 was generated and to simulate a selection rate of 90%, the Content-Length header will appear in a newly created SIP message, if the random number created is between 1 and 90. The probability of 90% was chosen, because the Content-Length header appeared in every SIP message, but is not a mandatory header according to RFC 3261.
All headers and header parameters that are mandatory parts of a SIP message received a probability of 97%. The idea here is to get a few non-RFC-conform message to show if our proposed method would find these faulty messages intentionally created with the SIPParameterShuffler.

The header User-Agent appeared in every SIP REGISTER and INVITE message, but the chosen probability is still just 75%, because the User-Agent header has no important value within a SIP message.

Random selection, medium probability, e.g.: Expires Header

Table 3.14 shows the SIP header parameters that received a medium probability of being chosen by the SIPParameterShuffler to occur in a newly created SIP message.

Header	Parameter	% Phones	% Shuffler
Via	rport	34.78%	50%
From	Display Name	30.43%	40%
Expires		43.48%	50%
Allow	Invite	52.17%	45%
	Ack	52.17%	45%
	Cancel	52.17%	45%
	Bye	52.17%	45%
	Notify	52.17%	45%
	Refer	52.17%	45%
	Options	52.17%	45%
	Info	52.17%	45%
	Subscribe	43.48%	35%
	Prack	34.78%	30%
Content-Type	application/sdp	39.13%	50%

Table 3.14: SIP messages: medium probability header parameters.

The Expires header appears in approximately half of the SIP messages generated by real SIP phones. Again a random number between 1 and 100 was generated and if the number lies between 1 and 50, the Expires header will be used in the new SIP message.

Random selection, low probability, e.g.: Subject Header

Table 3.15 shows some header parameters that received a low probability to occur in a newly created SIP message.
The Subject header is hardly used by any real SIP phone, so it will only appear in a newly created message if the random number created between 1 and 100 lies between 1 and 5. Headers or header parameters which never occurred in messages derived from the tested VoIP phones either received a probability of

Header	Parameter	% Phones	% Shuffler
Request Line	user=phone	21.74%	25%
Contact	Display Name	13.04%	10%
Subject		0%	5%
Timestamp		0%	10%

Table 3.15: SIP messages: low probability header parameters.

5% or 10%, so that there is still a slight chance that they will be used in a newly created SIP message.
Listing 3.4 shows the basic mechanism for creating the knowledge-based random SIP messages.

As mentioned earlier for every SIP header and attribute there is one checkbox and every checkbox will be handled like the *ContentLength* checkbox shown in Listing 3.4. The SIP message itself will then be built by adding new text to the SIPMessage variable.

With this technique I generated 344 different SIP REGISTER messages and 122 different SIP INVITE messages. These messages will be used for further experiments.

```
[..]
Random r = new Random();
[..]
probability = r.nextInt(101);
if (probability<91) {contentlength.setSelected(true);}
else {contentlength.setSelected(false);}
[..]
if (contentlength.isSelected()==true) {SIPMessage=SIPMessage+"Content-
    Length: 0";}
[..]
```

Listing 3.4: SIPParameterShuffler Java code: Random selection.

SIPParameterShuffler Results

In our experiments we concentrated on the two most important SIP messages the INVITE and the REGISTER messages. As mentioned before 344 different REGISTER and 122 different INVITE messages were created with the SIPParameterShuffler. These messages were sent to the VoIP system and classified as either *accepted* or *rejected*.
A REGISTER message was marked *accepted* if the register process was successful, the 200 OK response was received and the user agent was successfully registered at the server. Otherwise it was marked *rejected*.
In case of the INVITE messages a basic call scenario was initiated and the INVITE message was marked *accepted* if the 180 Ringing response was successfully received. This means that a basic call would have been possible with the sent INVITE message. If there was no 180 Ringing response, the INVITE message was marked *rejected*.
Table 3.16 shows that out of the 344 REGISTER messages 266 were accepted and 78 were rejected by the VoIP server. Out of the 122 INVITE messages 53 were accepted and 69 were rejected by the VoIP server.

REGISTER messages	
Accepted	266
Rejected	78
Total	**344**
INVITE messages	
Accepted	53
Rejected	69
Total	**122**

Table 3.16: Test set, REGISTER and INVITE messages.

As mentioned before, the SIPParameterShuffler randomly created new SIP REGISTER and INVITE messages. The goal was to create messages that used different parameter combinations. For example Table 3.17 shows a SIP REGISTER message created with the SIPParameterShuffler. This message was sent to the commercial VoIP proxy and was then marked as *accepted* by the server.

Table 3.17 shows that the created SIP REGISTER message uses some SIP headers that did not occur in messages sent by the tested VoIP phones (e.g.:

```
REGISTER sip:SoftNetUniWien:5060 SIP/2.0
Via:  SIP/2.0/UDP 131.130.32.52:5060;branch=z9hG4bK-d87543;rport
From:  <sip:2001@SoftNetUniWien;user=phone>;tag=1f54431a
To:  "V2001 N2001" <sip:2001@SoftNetUniWien;user=phone>
Call-ID: NDYzYzMwNjJhMDRjYTFjMmI5NTE3NDRkOGFkYzY3OTc.
CSeq:  1 REGISTER
Max-Forwards:  70
Contact:  <sip:2001@131.130.32.52:5060;line=ab2001;transport=udp>;
          +sip.instance="<urn:uuid:767e30a9-0c85-4238-ab02-e04eb40f3722>"
User-Agent:  SIPRegParameterShuffler
Allow-Events:  hold
X-Real-IP: 131.130.32.16
WWW-Contact:  <https://131.130.32.52:443>
Content-Length:  0
Supported:  gruu
Route:  <sip:131.130.32.16;transport=udp;lr>
Record-Route:  <sip:131.130.32.16:5060;lr>
Accept-Encoding:  gzip
Date:  Sat, 13 Nov 2010
Timestamp:  54
Unsupported:  gruu,replaces
```

Table 3.17: SIP REGISTER message created with the SIPParameterShuffler: Accepted.

Date, Timestamp, Unsupported and so forth). Nevertheless the VoIP proxy accepted this message and a phone would have been able to register itself at the VoIP server using this message.

Table 3.18 shows another SIP message created with the SIPParameterShuffler. This time the message is marked as *rejected*. Table 3.18 shows that the SIP REGISTER message does not include a Via header. Although the Via header is not a mandatory SIP header in a SIP REGISTER message, the experiments showed that the tested commercial VoIP server needed a Via header to accept incoming messages.

Table 3.19 shows a SIP INVITE message created by the SIPParameterShuffler. Again, many optional headers (e.g.: Subject, Content-Encoding and so forth) are used within this message, but again, this message has been able to successfully start a SIP call.
Finally Table 3.20 shows a SIP INVITE message created by the SIPParameterShuffler that was marked *rejected*. The created message does not include the IP Address within the Contact header. Even though the Contact header

```
REGISTER sip:SoftNetUniWien SIP/2.0
From:  <sip:2001@SoftNetUniWien>;tag=vh347gh43f7
To:  <sip:2001@SoftNetUniWien>
Call-ID: sdbajdbbvfdjcajskaxvhjs
CSeq:  1 REGISTER
Max-Forwards:  70
User-Agent:  SIPGeneratorMN
Contact:  <sip:2001@131.130.32.52>
Expires:  3600
Content-Length:  0
```

Table 3.18: SIP REGISTER message created with the SIPParameterShuffler: Rejected.

is not a mandatory SIP header, if the header is included, the IP Address has to be included as well.

3.6 Autonomic SIP Adaption

This section deals with the autonomic adaption of SIP messages and the developed module Babel-SIP. Section 3.6.1 explains the basic motivation and the idea behind our approach. Section 3.6.2 deals with C4.5 Decision Trees and how they will be used by Babel-SIP. Section 3.6.3 presents the functionality of the Babel-SIP module. And Section 3.6.4 describes the experiments and shows the results we achieved with our approach.

3.6.1 Motivation and Introduction to Babel-SIP

When dealing with new protocols that are defined in a very open standard, like SIP, and defined in one or in many cases several RFCs, there are a few problems that may present themselves. First of all an open standard leaves a lot of room for interpretation which can lead to the problem that end devices may not be able to communicate with each other, even if they use the same protocol. In the case of SIP, that can lead to a great number of different dialects making it impossible to use a certain hard or soft phone with the used VoIP server.
Another problem for software developers, especially in early stages of development, is that software implementing open standards like SIP evolves over time, and often during the first years of deployment, products are either immature or do not implement the whole standard right away but rather only a subset.

```
INVITE sip:1001@SoftNetUniWien;user=phone SIP/2.0
Via:  SIP/2.0/UDP 131.130.32.52:5060;branch=z9hG4bK-d87543-
From:  "V2001 N2001" <sip:2001@SoftNetUniWien:5060>;tag=1f54431a
To:  <sip:1001@SoftNetUniWien;user=phone>
Call-ID: NDYzYzMwNjJhMDRjYTFjMmI5NTE3NDRkOGFkYzY3OTc.
CSeq:  1 INVITE
Max-Forwards:  70
Contact:  <sip:2001@131.130.32.52:5060;line=ab2001;
          rinstance=e81d72cc1e7e2768>;expires=3600
User-Agent:  SIPRegParameterShuffler
Allow-Events:  dialog,presence,hold
X-Real-IP: 131.130.32.16
Content-Length:  368
Event:  message-summary
Accept:  application/sdp
RSeq:  437322310
Content-Type:  application/sdp
Authentication-Info:  nextnonce="d62ws4942bd3okj362fc144c1dd8acf5"
Call-Info:  <http://www.example.com/photo/pic1.jpg>;purpose=icon
Content-Encoding:  tar
Subject:  Test
Unsupported:  callerid
```

Table 3.19: SIP INVITE (SDP part not shown) message created with the SIPParameter-Shuffler: Accepted.

As a result, standard compliant messages are sometimes wrongly rejected and communication fails [HAAM08].

So the basic idea was to create a system that can automatically adapt messages to improve acceptance rates, classify incoming messages, remember if an incoming message was successfully and correctly processed by the server, continuously learns which messages were problematic and suggests a way to alter problematic messages into messages that will probably be accepted by the server.
We therefore had to develop an independent software module that can be put in front of a VoIP proxy without affecting the code of the VoIP proxy (see Figure 3.24). That way Babel-SIP was of course developed for the use with a VoIP system, but the functionality of Babel-SIP can be reused with any other new protocols.

Figure 3.24 and 3.25 show Babel-SIP being installed right in front of the VoIP system. Incoming SIP messages will be intercepted by Babel-SIP, possibly

```
INVITE sip:1001@SoftNetUniWien;user=phone SIP/2.0
Via:  SIP/2.0/UDP 131.130.32.52:5060;branch=z9hG4bK-d87543-;rport
From:  <sip:2001@SoftNetUniWien>;tag=1f54431a
To:  <sip:1001@SoftNetUniWien;user=phone>
Call-ID: NDYzYzMwNjJhMDRjYTFjMmI5NTE3NDRkOGFkYzY3OTc.
CSeq:  1 INVITE
Max-Forwards:  70
Contact:  <sip:2001@5060;line=ab2001>;expires=3600
User-Agent:  SIPRegParameterShuffler
Allow-Events:  hold,conference
WWW-Contact:  <https://131.130.32.52:443>
Expires:  3600
Content-Length:  368
Session-Expires:  3600;refresher=uas
Content-Type:  application/sdp
Require:  timer
Timestamp:  54
```

Table 3.20: SIP INVITE (SDP part not shown) message created with the SIPParameter-Shuffler: Rejected.

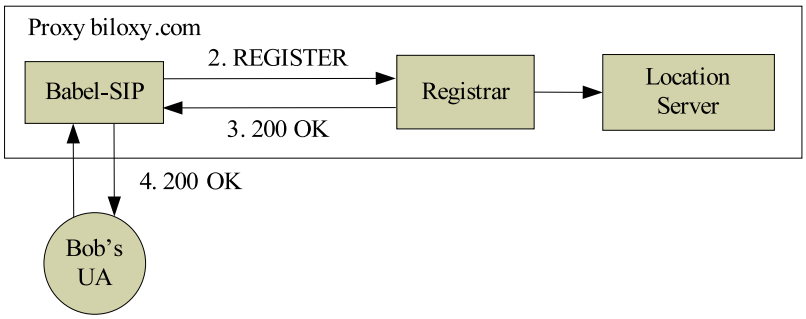


Figure 3.24: SIP registration of a user agent with Babel-SIP.

adapted and then forwarded to the VoIP server. Babel-SIP maintains a C4.5 decision tree, and observes which messages are accepted by the proxy, and which are not. This information is fed into the decision tree, the tree thus learns which headers are likely to cause trouble for this particular release of the proxy software [HNHH08].
The functionality of Babel-SIP will be presented in the following sections.

3.6.2 C4.5 Decision Trees

The first job of Babel-SIP is to classify incoming SIP messages. For classification a C4.5 decision tree is used, which is capable of further identifying relevant

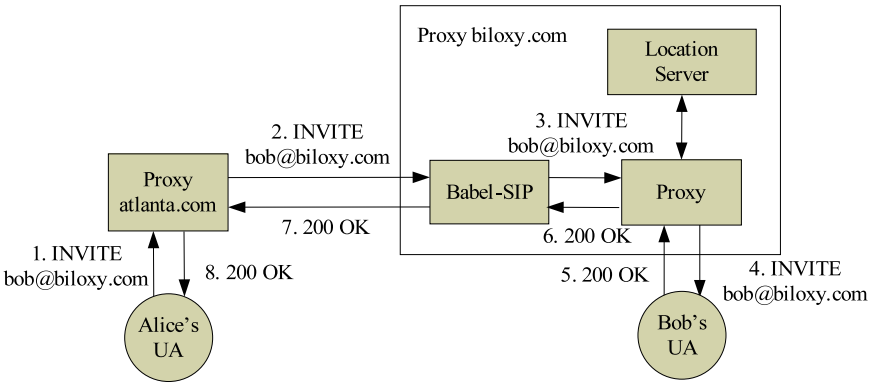


Figure 3.25: SIP call setup with Babel-SIP.

header parameters causing rejections. The C4.5 decision tree implementation (J48) used is based on the Weka machine learning library [WF05].

All 145 mentioned headers and parameters used by the SIPParameterShuffler (see Section 3.5.5), plus some additional headers, header fields and header field values (defined by RFC 3261) are defined as attributes. Altogether 274 attributes were defined and for each attribute a numerical value is defined to describe an incoming SIP message. So for every SIP message a vector of dimension d=274 is created. The value for each attribute can be 0 (if the corresponding header is not present), 1 (if the corresponding header is present and of type string), or any numerical value (if the corresponding header/header parameter is present and of numeric type) (see Listing 3.5) [HNHH08].
Note: String formats are not checked in our approach, because we assume that incoming messages are RFC 3261 conform.

```
1 Input:  attribute vector A
  Output: attribute vector A with new values
3
  FOREACH (Ai in A)
5    Ai.value = 0
     IF (Ai.name in SIP message) THEN
7      IF (SIP message field is numeric) THEN
         Ai.value = value of SIP message field
9      ELSE Ai.value = 1
```

Listing 3.5: Translation of SIP header into C4.5 attribute values.

After an incoming message is classified, a C4.5 decision tree is used to determine if the message is likely to be accepted or might be rejected by the server. A C4.5 decision tree basically represents a hierarchy of nested if-then rules.

Table 3.21 shows one example rule that can occur in such a form in a C4.5 decision tree. The two lines displayed in Table 3.21 represent leaves of the decision tree that are used to classify an incoming message. For example if the SIP header Call-ID occurs in the SIP message, the message will be classified as probably *accepted*. Otherwise the message will be marked probably *rejected*.

```
Call-ID <= 0:  REJECTED
Call-ID > 0:  ACCEPTED
```

Table 3.21: C4.5 example if-then rule.

Sections 3.6.3 and 3.6.4 will present further information on the usage of C4.5 decision trees by Babel-SIP.

3.6.3 Babel-SIP

Babel-SIP is an automatic protocol adapter and is placed in front of the SIP proxy that processes incoming messages (see [HNHH08] and [HAAM08]). Basically Babel-SIP handles four tasks:

- Classify incoming messages. An incoming message should be represented as a d=274 vector (see Chapter 3.6.2).
- A C4.5 decision tree should be generated with SIP messages (see Section 3.6.4).
- New incoming messages should be classified with the created decision tree in probably *accepted* and probably *rejected* (see section 3.6.4).
- For messages that are classified as probably *rejected*, the vector should be used to find a similar message that is known to be *accepted* and a suggestion should be made to change the incoming message accordingly. *Note:* Within our experiments, the problematic SIP messages were actually altered, since our goal was to see if Babel-SIP provides an improvement. In real-time applications altering messages should be done

carefully and semantics of the individual headers must also be taken into account (which has not been addressed here). Therefore if a message is classified to be probably *rejected* by the server, Babel-SIP tries to adapt messages in such a way that the result turns into a probably *accept*. For that purposes Babel-SIP stores messages that have been accepted in a local database M. For every message that is classified as probably *rejected*, Babel-SIP searches through its database to find the closest similar message that has already been *accepted*. The distances between two messages will be estimated with the Euclidean distance metric provided by Weka.

$$m_c = \arg \min_{m_i \in M \wedge m_i \neq m_1} d(m_1, m_i).$$

identifies the new incoming message as m_1 and the nearest accepted message within the database as m_c. Once a message m_c is found, Babel-SIP identifies those headers of m_1 that are classified as being problematic. These headers and header fields are then compared to the reference message. If the same header or header field is found in both messages, the values of m_c are copied into m_1. If a header or header field from m_1 is not found in m_c, it will be erased. If header or header fields of m_c are not used in m_1, they are inserted into m_1. The final step is to forward the new message to the proxy.

3.6.4 Experiments and Results

This chapter presents the experiments we made in our lab at the Department of Distributed and Multimedia Systems (now Entertainment Computing). In our lab we again used the same test environment as before, using the commercial VoIP system created by an industrial partner. We used the SIP messages created with the SIPParameterShuffler and ran several tests to evaluate the effectiveness of Babel-SIP, which is measured by the improvement of acceptance of previously *rejected* messages.

Figure 3.26 shows the test environment we used for our tests of Babel-SIP. Babel-SIP was installed in front of the VoIP proxy. Therefore every SIP message created with the SIPParameterShuffler was sent to the SIP proxy, but intercepted by Babel-SIP.

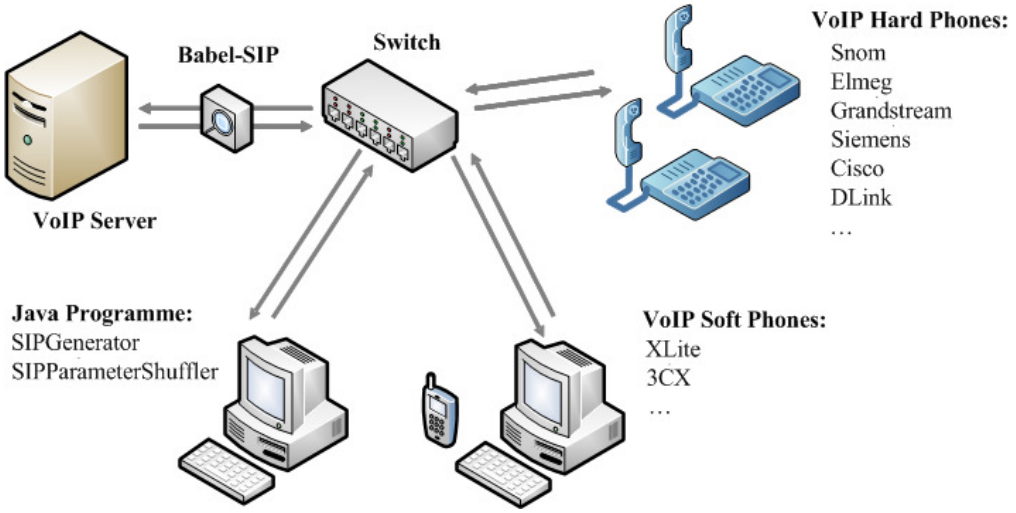


Figure 3.26: VoIP test environment, with Babel-SIP.

As mentioned in Section 3.5.5, 344 different REGISTER and 122 different INVITE messages were created.

REGISTER messages

The first thing Babel-SIP had to do was create a starting C4.5 decision tree. We therefore had to choose a training set out of the created messages.
In the first experiments we concentrated on the SIP REGISTER messages. To create a starting C4.5 decision tree a set of 50 REGISTER messages (of which 22% are known to be rejected) were randomly selected from the artificial messages created (see Table 3.22) [HNHH08].

REGISTER messages			
Training data set	Accepted	39	78%
	Rejected	11	22%
	Total	**50**	**100%**
Test data set	Accepted	227	77.21%
	Rejected	67	22.79%
	Total	**294**	**100%**

Table 3.22: Initial training and test data sets (REGISTER messages).

Figure 3.27 shows a C4.5 example tree generated by the training set. As mentioned in Section 3.6.2 the C4.5 decision tree shows a list of if-then rules. This

starting tree helps to classify new incoming messages in probably *accepted* or probably *rejected*. For example, if the parameter message-summary of the Event header occurs in a new incoming message, it will be classified as probably *rejected*, otherwise the next rule of the decision tree is taken into account.

```
Event_message-summary <= 0
|   Error-Info <= 0
|   |   Contact_transport <= 0: ACCEPTED (46.0/4.0)
|   |   Contact_transport > 0
|   |   |   Via_rport <= 0: REJECTED (2.0)
|   |   |   Via_rport > 0: ACCEPTED (8.0/1.0)
|   Error-Info > 0
|   |   Allow-Events_presence <= 0: REJECTED (2.0)
|   |   Allow-Events_presence > 0: ACCEPTED (2.0)
Event_message-summary > 0: REJECTED (4.0/1.0)
```

Figure 3.27: Example C4.5 tree after training with 70 REGISTER messages.

Also it can be noted that the learned rule hierarchy shows the importance of the parameter for its final acceptance. For example, the most important rule given by the starting tree, would be at the top of the tree (Event_message-summary <= 0). The leaves of the decision tree show the final classification in probably *accepted* or probably *rejected*. The numbers calculated for the tree leaves correspond to the number of messages that have been classified in this branch. The second number, in case it exists, shows the number of wrong classifications.

After a C4.5 starting tree exists, the test data set can be sent. To get a greater test set, the 294 messages were replicated 15 times. This led to a test set of 4410 SIP REGISTER messages. Then a set of experiments, consisting of 15 experimental runs, were started (each time 4410 messages were sent).

Therefore the first step was to create a starting decision tree. The next step was to choose a random message out of the 4410 test messages and send it to the VoIP proxy. The incoming message was then intercepted by Babel-SIP and ran through the C4.5 decision tree. With the help of each of the single rules, the decision tree classified the incoming message in probably *accepted* or probably *rejected*.
If the incoming message was classified as probably *accepted* it was simply for-

warded to the VoIP proxy. If the message was classified as probably *rejected*, the message was altered with the help of the local database of Babel-SIP (see Section 3.6.3).

Seeing our approach as self learning, the C4.5 decision tree of course has to be trained continuously. Training a C4.5 decision tree however is a very time and resource consuming task, so Babel-SIP creates a new C4.5 decision tree after each 20 new incoming messages. Therefore the first 20 messages will be classified with the C4.5 decision tree created with the training set consisting of 50 REGISTER messages. After these 20 messages were sent, a new decision tree is generated with the 70 available REGISTER messages. The next decision tree will then be created with 90 SIP REGISTER messages and so forth. Of course that means that the decision tree constantly learns and most certainly gets bigger with every new run.

For example the decision tree created after 70 SIP REGISTER messages only contains 5 rules that decide whether the incoming message is classified as probably *accepted* or probably *rejected*. Nevertheless the C4.5 decision tree (created with 70 REGISTER messages, see Figure 3.27) already correctly classifies 91.43% of the messages.

Figure 3.28 shows part of a C4.5 decision tree at the end of the REGISTER experiments. Of course it is obvious that the size of the tree has enormously increased. The entire C4.5 tree at the end of the REGISTER experiments can be found in Appendix A. Figure 3.28 shows that there are more rules and therefore it gets harder for a programmer to analyze the tree. On the other hand a decision tree derived at the end of the experiments can sufficiently explain why REGISTER messages may not work with the current version of the VoIP proxy.

For VoIP phone manufacturers such a decision tree can explain why a hard or soft phone fails to register itself at a proxy. For example, the decision tree shows which headers may have to be inserted into REGISTER messages or which parameter combinations might have to be avoided. The tree thus explains registration failure without knowing anything about the implemen-

tation. For VoIP proxy developers on the other hand, a decision tree might show failures within the code that can then be repaired for the next software update.

```
CSeq <= 0: REJECTED (104.0)
CSeq > 0
| Call-ID <= 0: REJECTED (101.0)
| Call-ID > 0
| | Replaces <= 0
| | | [...]
| | | To_IP <= 0
| | | | [...]
| | | To_IP > 0
| | | | [...]
| | | | | Via_IP <= 0
| | | | | | [...]
| | | | | Via_IP > 0
| | | | | | [...]
| | | | | | From_IP <= 0: REJECTED (2.0)
| | | | | | From_IP > 0: ACCEPTED (38.0)
| | | | | | [...]
| | | | | | Via_rport <= 0: REJECTED (11.0)
| | | | | | Via_rport > 0: ACCEPTED (30.0)
| | | | | | [...]
| | Replaces > 0: REJECTED (38.0)
```

Figure 3.28: Part of a C4.5 tree at the end of the REGISTER experiments.

Of course within the experiments the decision tree is used to classify incoming message in probably *accepted* or probably *rejected*. The classification accuracy of the decision tree throughout the experiments increased from 91.43% (Figure 3.27) to 99.32% (Figure 3.28, Appendix A). Thus, it can be assumed that this tree indeed classifies almost every incoming message correctly. Section 3.6.5 gives a more detailled analysis on the C4.5 decision trees derived at the end of the experiments.

The important question with Babel-SIP was if there would be an improvement of the acceptance rate of SIP messages. The experiments resulted in 15 time series of 4410 binary observation (yes or no). For each experiment $l, 1 \leq l \leq 15$ we then calculated rejection rates over overlapping bins of size 100 messages. The first bin $B_1^l = \{m_i^l \,|\, 1 \leq i \leq 100\}$ includes messages 1 to 100, the rejected messages of this bin are given by $\hat{B}_1^l = \{m_i^l \in B_1^l \,|\, m_i^l \text{ was rejected}\}$. B_2^l and

$\hat{B}_2^l$ are then computed over messages 21 to 120 from experiment l. In general, for $1 \leq k \leq 216$ we define

$$B_k^l = \{m_i^l \,|\, 20 \times (k-1) + 1 \leq i \leq 20 \times (k-1) + 100\}$$

and

$$\hat{B}_k^l = \{m_i^l \in B_k^l \,|\, m_i^l \text{ was rejected}\}.$$

Thus, an estimator $\hat{R}^l(i)$ for the rejection rate (in %) around message $m_i^l, 1 \leq i \leq 4400$ is given by

$$\hat{R}^l(i) = 100 \times |\hat{B}^l_{\lceil i/20 \rceil}|/|B^l_{\lceil i/20 \rceil}| = |\hat{B}^l_{\lceil i/20 \rceil}|$$

[HNHH08].

Figure 3.29 shows a smoothened curve of the estimated rejection rate. By using

$$\bar{B}_k = \left(\sum_{l=1}^{15} |\hat{B}_k^l| \right) /15, \; 1 \leq k \leq 216, \tag{3.1}$$

we define a mean estimator by

$$\bar{R}(i) = \bar{B}_{\lceil i/20 \rceil},$$

i.e., the mean is calculated for each bin over all 15 experimental runs. In Figure 3.29 it can be seen that the time series shows a transient phase at the start, in which the rejection rate decreases [HNHH08]. Therefore especially in the beginning Babel-SIP gradually learns and increases its effectiveness. After some time, the decision tree enters a stationary phase, were no more new things can be learned and therefore no more gain is achieved.
Taking messages m_{600}^l to m_{4400}^l into account, we have computed the mean rejection rate $R^s \approx 13.12\%$ for messages in the stationary phase. This represents the main results of the Babel-SIP REGISTER experiments. Thus, Babel-SIP is able to drop the rejection rate from 22.79% (see Table 3.22) to 13.12%.

Table 3.23 shows aggregated results over the REGISTER experiments in more detail. Therefore 18.37% of the messages have been modified by Babel-SIP. From all sent messages 9.33% have been successfully modified. From those

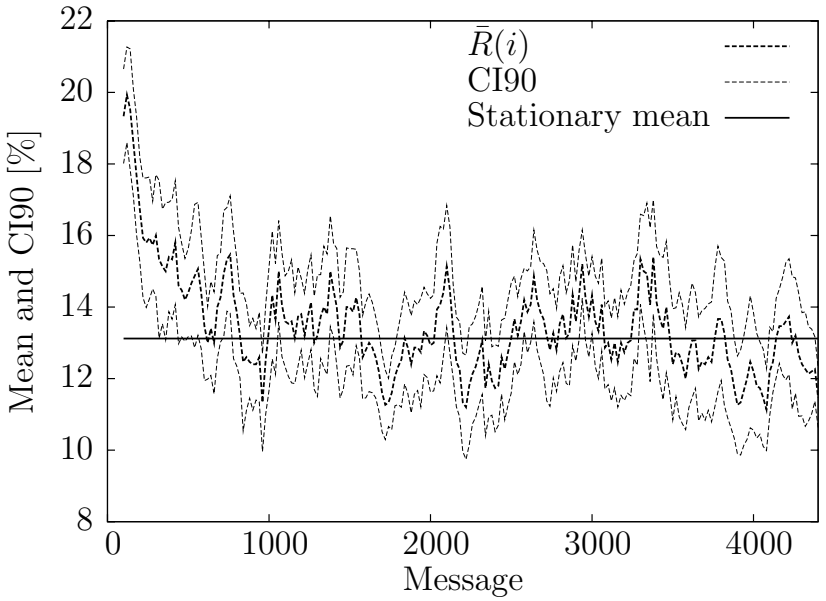


Figure 3.29: Result REGISTER messages.

messages classified as *rejected* 48.02% were successfully modified.

False positives are messages that were classified as *rejected*, although they would have been *accepted* by the proxy. False negatives are messages that were classified as *accepted*, although they would have been *rejected* by the proxy.

	Mean [%]	Std.dev. [%]
Modified	18.37	5.8310^{-4}
Successfully modified (from all)	9.33	0.003
Successfully modified (of those classified as rejected)	48.02	0.108
False positive	6.13	2.04110^{-4}
False negative	10.53	4.23310^{-4}

Table 3.23: Aggregated results of the Babel-SIP REGISTER experiments.

Of course false positives are very critical, because this means that an incoming message that would have been *accepted* by the server is wrongly classified as *rejected* and then modified. An unacceptable behavior for Babel-SIP would be if accepted messages were modified in a way that the new version $\hat{m}$ of m would be actually rejected. Table 3.23 shows that 6.13% of the REGISTER

messages that would have been *accepted* by the proxy were classified as *rejected* and therefore modified. However all altered messages have still been accepted by the proxy. Additionally we tested the 9 hard and 5 soft phones (see Table 3.5) with Babel-SIP installed, and all messages were classified as *accepted*.

Table 3.23 shows the mean over all 15 runs, as well as the standard deviation between the individual runs. Seeing that the standard deviations are very small, all the 15 experiments show almost equal results.

INVITE messages

In the second step we used INVITE messages to test Babel-SIP. Basically the same setup was used. For creating an initial C4.5 decision tree we used 20 randomly selected INVITE messages (see Table 3.24). Figure 3.30 shows the

INVITE messages			
Training data set	Accepted	9	45%
	Rejected	11	55%
	Total	**20**	**100%**
Test data set	Accepted	44	43.14%
	Rejected	58	56.86%
	Total	**102**	**100%**

Table 3.24: Initial training and test data sets (INVITE messages).

initial C4.5 decision tree after the initial training phase and one testing phase. A new decision tree is again created after 20 new incoming messages. Again the decision tree at the beginning of the experiments is pretty small and again contains only five rules. Figure 3.31 shows part of the C4.5 decision tree at the end of the INVITE experiments, while Appendix B shows the entire tree. Also, as mentioned before, Section 3.6.5 presents a more intensive tree analysis.

For the INVITE experiments the remaining 102 messages were replicated 10 times leading to a test set of 1020 messages. Again 15 experimental runs were started and we again computed a mean estimator $\bar{R}(i)$ and the 90% confidence interval, as well as an estimator for the stationary rejection rate (see Figure 3.32). For the stationary mean we used the messages m^l_{560} to m^l_{1000} resulting in a stationary mean of 30.79%. Therefore Babel-SIP was able to improve the rejection rate from 56.86% to 30.79%, resulting in an improvement of 45.85% (see Figure 3.32). Details on the results of the INVITE experiments are shown

```
| | Replaces > 0: REJECTED (38.0)
Accept_application/x-private <= 0
|   Allow-Events_refer <= 0
|   |   Call-Info <= 0
|   |   |   Content-Type_text/html <= 0
|   |   |   |   Timestamp <= 0: ACCEPTED (22.0/1.0)
|   |   |   |   Timestamp > 0: REJECTED (2.0)
|   |   |   Content-Type_text/html > 0: REJECTED (2.0)
|   |   Call-Info > 0: REJECTED (3.0)
|   Allow-Events_refer > 0: REJECTED (5.0)
Accept_application/x-private > 0: REJECTED (5.0)
```

Figure 3.30: Example C4.5 tree after training with 40 INVITE messages.

in Table 3.25. The same statistics are used for both the REGISTER and the INVITE results.

	Mean [%]	Std.dev. [%]
Modified	38.37	0
Successfully modified (from all)	22.39	1.210
Successfully modified (of those classified as rejected)	58.55	3.165
False positive	6.86	0
False negative	23.53	0

Table 3.25: Aggregated results of the Babel-SIP INVITE experiments..

Table 3.25 shows that the INVITE experiments result in a higher number of false negatives. Thus, it can be noted that the decision trees are not as accurate in distinguishing the critical headers as the trees derived within the REGISTER experiments. The zero standard deviation indicates that the decision trees of the 15 experimental runs each classify the messages equally. Seeing as the messages are selected and sent randomly, the decision tree should assumably learn some faulty parameters sooner or later than in other runs. That fact can probably be linked to the fact that the INVITE experiments only included 122 different messages, while the REGISTER experiments included 344 messages.

Retries

The REGISTER and INVITE experiments showed that Babel-SIP was able to drastically reduce the rejection rate of problematic SIP messages. Nevertheless

```
To_IP <= 0: REJECTED (72.0)
To_IP > 0
| Replaces <= 0
| | []
| | From_IP <= 0
| | | Allow-Events_conference <= 0: REJECTED (31.0/1.0)
| | | Allow-Events_conference > 0: ACCEPTED (10.0)
| | From_IP > 0
| | | []
| | | Content-Type_text/html <= 0: ACCEPTED(18.0)
| | | Content-Type_text/html > 0: REJECTED (2.0)
| | | []
| | | Privacy_header <= 0: ACCEPTED (58.0/2.0)
| | | Privacy_header > 0
| | | []
| | | Request-Line_transport <= 0: REJECTED (25.0)
| | | Request-Line_transport > 0: ACCEPTED (9.0)
| | | []
| Replaces > 0: REJECTED (48.0)
```

Figure 3.31: Part of a C4.5 tree at the end of the INVITE experiments.

there were still messages that had been rejected by the VoIP server, even after modifying them with Babel-SIP.

Therefore it became interesting to examine how many retries it would take to alter the same message before it may have been accepted by the server. For this, we ran separated REGISTER and INVITE experiments.

Before starting these experiments a decision tree was again generated with the training set used in the earlier experiments. This time we used the 67 REGISTER and 58 INVITE messages that were rejected by the server. Each experiment was driven by a parameter r, stating the maximum number of times a phone would try to register itself or try to initiate a call.

Table 3.26 shows the number of necessary tries up to that value of r from which onwards no improvement of the number of accepted messages was achieved.

The results reveal that for those phones that would need more than one try, most of them would succeed after at most four tries. Mapping this onto a realistic scenario, this means that a phone owner would have to attempt to

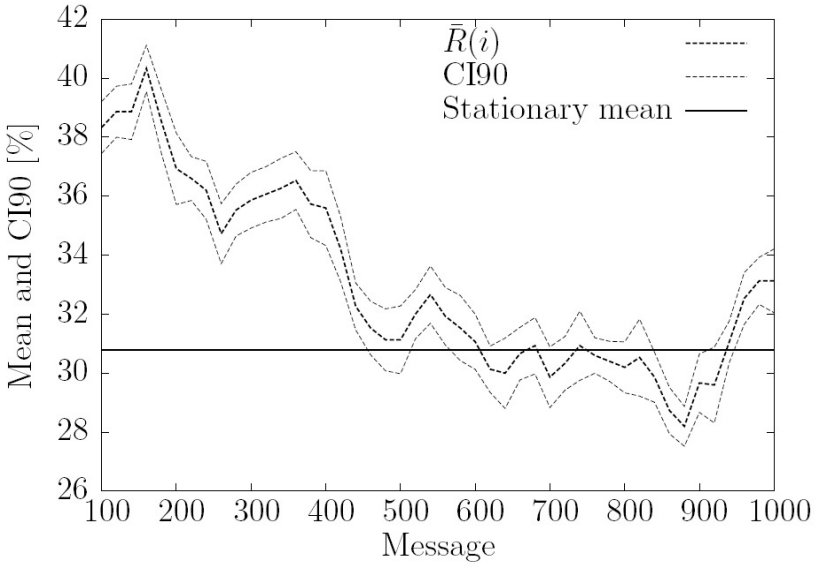


Figure 3.32: Result INVITE messages.

register his phone or to initiate a call only a few times [HNHH08].

REGISTERs	1	2	3	4	≥ 5	never
r=1	10					57
r=2	14	8				45
r=3	14	12	3			38
r=4	21	5	3	1		37
r=5	17	7	5	2	0	36
r=6	16	13	2	0	0	36
r=7	19	9	1	2	0	36
r=8	16	5	7	2	1	36
r=9	18	7	5	2	0	35

INVITEs	1	2	3	≥ 4	never	
r=1	22				36	
r=2	25	3			30	
r=3	21	5	2		30	
r=4	25	1	5	1	26	

Table 3.26: Number of necessary attempts for experiment r.

3.6.5 Qualitative Analysis of Decision Trees

One of the outcomes of the Babel-SIP experiments, besides the fact that it was able to drastically improve the acceptance rate of SIP messages, were the

C4.5 decision trees. These trees show why messages were rejected by the VoIP proxy. Messages could either be rejected because they were created in a non-RFC-conform way, or because of bugs in the VoIP server software.
The C4.5 decision trees do not state if the error occurred on the client or server side, but it points to the problematic headers, header parameters or header parameter combinations.
This section discusses the outcome of the previously presented Babel-SIP experiments, the C4.5 decision trees derived at the end of the experiments. Of course these two decision trees are very complex and large, but the following section will discuss the most important rules and parts of these trees.

C4.5 decision tree, REGISTER experiments

Appendix A shows the entire C4.5 decision tree derived at the end of the REGISTER experiments. The procedure in reading such a decision tree is to process the rules from the outside step-by-step to the inside or from the top to the bottom. The rules stated at the top are the most important ones and often state facts or problems right away.
Figure 3.33 shows the first few rules of the decision tree. The very first rule states that if an incoming SIP REGISTER message does not include a sequence number (a *CSeq* header) the message will be rejected by the server. If a *CSeq* header is included the decision tree checks the next rule. The second rule states that if a SIP REGISTER message does not include a *Call-ID*, the message will be rejected by the VoIP server. Of course both the *CSeq* and the *Call-ID* header are mandatory SIP headers, according to RFC3261.

```
CSeq <= 0: REJ.
CSeq > 0
| Call-ID <= 0: REJ.
| Call-ID > 0
| | Replaces <= 0
| | | Accept_level <= 0
| | | | [..]
| | | Accept_level > 0: REJ.
| | Replaces > 0: REJ.
```

Figure 3.33: C4.5 tree, REGISTER messages, extract 1.

Section 3.5.5 described that the Java tool SIPParameterShuffler, used to gen-

erate random SIP messages, uses a probability of 97% for all mandatory SIP headers. Therefore some created messages did not include mandatory headers. The idea behind that approach was to check if Babel-SIP finds these faulty messages. Figure 3.33 shows that Babel-SIP was not only able to find these faults, but given the fact that these rules are the first two rules of the decision tree, Babel-SIP treats such non-RFC-conform faults as highly important.

If an incoming SIP REGISTER message does include a *CSeq* and a *Call-ID* header, the next rule of the decision tree states that incoming messages will be rejected if a *Replaces* header is included in the message. The *Replaces* header is usually used in e.g.: Attended Call Transfer scenarios, replacing one participant with another. Therefore the *Replaces* header should look like a *To* or *From* header. In the created SIP REGISTER messages, instead of including a SIP address, a Call-ID-like entry is used to simulate a wrongly developed soft phone. Therefore the used *Replaces* header leads to a rejection of the message, because of wrong parameters and values.

The fourth rule states that an incoming message will be rejected if the *level* parameter is used within an *Accept* header.

Figure 3.34 shows that there can also be header parameter combinations that obviously lead to problems. E.g.: An incoming message does not include problems that could have already triggered a rule to reject the message, but the *application/x-private* parameter is used within the *Accept* header and the *mobility* parameter is used in the *Contact* header, then a message will be rejected as well. But Figure 3.34 also shows that if the *application/x-private* parameter is used within the *Accept* header and the *mobility* parameter is not used in the *Contact* header, the message will still be rejected if the *ttl* (time to live) parameter is used in the *Via* header. In case the *ttl* paramter is not used within the *Via* header and the *Subscription-State* header is not used the message will be accepted, otherwise it will be rejected.

Figure 3.35 shows the third interesting part of the decision tree. If all previously discussed problems do not occur in an incoming message, and a *Min-Expires* header is used in combination with a *line* paramter within a *Contact*

header, the message will be rejected.

```
[..]
| | | | | Accept_application/x-private <= 0
| | | | | | [..]
| | | | | Accept_application/x-private > 0
| | | | | | Contact_mobility <= 0
| | | | | | | Via_ttl <= 1
| | | | | | | | Subscription-State <= 0: ACC.
| | | | | | | | Subscription-State > 0: REJ.
| | | | | | | Via_ttl > 1: REJ.
| | | | | | Contact_mobility > 0: REJ.
```

Figure 3.34: C4.5 tree, REGISTER messages, extract 2.

Figure 3.36 shows the fourth extract of the REGISTER decision tree. If a message includes the *received* paramter of the *Via* header and the *q* parameter of the *Contact* header, it will be marked as rejected.

```
[..]
| | | | | | | Min-Expires <= 20
| | | | | | | | [..]
| | | | | | | Min-Expires > 20
| | | | | | | | Contact_line <= 0
| | | | | | | | | Allow_ACK <= 0: ACC.
| | | | | | | | | Allow_ACK > 0: REJ.
| | | | | | | | Contact_line > 0: REJ.
```

Figure 3.35: C4.5 tree, REGISTER messages, extract 3.

Figure 3.37 shows that if a *Reply-To* header is used in an incoming SIP message and the *sip.instance* parameter is used within the *Contact* header, the message will be rejected.

```
[..]
| | | | | | | | Via_received <= 0
| | | | | | | | | [..]
| | | | | | | | Via_received > 0
| | | | | | | | | Contact_q <= 0: ACC.
| | | | | | | | | Contact_q > 0: REJ.
```

Figure 3.36: C4.5 tree, REGISTER messages, extract 4.

Figure 3.38 shows that messages that do not include a *Via* header will be rejected. As mentioned in Section 3.5.5, the *Via* header is not a mandatory header according to RFC 3261, but the tested VoIP proxy rejects all messages

```
[..]
| | | | | | | | Reply-To <= 0
| | | | | | | | | [..]
| | | | | | | | Reply-To > 0
| | | | | | | | | Contact_sip.instance <= 0
| | | | | | | | | | Allow-Events_hold <= 0: ACC.
| | | | | | | | | | Allow-Events_hold > 0
| | | | | | | | | | | Request-Line_user <= 0: ACC.
| | | | | | | | | | | Request-Line_user > 0: REJ.
| | | | | | | | | Contact_sip.instance > 0: REJ.
```

Figure 3.37: C4.5 tree, REGISTER messages, extract 5.

that do not include a *Via* header.

```
| | | | | | | | | | Via <= 0: REJ.
| | | | | | | | | | Via > 0: ACC.
```

Figure 3.38: C4.5 tree, REGISTER messages, extract 6.

Figure 3.39 shows that a message where the *Privacy* header is set to *none* and the *Error-Info* header is included, the message will be rejected.

```
[..]
| | | | | | | | | | Privacy_none <= 0
| | | | | | | | | | | [..]
| | | | | | | | | | Privacy_none > 0
| | | | | | | | | | | Error-Info <= 0: ACC.
| | | | | | | | | | | Error-Info > 0: REJ.
```

Figure 3.39: C4.5 tree, REGISTER messages, extract 7.

Appendix A shows the entire REGISTER decision tree. Some rules state right away which headers have to be included in a REGISTER message. The tree also shows some unexpected parameter combinations that lead to rejected messages.

Besides the rules, Appendix A also shows how many messages were accepted or rejected by which rule. The suggested procedure for either VoIP proxy developers or VoIP phone developers should be to look through the decision tree and focus on the rules which mark the most messages as rejected. Of course the rules on top of the tree are the most important ones, but parameter combinations that lead to an enormous amount of rejected messages, should also be taken very seriously.

C4.5 decision tree, INVITE experiments

The decision tree derived at the end of the INVITE experiments is even a little bit larger than the tree derived at the end of the REGISTER experiments. Figure 3.40 shows the first few rules of the tree. The first rule states that incoming INVITE messages that do not include a *Domain* within the *To* header will be rejected by the server. Of course the *To* header is a mandatory header according to RFC 3261, and the *Domain* is an important parameter of that header, so it is no surprise that the mentioned rule is very important. The second and third rule have already been discussed in the previous section, stating that messages that include the *Replaces* header or the *application/x-private* value within the *Accept* header, will be rejected as well.

```
To_IP <= 0: REJ.
To_IP > 0
| Replaces <= 0
| | Accept_application/x-private <= 0
| | | [..]
| | Accept_application/x-private > 0: REJ.
| Replaces > 0: REJ.
```

Figure 3.40: C4.5 tree, INVITE messages, extract 1.

Figure 3.41 shows that messages that include a *Domain* within the *To* header and do not include the *Replaces* header and the *application/x-private* value within the *Accept* header, and also include the *non-urgent* value within the *Priority* header will be rejected if the INVITE message does also include the *rport* parameter within the *Via* header. If the *rport* parameter is not present in an INVITE message, the message will still be rejected if the *Content-Expires* header is included.

```
[..]
| | | Priority_non-urgent <= 0
| | | | [..]
| | | Priority_non-urgent > 0
| | | | Via_rport <= 0
| | | | | Contact_expires <= 1: ACC.
| | | | | Contact_expires > 1: REJ.
| | | | Via_rport > 0: REJ.
```

Figure 3.41: C4.5 tree, INVITE messages, extract 2.

Figure 3.42 shows a very strange behavior. The two rules state that if a

message does not include a *Domain* within the *From* header, a message will still be accepted if the *conference* value is included within the *Allow-Events* header. Usually the *Domain* of the *From* header should be mandatory, but the INVITE message will only be rejected, if the *Domain* of the *From* header and the *conference* value of the *Allow-Events* header are missing. There was no opportunity for me to talk to the developers of the tested VoIP server to explain that phenomenon.

```
[..]
| | | | From_IP <= 0
| | | | | Allow-Events_conference <= 0: REJ.
| | | | | Allow-Events_conference > 0: ACC.
| | | | From_IP > 0
| | | | | [..]
```

Figure 3.42: C4.5 tree, INVITE messages, extract 3.

Figure 3.41 showed a rule regarding the value *non-urgent* within the *Priority* header. Figure 3.43 shows one of the next rules, which deals with the value *urgent* within the *Priority* header. The rule states that messages which include the *urgent* value within the *Priority* header will be rejected if the message also contains the *transport* parameter within the *request line*.

```
[..]
| | | | | Priority_urgent <= 0
| | | | | | [..]
| | | | | Priority_urgent > 0
| | | | | | Request-Line_transport <= 0: REJ.
| | | | | | Request-Line_transport > 0: ACC.
```

Figure 3.43: C4.5 tree, INVITE messages, extract 4.

Finally Figure 3.44 shows that INVITE messages that include the *Timestamp* header will be rejected if the *hold* value is used within the *Allow-Events* header.

As mentioned before Appendix B shows the entire decision tree with all the rules used to decide whether an incoming message will be accepted or rejected by the server.

```
[..]
| | | | | | | Timestamp <= 0
| | | | | | | | [..]
| | | | | | | Timestamp > 0
| | | | | | | | Allow-Events_hold <= 0: ACC.
| | | | | | | | Allow-Events_hold > 0: REJ.
```

Figure 3.44: C4.5 tree, INVITE messages, extract 5.

3.7 Conclusion

Working with a commercial VoIP server was really interesting. Creating test cases that help to automate a testing process of a commercial SIP proxy helped to understand the SIP protocol in every detail and was a great experience. Of course the SIP dialect problematic evolved to the focus of my work. Especially creating Java test tools that simulated VoIP devices with different SIP dialects was very important to automatically generate a great set of SIP messages for the Babel-SIP experiments.

With Babel-SIP we were able to show that with our approach using C4.5 decision trees we can reduce the rejection rate of faulty SIP messages. Its main use can therefore be in the transient phase between creating a new SIP stack implementation or a whole new proxy, and the final release of a 100% reliable proxy version. Of course the idea can be re-used during the implementation of systems using other new protocols as well.

By carrying out numerous experiments, we have demonstrated that with our automatic, self-learning SIP message translator great improvements can be achieved. Additionally we have shown that the resulting decision trees indeed provide good insight into the faulty behavior of either the SIP parser or the SIP clients and phones themselves. As a consequence, the decision trees can be used by SIP programmers to remove implementation bugs [HNHH08].

Chapter 4

Automatically adapting software to specific hardware

This chapter presents an optimization tool that finds the optimal number of threads for multi-thread software. Threads, are assumed to encapsulate concurrent executable key functionalities, are connected through finite capacity queues, and require certain hardware resources.

This chapter also shows how a combination of measurement and calculation, based on *Queueing Theory*, leads to an algorithm that recursively determines the best combination of threads, i.e. the best configuration of the multi-thread software on a specific host.

Section 4.1 deals with the basic idea and the motivation behind my work. Section 4.2 presents related work using autonomic approaches to find an optimum solution for a given host. The described algorithm proceeds on the directed graph of a queueing network that models the used software, therefore Section 4.3 describes queueing networks and especially M/M/k/B graphs. Section 4.4 introduces software that is based on queueing networks and describes commercial software developed by our industrial partner. Optimization towards hardware consolidation, where CPU cores, memory, disk space and speed, and network bandwidth are constraints, but also towards throughput is described. Two experiments on different SUN machines verify the optimization approach. Section 4.5 describes the developed autonomic test tool to improve the performance of software using queueing networks.

4.1 Basic Idea and Motivation

The trend to many cores inside CPUs enables software engineering towards concurrency. Software is split up in atomic actions that can run in parallel. This may considerably speed up computation, but also causes extra overhead through thread coordination for both, the operation system and also the software engineer.
Developing software made of several threads is much more complicated than creating an old-fashioned single thread software. One fundamental problem of the developer might be to find the right strategy for determining the appropriate number of threads.

My approach aims to solve the problem of finding the best number of threads under two aspects: with optimization towards consolidation, the goal is to find the optimal number of threads for a given external arrival rate. With optimization towards throughput, the idea is to increase the arrival rate and find the maximal external arrival rate the system can cope with.

This work discusses the approach to use a combination of analytical modeling techniques, measurements and simulations to find an optimal configuration of threads on a given host by iteratively increasing the number of threads.

4.2 Self-Adaptive Systems and Related Work

The goal of the second part of my work is to improve the performance of software system, by finding the optimal configuration of the system for a given host. The software should therefore optimize itself on a specific hardware platform. This section presents work related to that goal, where systems are trying to achieve the best possible performance on a given hardware system.

FFTW (fast Fourier transform) [FSJ05] is a free-software library that computes the discrete Fourier transform (DFT) and its various special cases. It uses a planner to adapt its algorithms to the hardware in order to maximize performance. The *FFTW* planner works by measuring the actual run time of many different plans and by selecting the fastest one. The input to the planner is a problem, a multidimensional loop of multi-dimensional DFTs. The plan-

ner then applies a set of rules to recursively decompose a problem into simpler sub-problems of the same type. *Sufficiently simple* problems are solved directly by optimized, straight-line code that is automatically generated by a special-purpose compiler.
In *FFTW*, most of the performance-critical code was generated automatically by a special-purpose compiler, called *genfft* that outputs C code. [Fri04] describes that compiler in detail.

The project *SPIRAL* [PMJ+05] aims at automatically generating high performance code for linear digital signal processing (DSP) transforms, that is tuned to a given platform. *SPIRAL* implements a feedback-driven optimizer that *intelligently* generates and explores algorithmic and implementation choices to find the best match to the computer's micro-architecture.
SPIRAL uses a formal framework to efficiently define alternative algorithms and implementations of the same transform and translate them into code. The different algorithms and implementations are therefore formulated as an optimization problem. Search and learning techniques are then used to find the one alternative implementation that is best tuned to the desired platform while visiting only a small number of alternatives.
The *SPIRAL* code generation system therefore replaces a human expert in both DSP mathematics and code tuning. It autonomously explores algorithm and implementation choices, optimizes at the algorithmic and at the code level, and exploits platform-specific features to create the best implementation for a given computer.

By identifying rewriting rules [BFL+06] extends *SPIRAL* by the automatic parallelization of FFTs given as mathematical formulas. Expensive compiler analysis is thereby replaced by simple pattern matching. As part of the program generation and optimization system *SPIRAL*, [BFL+06] introduces a formal framework for automatically generating performance optimized implementations of the discrete Fourier transform (DFT) for distributed memory computers. By integrating rules to rescale the computation to a different number of CPUs during the computation in *SPIRAL*'s rewriting system, automatic search mechanism can find the fastest among alternatives and generate DFT MPI code that is adapted to a given computing platform.

ATLAS (Automatically Tuned Linear Algebra Software) [WD98] aims at automatically generating code that provides the best performance for matrix multiply on a given platform. In general, matrices will be too big to fit into cache. Depending on the platform *ATLAS* divides matrices into blocks and uses a block-partitioned algorithm for the matrix multiply. That way it is still possible to arrange for the operations to be performed with data for the most part in cache. A code generator automatically creates code that uses timings to determine the correct blocking and loop unrolling factors to perform an optimized on-chip multiply.

Self-optimizing computing systems that can optimize their own behavior on different platforms without manual intervention, like *FFTW* and *ATLAS* need values for hardware parameters, e.g. the capacity of the L1 cache (*ATLAS*). [YPS05] introduces *X-Ray*, a system for implementing micro-benchmarks to measure such hardware parameters automatically.

In [OK07] an algorithm is developed for finding the nearly best configuration for a Web system consisting of one HTTP server, one application server, and a database server. Up to 500 emulated clients generate traffic for various trading operations like *buy* or *sell*. The Web system's configuration parameters are *MaxClients*, *ThreadPoolMax*, *HeapMax*, and *PSCacheSize* for testing the response time. Instead of trying all combinations, they start with a typical setting and let their algorithm find the next better setting. If such setting is found, the measurement is repeated until there are enough samples, upon the algorithm decides that setting is the new optimum. Only better settings are measured up to full accuracy, thus the algorithm avoids measuring initially worse settings. The algorithm works with a confidence interval and a regression function to decide if a setting is better or worse than the current and to predict the performance of settings based on previously measured settings. Since our approach also uses a metric to rank our system's configurations, the work [OK07] goes in the same direction, whereas our problem is twofold. First, we have a strong iterative process where increased load affects the front end of our system. Second, a modification at this point affects the performance of succeeding elements and causes new configurations. Our idea is to find a path through configurations of the system by modifying over-utilized parts of the

system.

The technology *Grand Central Dispatch* (GCD) by Apple enables to use multi-core processors more easily. Firstly released in the Mac OS X 10.6, GCD is implemented by the library *libdispatch*. GCD is a scheduler for tasks organized in a queuing system and acts like a thread manager that queues and schedules tasks for parallel execution on processor cores. The functionality is a layer between operation system threads and the application. The developer must not take care about traditional thread handling; taking complexity off. Source code is grouped by GCD-specific commands called *closures*. GCD then creates threads for every closure. Created threads are put into queues. Per core there are up to 3 global queues with different priorities. Serial queues are conceptually situated before a global queue. With a serial queue tasks can be executed consecutively. In this work I also intend to use threads in a way that available cores of a processor are optimally used. The analogy to GCD is that I also want to find a currently best thread configuration based on the load, therefore a method for determining the best combination of threads is described and implemented.

In [BPI03] a queueing network model with finite/single capacity queues and blocking after service discipline is used to model software architectures; more exactly the synchronization constraints and synchronous communication between components of them. The information flow (trace) between components is analyzed to identify the kind of communication (fork, join) and reveal the interaction pairs among components that enables to model a queueing network. This way a performance model for specific software architecture can be derived. But this work considers consider asynchronous communication between component-threads with (in)finite capacity queues.

[Xu08] introduces a diagnostic assistant (Performance Booster) for *"rule-based automatic software performance diagnosis and improvement"*. The performance booster will either modify the planned runtime configuration or the software design itself. With the help of performance measurement, the Performance Booster uses diagnostic rules to find saturated resources and resources with critical response times. The algorithm of the performance booster works

in cycles and it begins with one possible design and ends up with a set of new candidate designs and their evaluation. To find the best configuration for a model, the Performance Booster increases the multiplicity of bottleneck resources step by step until the bottleneck has been removed or the hardware limits have been reached.

[TDB08] tries to develop dense linear algebra algorithms for hybrid multicore and GPU architectures. [TDB08] states that

> *"a fundamental concept in programming current parallel architectures is the flexible control over the data and execution flow. Algorithms and their execution flows can be represented as Directed Acyclic Graphs (DAGs), where nodes represent the tasks and the edges the dependencies among them."*

Task splitting and scheduling results in an algorithm that reduces the communication between a multicore and a graphics processor, making them more balanced and efficient.

[MM10] presents *PARSY* (*"Performance Aware Reconfiguration of software SYstems"*), which is used to re-configure degradable software systems in a performance-aware way, by tuning individual software components. The approach uses the system's response time as decisive performance metric. The system uses a *"Queueing Network (QN) performance model"* to find new configurations and estimate the response time of these configurations. [MM10] states that a

> *"controller can estimate the system response time for different configurations by using a single-class, closed QN model."*

[KP11] presents Perpetuum, *"a novel operating-system-based auto-tuner that is capable of tuning applications cooperatively at run-time."* The approach shows that Perpetuum *"is integrated into the Linux OS kernel"*, *"monitors workloads and adapts tuning parameter values of all running programs to improve performance."*

4.3 Background: Queueing Networks

Software using a flow chart-like structure sending units of work from one processing node to another is commonly modeled as a queueing network. Markov chains are used as state equations to model queueing networks. According to [BGdMT06] a Markov chain consists of a set of states and a set of labeled transitions between the states.
Usually states can model or describe certain conditions. In this work the entire software is built as a queueing network and is used to handle incoming messages. States therefore stand for certain types of tasks. Incoming messages are passed through the queueing network, visiting different states where various tasks are executed.
Incoming jobs or tickets are passed from one state or node to an other. Each node has a queueing buffer of finite or infinite size to hold or store tickets. The node then takes the first ticket out of the queue, executes its task and sends the ticket through transitions along the queueing network. Each node or state can furthermore have one or more identical servers. Each server can handle one ticket at a time and therefore multiple servers are used in parallel to process multiple tickets.
One server is busy as long as the task has been executed and the ticket left through an outgoing transition. If all servers of one node are busy, the new incoming ticket is stored in the queue. If an incoming ticket cannot be stored because of a full queue, the ticket is lost.

Figure 4.1 shows one node of a queueing network having m identical servers. Arriving jobs go into the queue and are then taken out by a server. After they are processed they leave the node. Figure 4.2 shows that incoming tick-

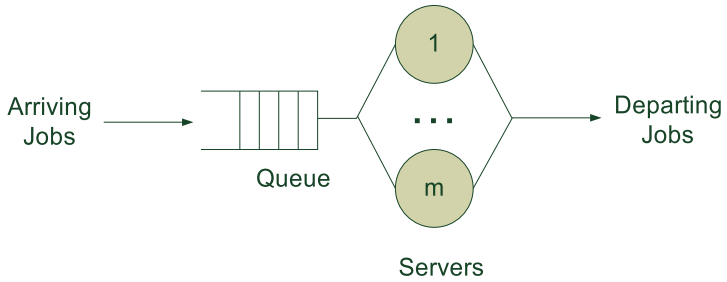


Figure 4.1: Single Station Queueing (see [BGdMT06]).

ets arrive with rate λ. Furthermore every node and server finishes its work in μ time. The arrival rate λ and the service rate μ are two very important parameters for calculating performance measures.

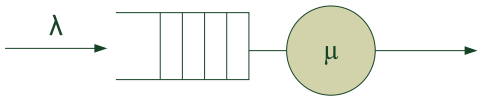


Figure 4.2: Arrival and service rate.

Queueing networks usually consist of a number of nodes that are connected to each other. Therefore jobs can visit any node at some point. Also, a job can be transferred back to the node it just left.

Queueing networks can be classified as *open* or *closed*. Figure 4.3 shows a closed queueing network in which jobs can neither enter, nor leave the network. A closed queueing network always deals with a constant number of jobs within the network. A queueing network can also be called *closed* if, whenever a new jobs enters the system, another job leaves the system at the same time.

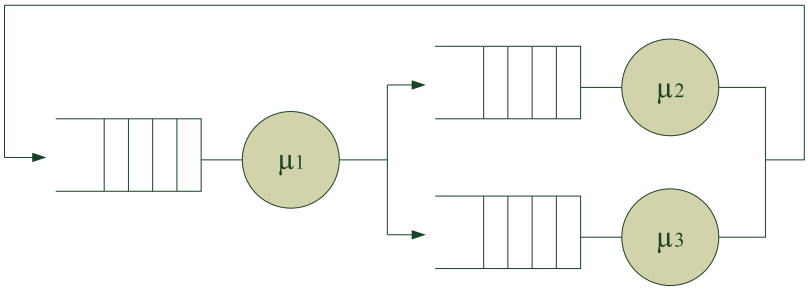


Figure 4.3: A closed queueing network (see [BGdMT06]).

This work concentrates on open queueing networks. Figure 4.4 shows an open queueing network in which jobs enter the network from a certain source from the outside. Also, jobs can leave the system at every node leading into a sink.

Section 4.3.1 describes the queueing network used in this work in more detail. Section 4.4 presents performance metrics and also describes the software implemented by our industrial partner that is based on a data flow queueing

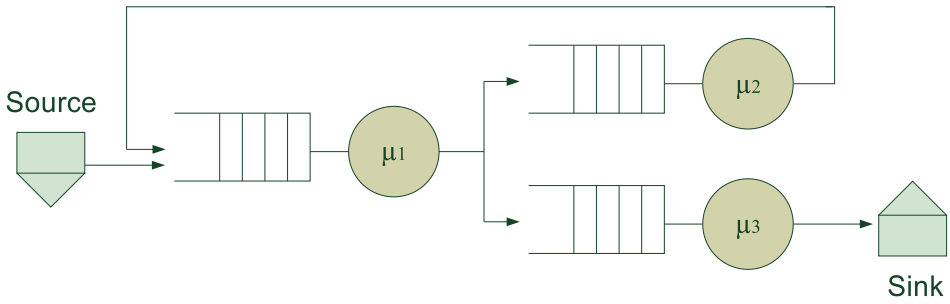


Figure 4.4: An open queueing network (see [BGdMT06]).

network.

4.3.1 Background: Kendall's Notation

To properly describe queueing networks, the following notation, known as Kendall's Notation, is usually used:

$$A/B/C/K/N/D$$

A ... the first indicator A describes the distribution of the inter-arrival times.

B ... the second indicator B describes the distribution of the service times.

C ... the third indicator C describes the number of servers.

K ... the fourth indicator K describes the capacity of both queue and server. If the number of jobs in the system reaches K, new arriving jobs are lost.

N ... the fifth indicator N describes the size of the calling source.

D ... the sixth indicator D describes the priority order in which jobs in the queue are handled.
In my dissertation a large number of independent external sources are sending tickets to the queueing network. Therefore, a Poisson process results in a Markovian inter-arrival distribution ($M/././././.$).

The service time of each node and server is independent from the arrival pro-

cess, resulting in an exponential service time distribution. Therefore the service time distribution in this work is also Markovian (*./M/././.*), or a general distribution (*./G/././.*), respectively.

Since the overall goal is to automatically change the number of threads for each node, the number of servers of the system, equals the number of cores for a given platform (*././k/./.*). Section 4.4 discusses this issue in more detail.

The size of the buffers in the system is adjustable (*./././B/./.*).

The size of the calling source for our particular system is assumed to be infinite, therefore the fifth indicator can be omitted.

The priority in which jobs are taken out of the queue is FIFO (first in, first out). Therefore, jobs are taken out of the queue in exactly the same order as they have arrived.

In summary, the system used in this worked can be described as a M/M/k/B queueing network according to Kendall's Notation (see [BGdMT06] and [Wik10b]).

4.4 Software using queueing networks

During the work on my dissertation I worked with commercial software that is based on a queueing network. This chapter describes the model and the functions of this software and also the optimization goals.

4.4.1 The model

The system under suggestion is software called *Data Flow Engine* (DFE) for processing event-based data in form of packets according to the *Diameter* protocol defined in RFC 3588 (see [CLG+03]) maintained by the *The Internet Engineering Task Force* (IETF).

The queueing network software receives, extracts, converts, and stores incoming Diameter-packets. The functionality of each operation is internally represented by interconnected nodes. Queues buffer the output for succeeding

nodes. Since these nodes technically are lightweight processes (threads), they can be replicated for splitting the load.

The final aim of the queueing network software is to provide software that covers some self-⋆ properties [BCDW04, KASH05, VR07] like self-monitoring and self-configuring, thus self-adaption.

The proposed tool could measure the performance of a host at runtime and then decide how many nodes/threads are necessary to fulfill a given optimization goal. Also, detection of decreasing load could lead to fewer threads and therefore less active CPU cores.

The queueing network software is modeled as an open queueing network consisting of four nodes. A node models an atomic action, but several instances of a node can exist as threads. Defined nodes are:

- The *Decoder* node takes packets from its queue, extracts the data, and forwards it to the next node's queue. Hence, the extraction can be done concurrently by several nodes. A *Decoder* can be replicated and each *Decoder* forwards the extracted data to the same queue.
- The *Converter* node takes extracted data from its queue and converts it into a format appropriate for storing. The *Converter* forwards the ticket to the *Serializer* and *Feeder* node, thus the extracted and formated data is persisted always twice. Also conversion can be done concurrently by several threads of the *Converter*.
- The *Serializer* node takes data from its queue and stores it to disk. Several *Serializers* may write to disk concurrently.
- The *Feeder* node takes data from its queue and sends it to a database. Since the database is assumed to be capable to provide a connection pool, many *Feeders* may send in parallel.

All nodes require certain CPU time, memory, disk space and speed, and network bandwidth, thus, the replication of nodes is bounded. Starting with an initial configuration with one thread per node, the goal is to find the optimal

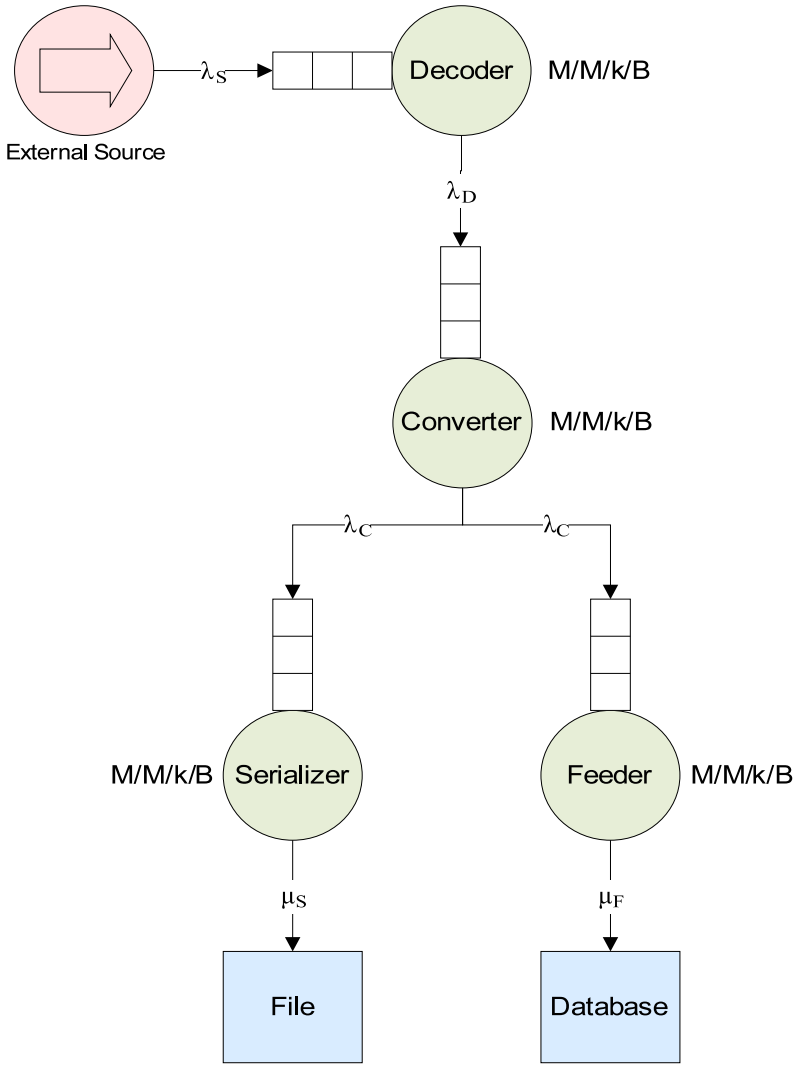


Figure 4.5: The Data Flow Engine (DFE) modeled as queueing network.

configuration of threads for a certain host. The structure of the software is shown in Figure 4.5.

4.4.2 Usage

As mentioned before the Data Flow Engine receives incoming Diameter-packets. It is used for an automated transfer of files or real-time data streams. The operational area of the queueing network software could be to store VoIP call data for billing purposes. Basically, the queueing network software receives Diameter-packets from devices through an RMI interface, and then the packet

is sent through the queueing network for further processing.

Incoming Diameter-packets usually contain information about VoIP calls. *Start*, *Interim* and *End* messages are used to create individual calls. The *Start* Diameter-packet usually contains a SIP INVITE message, while the *End* Diameter-packet contains a SIP BYE message. The *Interim* Diameter packets are used in case of missing *Start* or *End* Diameter-packets. They are periodically sent from the devices to provide an opportunity for the exact billing of a certain call.

One goal of this dissertation was to create a Diameter Ticket-Generator for the purpose of creating a test tool for the queueing network software. As mentioned in the first part of my dissertation, there are a lot of different call scenarios when dealing with SIP VoIP calls. Figure 4.6 shows a basic call scenario where one call is represented by a *Start* Diameter-packet, a number of *Interim* Diameter-packets and one *End* Diameter-packet. The created Di-

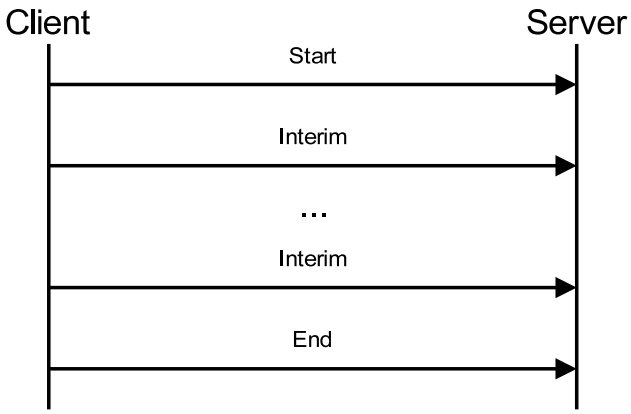


Figure 4.6: Diameter-packets: Call scenario

ameter Ticket Generator (see Section 4.5.4) tries to extend this scenario by randomly leaving out *Start*, *Interim* or *End* packets.

4.4.3 M/M/k/B Queues

The main idea of my work is to sequentially add new threads for over-utilized queueing network software nodes until an optimization goal is reached. For that reason, the queueing network, as base for an analytical model [Jai91,

Zuk09, BGdMT06, KM08] for describing the queueing network software, is shown in Figure 4.5 as an open queueing network consisting of 4 queues. The queueing network is defined as open in the sense that jobs come from an external source, are serviced by an arbitrary number of queues inside the network and eventually leave the network.

Further, the suggested queueing network is aperiodic, because no job visits a queue twice, the flow goes to one direction. Because no jobs are lost inside the queueing network, the overall service rate is equal to the overall arrival rate ($\mu = \lambda$). This kind of queueing network is called *Jackson network*. The queueing discipline is always *First-Come, First-Served* (FCFS).

An external source sends events with rate λ_S to the M/M/k/B *Decoder* queue whereas the arrival process (M/././.) is Markovian and follows a Poisson process [Agr02] with independent identically distributed (iid) and exponentially inter-arrival times $\frac{1}{\lambda}$, which postulates that the next arrival at $t + 1$ is completely independent from the arrival at t.

The service time $1/\mu$ of each server k is independent from the arrival process and iid and exponentially distributed (memoryless) with parameter μ and therefore the departure process (./M/./.) is also Markovian.

Note: The Markovian departure process is an assumption for convenience, but in reality might follow any general distribution.

The queue capacity (buffers) is defined by parameter B. An arrival reaching a full queue is blocked. This can be avoided by increasing the queue size B, decreasing the arrival rate λ, or increasing the number of servers k and thereby increasing the joint service rate μ. This holds also for the *Converter*, *Feeder*, and *Serializer* queues.

After the data is extracted by the *Decoder*, it is forwarded with rate λ_D to the *Converter* queue. The *Converter* then converts the data in file and database format and forwards it to the corresponding queues with rate λ_C. Thus, the data is duplicated among this path. The *Feeder* sends the data to the database

with rate μ_F whereas the *Serializer* writes data to disk with rate μ_S.
There is a race between λ and μ in the sense that under the assumption $\lambda \leq \mu$ for the utilization ρ and given number of servers k, the following must hold:

$$\rho = \frac{\lambda}{k\mu} \leq 1 \quad , \tag{4.1}$$

which leads to a stable system; all jobs can eventually be worked out. Unfortunately, due to different nodes, the bottleneck is the node where $\lambda > \mu$. For finding the best configuration of the queueing network, as shown in Figure 4.5, following performance measures [Jai91, Zuk09, BGdMT06, KM08] are considered for the queueing systems *Decoder*, *Converter*, *Serializer* and *Feeder*.
The utilization ρ (4.1) is the base for most other measures. The probability P_n of n jobs in the queueing system is given by

$$P_n = \begin{cases} \frac{(k\rho)^n}{n!} P_0 & \text{for } \; 0 \leq n < k \\ \frac{k^{k\rho^n} k!}{P_0} & \text{for } \; k \leq n \leq B \;\text{ and }\; B \geq k \end{cases} \tag{4.2}$$

where k is the number of servers, B the number of buffers (slots in the queue), and P_0 the probability of no jobs in the system which for $k = 1$ is

$$P_0 = \begin{cases} \frac{1-\rho}{1-\rho^{B+1}} & \text{for } \; \rho \neq 1 \\ \frac{1}{B+1} & \text{for } \; \rho = 1 \end{cases} \tag{4.3}$$

and for $k > 1$

$$P_0 = \left(1 + \frac{(1-\rho)^{B-k+1}(k\rho)^k}{k!(1-\rho)} + \sum_{n=1}^{k-1} \frac{(k\rho)^n}{n!} \right)^{-1} . \tag{4.4}$$

The expected number of jobs in the system E_s for $k = 1$ is

$$E_s = \frac{\rho}{1-\rho} - \frac{(B+1)\rho^{B+1}}{1-\rho^{B+1}} \tag{4.5}$$

and for $k > 1$

$$E_s = \sum_{n=1}^{B} np_n \tag{4.6}$$

where the expected number of jobs in the queue E_q for $k = 1$ is

$$E_q = \frac{\rho}{1-\rho} - \rho\frac{1+B\rho^B}{1-\rho^{B+1}} \tag{4.7}$$

and for $k > 1$

$$E_q = \sum_{n=k+1}^{B} (n-k)p_n \tag{4.8}$$

Since we have queues with finite buffers B, some traffic is blocked. The initial traffic that reaches the queueing network is not equal to the traffic that passes through the queueing network. Reduced to the single queueing system this means, that λ depends on a certain blocking probability P_b. This leads to the effective arrival rate λ'

$$\lambda' = \lambda(1 - P_b)$$

where the blocking probability $P_b = P_B$, i.e. the probability of B jobs in the queueing system. The loss rate ϵ is

$$\epsilon = \lambda P_B \tag{4.9}$$

and the effective utilization ρ' is

$$\rho' = \frac{\lambda'}{k\mu}$$

again with k servers. Next, the mean response time R of the queueing system is

$$R = \frac{E_s}{\lambda'} \tag{4.10}$$

and the mean waiting time W of a job in the queue is

$$W = \frac{E_q}{\lambda'} \tag{4.11}$$

Finally, the probability that the system is full is denoted by

$$P_k = \frac{\frac{(k\rho)^k}{k!}}{\sum_{j=0}^{k} \frac{(k\rho)^j}{j!}} \tag{4.12}$$

Configurations of the queueing system can be evaluated according to these measures. An optimization algorithm, presented in Section 4.5, determines the best configuration.

4.4.4 M/G/k/B Queues

As mentioned before, in my dissertation I assumed that the service times are exponentially distributed. In this section an M/G/k/B system will be briefly discussed and some important performance measures will be presented.

Note: [BGMT05] differentiates between the usage of G (general distribution) and GI (general distribution with independent interarrival times) within Kendall's notation.

As mentioned before, Kendall's notation states that M/G/k/B queueing systems have an Markovian arrival process (M/././.), the service times follow a general distribution (./G/./.), there are k servers (/././k/.) and the capacity of the entire system is limited to B (./././B).

[CCK+11] states that

> *"the service times form an independent and identically distributed (i.i.d.) sequence of nonnegative random variables with finite mean τ."*

This leads to a utilization of

$$\rho = \frac{a}{k}, \tag{4.13}$$

where

$$a = \lambda\tau. \tag{4.14}$$

The blocking probability $P_b = P_B$, i.e. the probability of B jobs in the queueing system.

And the probability that the system is full is denoted by

$$P_k = \frac{\frac{a^k}{k!}}{\sum_{j=0}^{k} \frac{a^j}{j!}} \tag{4.15}$$

4.5 Improving performance by using self-adaptive software

The following section tries to implement the aforementioned optimization goals for the queueing network software by using a combination of an analytical model, a measurement and a simulation approach.

The focus is to find the optimal configuration of multi-thread data-flow software for a specific host. A configuration is specified as a vector of n tuples with

$$(k_1 - \ldots - k_i - \ldots - k_n) \tag{4.16}$$

where k_i denotes the number of threads of node i. With the constraint of

$$\sum_{i=1}^{n} k_i \leq c \tag{4.17}$$

where c is the number of available cores on a specific host.

E.g.: The initial configuration of Decoder, Converter, Serializer and Feeder nodes (1-1-1-1) contains one thread per node.

4.5.1 Analytical Model

This section presents an offline-optimization tool, implemented in Java, for calculating the best configuration of nodes, based on a host's resources (e.g.: by Table 4.1).

Resource Type	**Quantitiy**
CPU cores (#)	32
Memory (MB)	32000
Disk space (MB)	100000
Disk speed (MB/s)	30
Network (Mbit/s)	100

Table 4.1: Resource offer of a hypothetical host.

The number of CPU cores is the upper bound for the number of nodes that can run concurrently. It is assumed that each node is executed as single thread on a dedicated core and since core sharing is ignored for now, so many nodes as free cores available are possible. Memory, disk space and speed, and network bandwidth are shared by all nodes and are the constraints for optimization. When a configuration exceeds given resources, the algorithm terminates.

Example requirements of *Decoder*, *Converter*, *Serializer*, and *Feeder* nodes are shown in Table 4.2. Resource requirements underly the following assumptions: The *Decoder* is listening on the network for new messages and decodes the data. Most decoding work is done in memory and only state information is stored on disk. The *Converter* transforms the data into another format and relies more on memory. Since the *Converter* only communicates with other nodes, no network traffic is produced. The *Serializer* writes data to disk and also requires no network. The *Feeder* holds connection to the database where aggregated data is stored.

Host and node resources must be known in advance based on experience or determined by hardware monitoring of a real system. What is up to the adaption tool is the detection of the service rate μ of each node on a specific host and configuration. Dependent on host performance and load, service rates vary. One additional thread can change service rates of other nodes.

Resource Type	Decoder	Converter	Serializer	Feeder
Memory (MB)	50	100	10	30
Disk space (MB)	5	5	5000	5
Disk speed (MB/s)	0.05	0.1	5	0.05
Network (Mbit/s)	2	0	0	1

Table 4.2: Hypothetical resource requirements of nodes.

An algorithm for finding the optimal configuration of nodes, depending on their utilizations, is outlined in Listing 4.1. The optimal configuration features only nodes with a utilization below the defined threshold. The recursive method *optimize()* is called as long as the queueing network can be extended. Before calculating a new configuration, service rates of each node are needed. In the first step of this solely analytical approach, hypothetical service rates are used.

```
1  optimize() {
     Measure service rates;
3    Calculate config;
     /* if optimization towards throughput */
5    if (extARInc > 0) {
       while (each node util. < limit) {
7        Increase external arrival rate;
         Calculate config;
9      }
     }
11   /* optimization towards consolidation */
     if (network is extented) optimize();
13   else {
       Get last valid external arrival rate;
15     Calculate OptConfig;
       return OptConfig;
17 }
```

Listing 4.1: Algorithm for finding the best configuration of nodes modeled as queueing network.

In the following sections measured service rates of artificial nodes (see Section 4.5.2) or exact service rates from a real software (see Section 4.5.4) are used. After service rates are determined, the performance measures (already intro-

duced above) (4.12), (4.11), (4.10), (4.9), (4.7) or (4.8), and (4.5) or (4.6) and over all the utilizations of nodes (4.1) are calculated. Depending on these utilizations the algorithm decides about the increase of the external arrival rate or the number of threads for a specific node.

Dependent on $extARInc$ for increasing the external arrival rate, two optimization goals exist:

- **Consolidation**. With optimization towards consolidation a desired arrival rate is given and the tool determines the minimum number of threads required to ensure that all nodes are below a predefined utilization threshold. With a constant external arrival rate only the number of threads of an over-utilized node with utilization $\geq limU$ $(0 < limU < 1)$ will be incremented for splitting load among available cores as evenly as possible.
- **Throughput**. With optimization towards throughput the system starts with the lowest arrival rate of one job per second and the tool increases the arrival rate stepwise until one node exceeds the predefined utilization threshold. Then the number of the corresponding threads is increased as long as the utilization is below the threshold. Again, the arrival rate will be increased and optimization goes on as long as no hardware limit is reached. With $0 < extARInc < 1$ the external arrival rate is increased for each new configuration by $extARInc$ as long as the maximum utilized node does not reach $limU$. For the maximum possible number of threads the highest possible throughput is determined.

The maximum number of iterations I, required to calculate the best configuration, is

$$I = N + 1 - C \tag{4.18}$$

where N is the number of cores, and C denotes the number of required cores in the initial configuration, i.e. one thread per core. This algorithm features $O(N)$, and the actual number of iterations is somewhat smaller than I, because the best configuration can be found earlier due to these termination conditions:

- None of the nodes is over-utilized according to $limU$.
- The number of threads of an over-utilized node cannot be increased due

to resource shortage.

Analytical Evaluation

Figure 4.7 shows, based on Tables 4.1 and 4.2, the outcome of optimization towards consolidation with $extARInc = 0$, $limU = 0.8$, external arrival rate $\lambda = 3$, $B = 100$, and hypothetical service rates μ_i $(0 \leq i \leq 3)$ for *Decoder*, *Converter*, *Serializer* and *Feeder* nodes of $\mu_0 = 4$, $\mu_1 = 1$, $\mu_2 = 2$, and $\mu_3 = 3$. The initial configuration has one thread of each node. The loss rate ϵ decreases

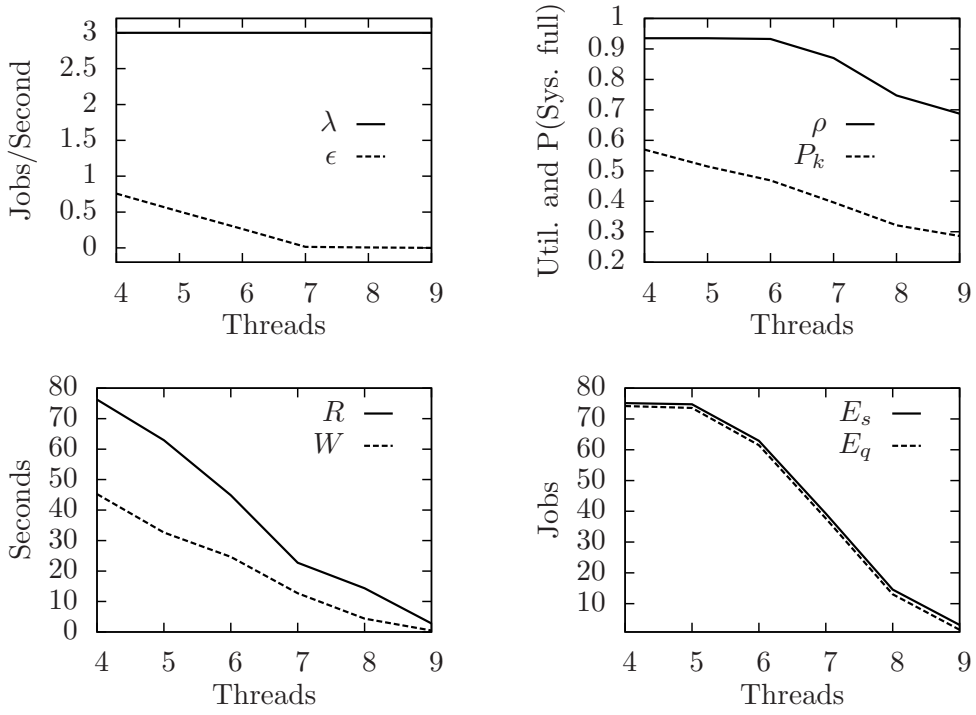


Figure 4.7: Optimization towards consolidation; plots represent queueing network-wide mean values over all threads.

as the number of nodes increases and reaches zero lost jobs per second after optimization is terminated. The system's utilization ρ is about 90% with 4 nodes and decreases clearly below the desired limit to approximately 70%, whereas the system's probablity to be full P_k decreases to approximately 30%. The system's response time R reaches a value below 3 seconds and the waiting time W a value around 0.5 seconds. Finally, the exptected number of jobs in the system E_s is reduced by approximately 95%, and the expected number of jobs in queues E_q by approximately 98%. The optimum configuration with 9 nodes

fulfills the optimization goal having utilization below $limU = 0.8$ for each node.

In the second step λ is increased by $extARInc = 0.1$ as long as the utilization of nodes is below $limU$. Figure 4.8 shows, based on Tables 4.1 and 4.2, the outcome of optimization towards throughput with $limU = 0.8$, initial external arrival rate $\lambda = 3$, $B = 100$, and hypothetical service rates μ_i $(0 \leq i \leq 3)$ for *Decoder*, *Converter*, *Serializer* and *Feeder* nodes of $\mu_0 = 4$, $\mu_1 = 1$, $\mu_2 = 2$, and $\mu_3 = 3$. As long as λ is constant, this works like optimization towards

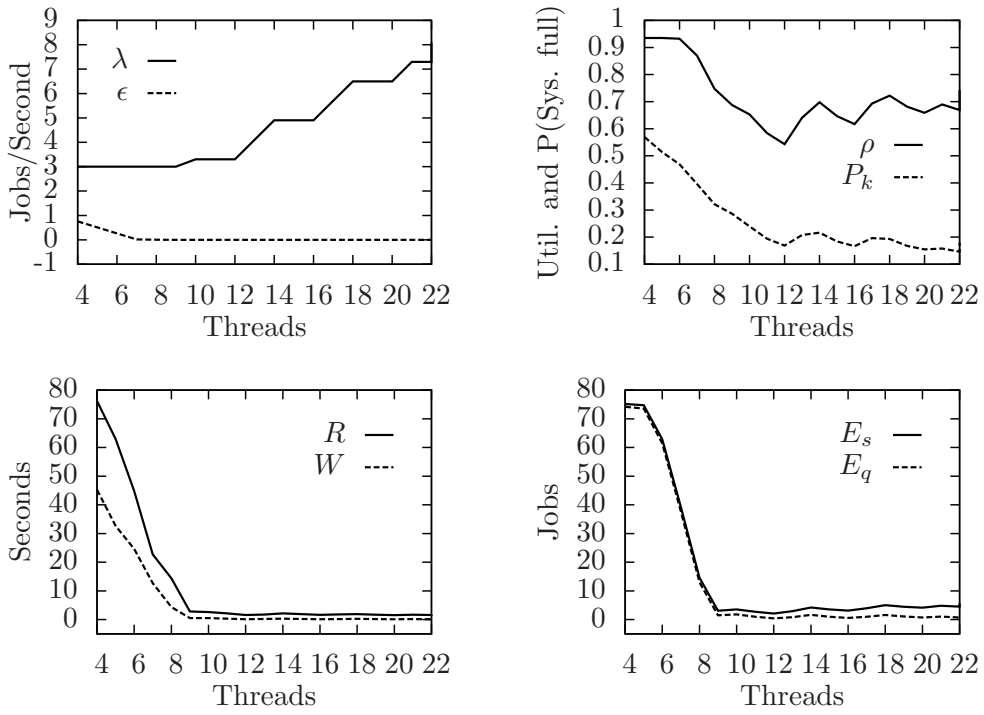


Figure 4.8: Optimization towards throughput; plots represent queueing network-wide mean values over all threads.

consolidation, i.e. only the number of nodes is increased until each node's utilization is below $limU$. For the next configuration, λ is increased by $extARfac$ and more arrivals cause again more utilization.

The initial configuration consists of 4 nodes; one thread per node. The best, bounded by host resources, possible configuration consists of 22 nodes. The loss rate ϵ decreases with rising number of nodes and reaches almost zero lost jobs per second. The utilization ρ of the system is about 90% with 4 nodes and

decreases clearly below the desired limit to approximately 70%, whereas the system's probablity to be full P_k goes down to approximately 16%. Further, the system's response time R reaches approximately 1.8 seconds, and the waiting time W approximately 0.2 seconds. Finally, the expected number of jobs in the system E_s is reduced by approximately 92%, and the expected number of jobs in queues by approximately 98%.

The optimal configuration with 22 nodes fulfills the optimization goal of each node's utilization below $limU = 0.8$, as proven by Table 4.3, not fully, because optimization must be stopped due to host resource shortage (disk speed). The

	Initial Config.		**Optimal Config.**	
Node	**Util.**	**Threads**	**Util.**	**Threads**
Decoder	0.75	1	0.675	3
Converter	1	1	0.81	10
Serializer	1	1	0.81	5
Feeder	0.9901	1	0.675	4

Table 4.3: Utilization and number of threads in initial and optimal configuration for optimization towards throughput.

mean utilization of the optimal configuration is approximately 0.74%. For a hypothetical host, nodes, and service rates we need exactly 3 *Decoders*, 10 *Converters*, 5 *Serializers*, and 4 *Feeders* to fulfill the optimization goal of a mean utilization below 80 % and reached a throughput of approximately 8.1 jobs per second.

Hitherto theoretical experiments are based on assumed service rates. In Section 4.5.2 a methodology for determining service rates for each node on a specific host under arbitrary configurations is explained. This allows to measure a host's performance for each calculated configuration and includes the effect of multi-threading and corresponding overhead.

4.5.2 Measurement

In the previous Section 4.5.1 a hypothetical host was the testbed for optimization towards consolidation and throughput in terms of threads. This section presents two experiments on real multi-core machines. The number of cores

as well as the available amount of memory is detected before measuring the actual service rates of nodes under arbitrary configurations. The optimization tool is now twofold: after a preparation phase, the measurement module gets the service rates and then the calculation module performs as already shown in Section 4.5.1, only this time real service rates are used instead of assumed ones.

For that purpose different node types were created, simulating work to consume different resource requirements. The following list describes work that may be done within a node:

- Sorting an Integer array (memory, CPU)
- Searching through an Integer array (memory, CPU)
- Writing data to a file (memory, CPU, disk)
- Searching through a file (memory, CPU, disk)
- String manipulation (memory, CPU)
- Writing data into a database (memory, CPU, disk, network)
- Searching through a database (memory, CPU, disk, network)
- Calculating mean values of an Integer array (memory, CPU)
- Creating random numbers (memory, CPU)
- ...

For example: to simulate the nodes of the queueing network software (see Figure 4.5), four different node types were used:

- The work of the simulated *Decoder* node is to create random numbers. For each ticket 1.000 random numbers are created to use memory and CPU resources.
- The work of the simulated *Converter* node is to sort an Integer array. For each ticket an Integer array of 50.000 random numbers is sorted to use greater memory and CPU resources.

- The work of the simulated *Serializer* node is to write data to a file. For each ticket the content of the ticket is written to a file to use memory, CPU and especially disk resources.
- The work of the simulated *Feeder* node is to write data into a database. For each ticket the content of the ticket is written into a table within an Oracle database to use memory, CPU, disk and possibly network resources.

The idea behind the proposed approach is to have a number of different tasks that can be applied to different nodes if, for example, the flow chart of the queueing network software changes, or a different queueing network based-software will be tested. In this work, the four mentioned nodes, *Decoder*, *Converter*, *Serializer* and *Feeder* were defined. To get the service rates for each of these nodes on the tested hosts, the tasks for each node were started repeatedly. For each individual execution, the service time was measured and a mean service rate was calculated. Listing 4.2 shows the Java code for the *Feeder* node, which is invoked repeatedly and writes one ticket into the database.

```
1 public static void sendTicketToDB(Connection conn) throws SQLException
    {
    Statement statement = conn.createStatement();
3   DateFormat sdf = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss");
    String s =
5     "INSERT INTO CDR "
        + "VALUES ('test_session_1', 32, TIMESTAMP sdf, 'alice@kcc.net
            ', 'bob@kcc.net', 1)";
7   statement.executeUpdate(s);
    statement.close();
9 }
```

Listing 4.2: Feeder node: task execution.

To calculate the service rate for the mentioned *Feeder* node, two approaches can be used. On the one hand, each method invocation is measured separately, resulting in i service times for every single task. Listing 4.3 shows the second approach, were just the start and end time is measured, the method is invoked i times and the entire simulation time is divided by i invocations, again resulting in a mean service rate for the given node.

```
1 double startTime = System.nanoTime();
  for (int i = 0; i < 10000; i++) {
3   sendTicketToDB(conn);
  }
5 double endTime = System.nanoTime();
  double simTime = endTime - startTime;
7 double servicetime = simTime / i;
  //service rate in ms
9 double servicerate = 1000000 / simTime;
```

Listing 4.3: Feeder node: task invocation and service rate calculation.

The first host *Goedel* is a SUN Fire v40z with four dual-core AMD Opteron processors Model 875, each core at 2.2 GHz, has 24 GB of main memory, five 300 GB Ultra320 SCSI HDs, 10/100/1000 Mb/s Ethernet, and runs Linux 2.6.16.60-0.42.7.

Figure 4.9 shows optimization towards consolidation on host *Goedel*. The

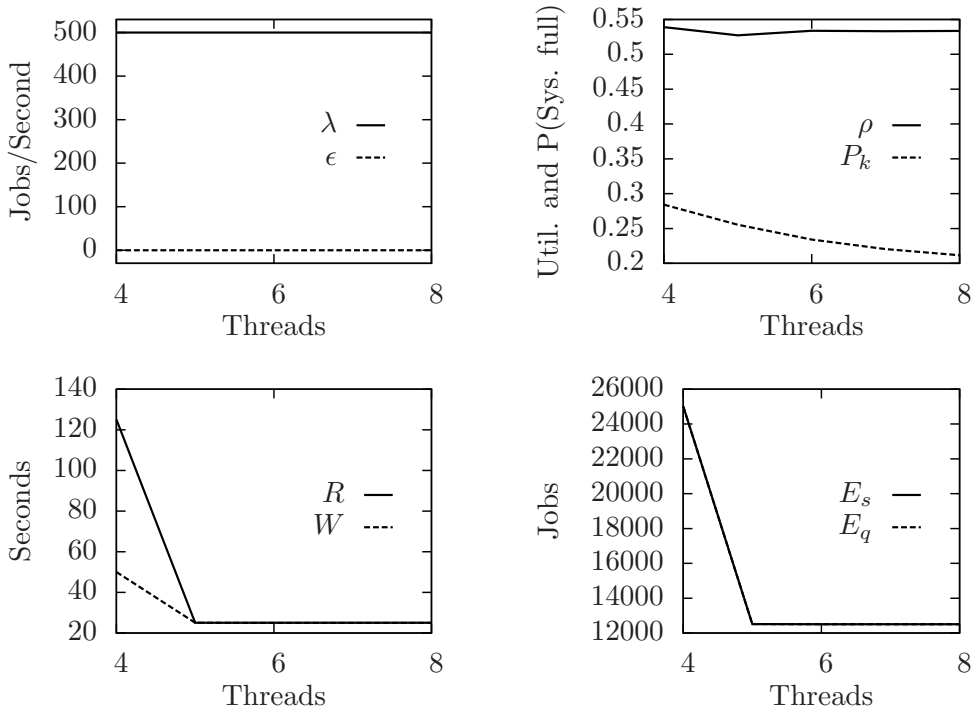


Figure 4.9: Optimization towards consolidation on host *Goedel*.

arrival rate is fixed to $\lambda = 500$ jobs per second. At 4 nodes there is almost

no loss rate ϵ and a mean utilization of approximately $\rho = 0.54$, whereas two nodes exhibit a utilization of $\rho = 1$ as shown in Table 4.4. At 8 nodes the mean utilization is approximately $\rho = 0.53$ where one node still exhibits $\rho = 1$. Table 4.4 shows the improvement from 4 to 8 nodes. Due to resource shortage on

	Initial Config.		**Optimal Config.**	
Node	**Util.**	**Threads**	**Util.**	**Threads**
Decoder	1	1	0.9641	2
Converter	0.1481	1	0.1606	1
Serializer	0.008	1	0.0092	1
Feeder	1	1	1	4

Table 4.4: Utilization and number of threads in initial and optimal configuration for optimization towards consolidation on host *Goedel*.

host *Goedel*, the number of nodes cannot be increased boundlessly. Because of database I/O, the *Feeder* is the natural bottleneck of the system. The rate at which data can be sent to the database cannot be increased by more threads. The possible rate is shared by all *Feeder*-threads.

Nevertheless, the probability of a full system goes down from initially approximately $P_k = 0.28$ to approximately $P_k = 0.21$. Both, times and jobs drop down at 5 nodes due to the second *Decoder*. Response and waiting time go down to $R \approx W \approx 25$ seconds. Expected jobs in the system and queues drop down to $E_S \approx E_q \approx 12506$ jobs. This shows how queueing helps alleviating a bottleneck.

Figure 4.10 shows optimization towards throughput on host *Goedel*. The arrival rate starts at $\lambda = 0$ and reaches $\lambda = 246$ jobs per second at 8 nodes. The mean utilization of the system is $\rho = 0.37$, whereas Table 4.5 shows the utilizations of all nodes of the initial ($\lambda = 1$) and optimal ($\lambda = 246$) configuration.

With optimization towards throughput each node is less than 80% utilized under a maximum arrival rate of $\lambda = 246$. Since the *Feeder* is again the bottleneck, the optimization only tried to increase the number of *Feeders*. This leads to both, a short response time of $R \approx 0.01273$ seconds and also to a short waiting time of $W \approx 0.00318$ seconds, and similar to an expected number of jobs in the system of $E_S \approx 1.9517$ or in queues of $E_q \approx 0.7816$. The system

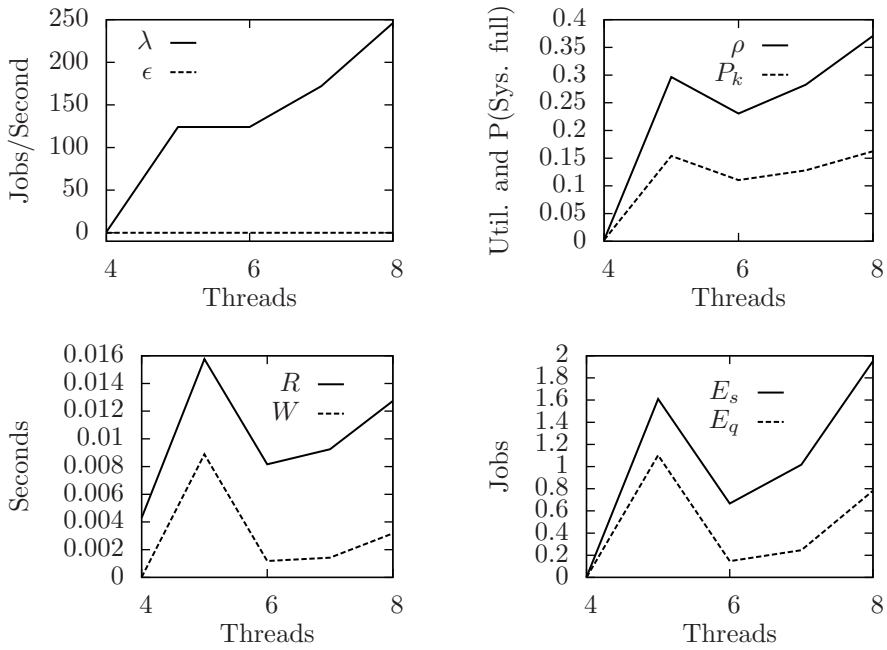


Figure 4.10: Optimization towards throughput on host *Goedel*.

is under $\lambda = 246$ still very responsive due to database connection pooling but under a relatively low throughput as shown in the next experiment.

For this experiment the number of *Feeders* is fixed to one and Figure 4.11 shows the outcome. The optimal configuration allows a maximum arrival rate

	Initial Config.		**Optimal Config.**	
Node	**Util.**	**Threads**	**Util.**	**Threads**
Decoder	0.0024	1	0.6028	1
Converter	3.0E-4	1	0.0764	1
Serializer	0	1	0.0037	1
Feeder	0.0065	1	0.7995	5

Table 4.5: Utilization and number of threads in initial and optimal configuration for optimization towards throughput on host *Goedel*.

of $\lambda = 494$ jobs per second. The loss rate ϵ is almost zero, whereas the mean utilization of the system reaches $\rho \approx 0.49$. The probability of the system being full stabilizes at $P_k \approx 0.21$. Table 4.6 confronts again initial ($\lambda = 1$) and optimal ($\lambda = 494$) configuration. Since the number of *Feeders* is fixed to one, the system increased the number of *Decoders* until 8 nodes are reached.

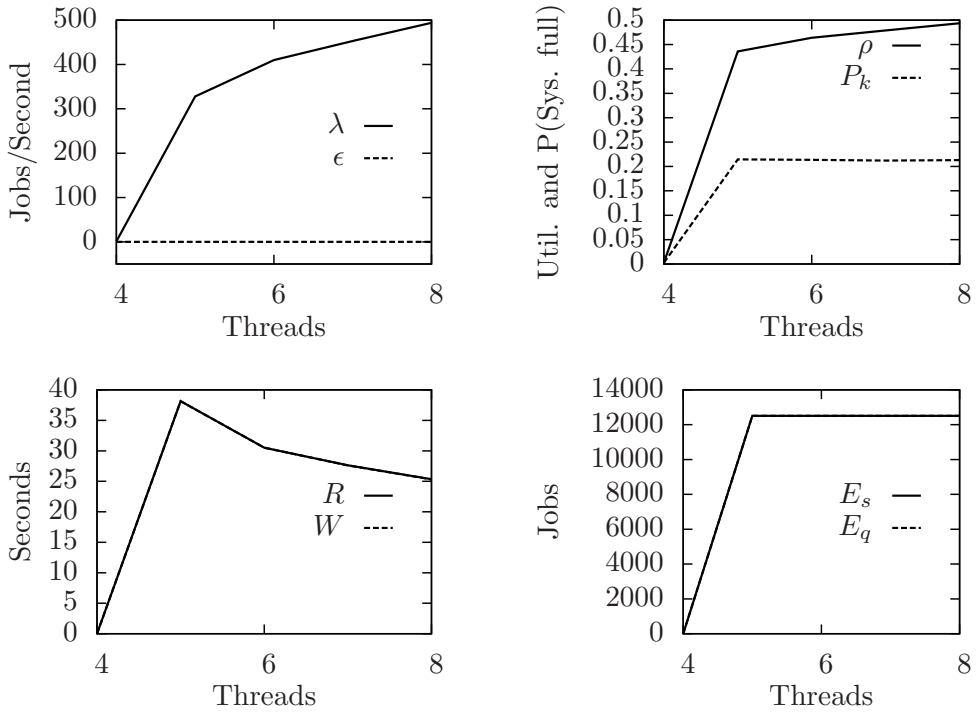


Figure 4.11: Optimization towards throughput on host *Goedel* with true bottleneck.

The nearly double arrival rate of 494 jobs per second, compared to 246 jobs per second of the previous experiment, comes from the fact that the system buffers the overload due to the high combined throughput of the 5 *Decoders* on cost of response and waiting time as well as on queue sizes. As seen in Figure

	Initial Config.		**Optimal Config.**	
Node	**Util.**	**Threads**	**Util.**	**Threads**
Decoder	0.0024	1	0.7996	5
Converter	3.0E-4	1	0.1583	1
Serializer	0	1	0.0167	1
Feeder (fixed)	0.0063	1	1	1

Table 4.6: Utilization and number of threads in initial and optimal configuration for optimization towards throughput on host *Goedel* with true bottleneck.

4.11, response and waiting times are very close together with $R \approx W \approx 25.3$ seconds which is much higher as the times of the previous experiment. Also the expected number of jobs in the system and queues ($E_s \approx E_q \approx 12510$ jobs) are far above the experienced value of the previous experiment. Sufficiently

large queues definitely enable a higher performance level without the influence of a bottleneck like database I/O.

The second host *Zerberus* is a Sun SPARC Enterprise T5220, model SED-PCFF1Z with a SPARC V9 architecture (Niagara 2) and a Sun UltraSPARC T2 eight-core processor, each core at 1.2 Ghz and with Chip Multithreading Technology (CMT) for up to 64 simultaneous threads, has 32 GB of main memory, two 146 GB Serial Attached SCSI disks, 10/100/1000 Mb/s Ethernet, and runs SunOS 5.10 Generic_127111-11.

Again the number of *Feeders* is fixed to one and Figure 4.12 shows optimization towards consolidation. The arrival rate is fixed to $\lambda = 3000$ jobs per second.

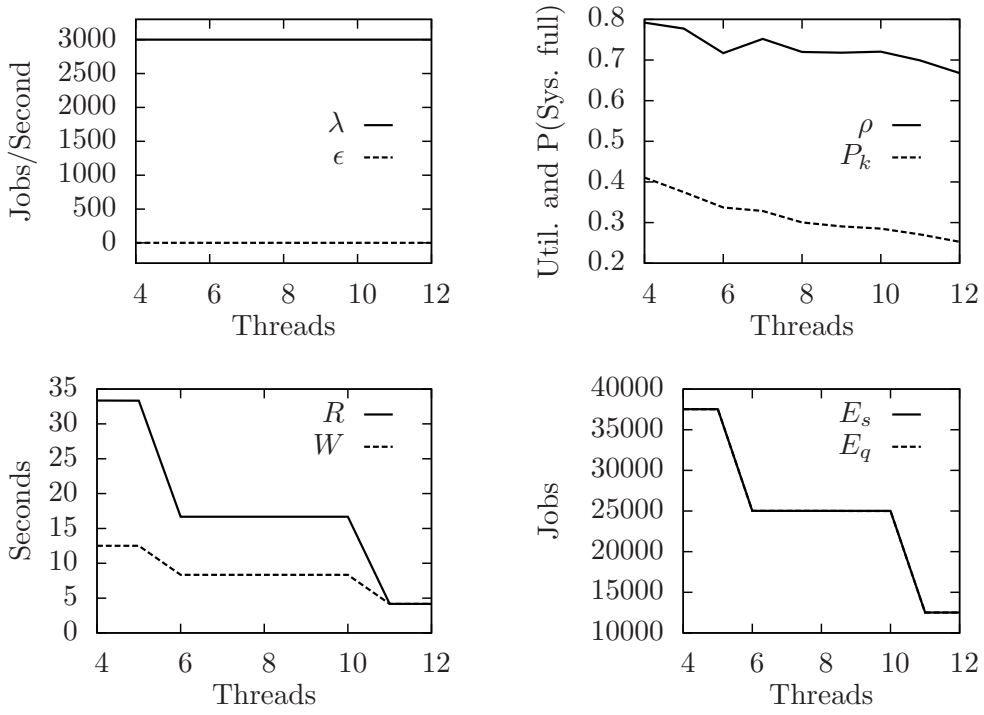


Figure 4.12: Optimization towards consolidation on host *Zerberus*.

At 4 nodes there is almost no loss rate ϵ and a mean utilization of $\rho \approx 0.79$, whereas three nodes exhibit a utilization of $\rho = 1$ as shown in Table 4.7. At 12 nodes the mean utilization is $\rho \approx 0.67$.

Table 4.7 shows the improvement from 4 to 12 cores. Optimization terminated

at 12 nodes, because *Decoders*, *Converters* and the *Serializer* exhibit a utilization less than 80%. The probability of a full system goes down to $P_k \approx 0.25$.

	Initial Config.		Optimal Config.	
Node	**Util.**	**Threads**	**Util.**	**Threads**
Decoder	1	1	0.7705	3
Converter	1	1	0.7887	7
Serializer	0.1676	1	0.112	1
Feeder (fixed)	1	1	1	1

Table 4.7: Utilization and number of threads in initial and optimal configuration for optimization towards consolidation on host *Zerberus* with one *Feeder*.

Response and waiting times merge at 11 nodes to $R \approx W \approx 4.17$ seconds, and the expected jobs in the system and queues decrease to $E_s \approx E_q \approx 12511$ jobs. Thus, the single *Feeder* functions as bottleneck, but can effectively be compensated by sufficiently large queues.

Figure 4.13 shows optimization towards throughput with the restriction of one single *Feeder*. At 12 nodes an arrival rate of $\lambda = 3061$ jobs per second

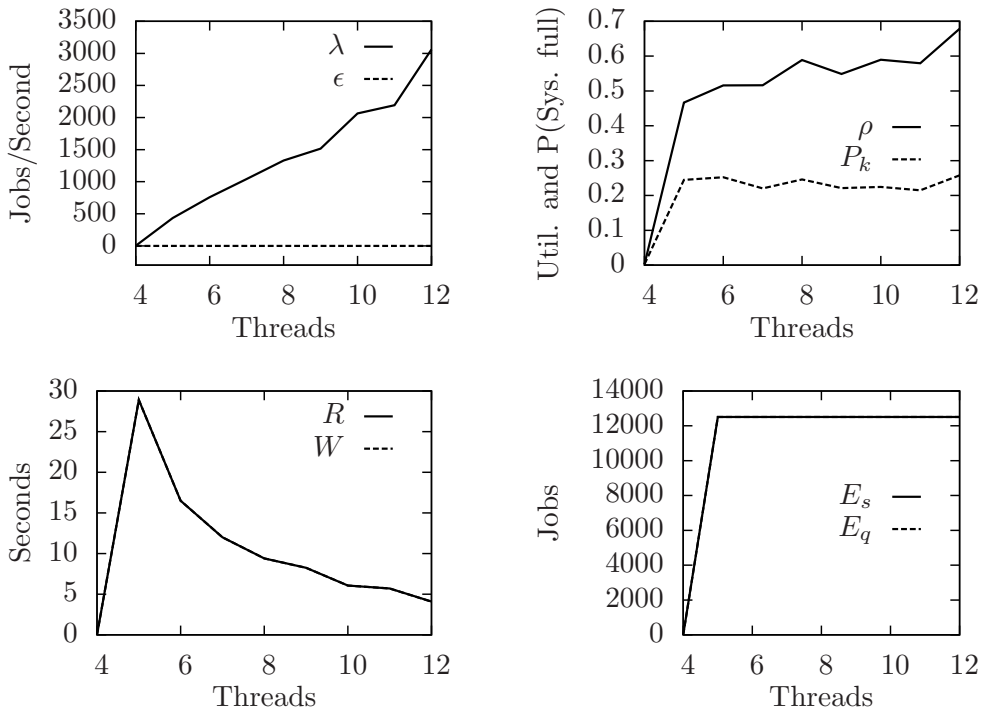


Figure 4.13: Optimization towards throughput on host *Zerberus*.

can be reached. The loss rate is $\epsilon \approx 0.0077$ and the utilization increases to $\rho \approx 0.68$, while the probability of a full system is $P_k \approx 0.26$.

The optimization terminates because of memory shortage at 12 nodes and Table 4.8 shows initial ($\lambda = 1$) and optimal configuration ($\lambda = 3061$). Response

	Initial Config.		**Optimal Config.**	
Node	**Util.**	**Threads**	**Util.**	**Threads**
Decoder	8.0E-4	1	0.7998	3
Converter	0.0019	1	0.797	7
Serializer	1.0E-4	1	0.1169	1
Feeder (fixed)	0.0073	1	1	1

Table 4.8: Utilization and number of threads in initial and optimal configuration for optimization towards throughput on host *Zerberus* with one *Feeder*.

and waiting times merge at 11 nodes to $R \approx W \approx 4.09$ seconds, and the expected jobs in the system and queues decrease to $E_s \approx E_q \approx 12511$ jobs.

Compared with the experiment of optimization towards consolidation, where $\lambda = 3000$ was assumed, we now see the true maximum arrival rate of $\lambda = 3061$.

4.5.3 Simulation

This section presents the basic idea of a simulation as additional way of validating the analytical approach. The main goal is to simulate a DFE-like system that is based on a queueing network. Therefore, a tool implemented in Java was developed to imitate the DFE software.

The basic idea is to imitate work that can be done within a node in an application like the Data Flow Engine, pass the tickets through the queueing network and measure the important performance metrics in real time.

The simulation module starts by creating and starting one *Decoder* thread, one *Converter* thread, one *Serializer* thread and one *Feeder* thread. These threads are then listening for incoming tickets in their queues, where threads of the same node share one queue.
Listing 4.4 shows the Java code to create and start the simulated node threads. The variable *servers* defines how many threads are created for each node. The

class variable *node* within the class *Node* defines the node type (e.g.: Decoder = 0).

```
1  public void createThreads() {
       threads = new TreeMap<Integer, List<Node>>();
3      for (Node node : nodes) {
         List<Node> threadList = new ArrayList<Node>();
5        for (int i = 0; i < nodes.get(node.node).servers; i++) {
           switch (node.node) {
7            case 0:
               threadList.add(new Decoder(node.node, this));
9              break;
             case 1:
11             threadList.add(new Converter(node.node, this));
               break;
13           case 2:
               threadList.add(new Serializer(node.node, this, i));
15             break;
             case 3:
17             threadList.add(new Feeder(node.node, this));
               break;
19         }
         }
21       threads.put(node.node, threadList);
         runningThreads += threadList.size();
23     }
       numThreads = runningThreads;
25   }

27   public void startThreads() {
       for (int i = 0; i < nodes.size(); i++) {
29       List<Node> threadList = threads.get(i);
         for (Node thread : threadList)
31         new Thread((Runnable) thread).start();
       }
33   }
```

Listing 4.4: Creating and starting node threads.

The next step is to put tickets into the *Decoder* queue, simulating a ticket generator, that uses a given external arrival rate. For every ticket that arrives in the *Decoder* queue, the *Decoder* thread takes the ticket out of the queue and starts its given task. After the task has been executed, the *Decoder* forwards

the ticket to the *Converter* queue.

The *Converter* then takes the ticket out of its queue and executes the given task. After that, the ticket leaves the *Converter* and is forwarded to the *Serializer* and *Feeder* queue. After the *Serializer* writes the content of the ticket to a file and the *Feeder* sends the content of the ticket to a database, the ticket leaves the queueing network.

For each task within each node and within each thread the duration is measured. After a given number of tickets have been sent to the queueing network by the simulated ticket generator, the mean service rate for each node type is calculated.

As mentioned before, all servers/threads of a given node share one queue. Listing 4.5 shows that if there is a ticket in a queue, a node thread will remove the ticket from the queue and execute its tasks.

```
queues = new TreeMap<Integer, List<Ticket>>();
[...]
Ticket ticket = null;
if (sim.queues.get(node).size() > 0) ticket = sim.queues.get(node).
    remove(0);
if (ticket != null) {
  //Execute Tasks
  [...]
}
```

Listing 4.5: Creating and starting node threads.

To simulate a ticket generator, two approaches have been developed:

- The basic version of the ticket generator puts a given number of tickets into the *Decoder* queue using a specified arrival rate (see Listing 4.6).
- The enhanced version of the ticket generator puts always an appropriate amount of tickets into the *Decoder* queue using a specified arrival rate, based on service times of all threads used to calculate their standard deviation

$$s = \sqrt{\frac{1}{N-1}\sum_{i=1}^{N}(x_i - \bar{x})^2} \quad , \tag{4.19}$$

where $\bar{x}$ is the mean service time for a single thread and N is the amount of tickets already sent. By calculating

$$E_{\bar{x}} = \frac{z_\beta \frac{s}{\sqrt{N}}}{\bar{x}} \tag{4.20}$$

Z_β is used as β-Quantile of the standard-normal distribution $N_{0,1}$.

$E_{\bar{x}}$ therefore defines the standard error for the measured service times (see Listing 4.7). The ticket generator stops sending tickets if all threads show, with a probability of β, an error smaller than 5%. (see Listing 4.8).

```
public void createTicketsStatic(int tickets) {
  for (int i = 0; i < tickets; i++) {
    sentTickets++;
    Thread.sleep((long) ((1 / nodes.get(0).extAR) * 1000));
    if (queues.get(0).size() < queueLengthsMax.get(0)) {
      queues.get(0).add(new Ticket(i));
    }
    else synchronized (losses) {
      losses.put(0, losses.get(0) + 1);
    }
  }
}
```

Listing 4.6: Creating Tickets: static.

During the simulation, the module records the time needed for each task within each thread, as well as the number of lost tickets for each thread and the queue sizes.

After all tickets are sent to the system and all tickets have left the system, the mean loss rate as well as the mean number of jobs in queue are calculated. The time needed for each task within each thread in relation to the entire simulation time is used to calculate the utilization for each thread and node.

```
public void createTickets() {
  boolean ErrorExceeded = true;
  float maxErr = 0;
  while (ErrorExceeded) {
    ErrorExceeded = false;
    sentTickets++;
    Thread.sleep((long) ((1 / nodes.get(0).extAR) * 1000));
    if (queues.get(0).size() < queueLengthsMax.get(0)) {
      queues.get(0).add(new Ticket(i));
    }
    else synchronized (losses) {
      losses.put(0, losses.get(0) + 1);
    }
  for (int i = 0; i < nodes.size(); i++) {
    List<Node> threadList = threads.get(i);
    for (Node thread : threadList) {
      float err = Tools.calculateError(thread.values);
      if (err > Start.limitStatErr) {
        ErrorExceeded = true;
        break;
      }
      else if (err > maxErr) maxErr = err;
    }
    if (ErrorExceeded) break;
  }
  if (!ErrorExceeded) break;
}
```

Listing 4.7: Creating Tickets: optimum.

In the next step, the mean utilization for each node type is calculated. The module then extends the configuration as mentioned in Subsection 4.4.3. Within the simulation module, the node utilization is therefore not calculated, but instead measured in real time. For example, if the mean utilization of the *Converter* node is higher than a specified utilization threshold (e.g. $limU > 0.8$), the new configuration adds one additional *Converter* thread.

The simulation module then sets up this new configuration by creating and starting the new optimum number of threads. Like the measurement module (described in Section 4.5.2), the simulation module recursively adds threads of over-utilized nodes until none of the nodes is over-utilized according to $limU$, or resource restrictions are reached. Therefore, in every new cycle, a new

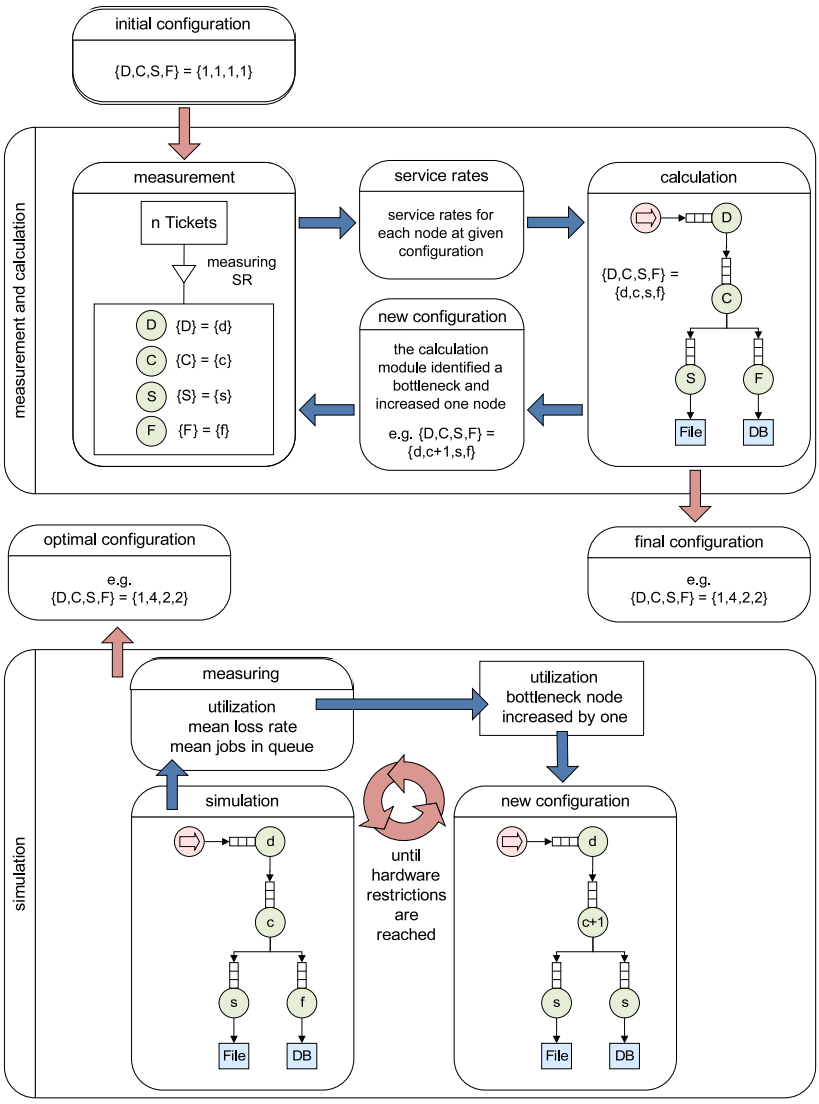
simulation is started.

```
1  public static float calculateError(List<Double> values) {
     float currErr = 1;
3    if (values.size() > 10) {
       double sum = 0;
5      synchronized (values) {
         for (double v : values)
7          sum += v;
         float mean = (float) sum / values.size();
9        float sumDiffs = 0;
         for (double v : values)
11         sumDiffs += (v - mean) * (v - mean);
         float varSample = sumDiffs / (values.size() - 1);
13       float stdDev = (float) Math.sqrt(varSample);
         float delta = 1.64485f * (stdDev / (float) Math.sqrt(values.size
             ())); /* 0.95 */
15       currErr = delta / mean;
       }
17   }
     return currErr;
19 }
```

Listing 4.8: Calculating the standard error.

E.g.: After the initial configuration was simulated and the service rates and utilizations were recorded, the simulation module suggests to add a second *Converter* node. The simulation module then creates and starts one *Decoder* thread, two *Converter* threads, one *Serializer* thread and one *Feeder* thread. The *Converter* threads both share the *Converter* queue. Both *Converter* threads can therefore take tickets out of the *Converter* queue and execute their tasks in parallel. Again, after a given amount of tickets have been sent to the system, service rates for each node type are measured. These new service rates may differ from previous recorded service rates due to the parallelizing of the nodes. Figure 4.5.3 shows the recursive use of the simulation and the calculation module. As soon as a configuration exceeds given resources, the measurement and the simulation module have found the optimal configuration. The procedure of the calculation/measurement approach can be summarized as follows:

- detecting the hardware specifications on the given host
- simulating node tasks and measuring the service rates



- calculating the optimum configuration based on the analytical model

On the other hand, the simulation approach can be summarized as follows:

- creating a thread-based simulation of the queueing network
- creating tickets to put into the queueing network
- using the artificial nodes to simulate the tasks within the node threads
- measuring performance metrics
- increase the most over-utilized (bottleneck) node by one and restart the simulation

Section 4.5.5 describes the combination of all approaches in more detail.

Table 4.9 shows the optimum configuration and the node utilizations in initial and optimal configuration on host *Goedel* with a fixed arrival of $\lambda = 1200$ jobs per second, presented by the simulation module. The optimum queueing network software configuration on host *Goedel* therefore should use two *Decoders*, one *Converter*, one *Serializer* and four *Feeders*. Table 4.10 shows initial and

	Initial Config.		**Optimal Config.**	
Node	**Util.**	**Threads**	**Util.**	**Threads**
Decoder	0.9936	1	0.9771	2
Converter	0.1778	1	0.2854	1
Serializer	0.0337	1	0.0196	1
Feeder	0.9864	1	0.9962	4

Table 4.9: Utilization and number of threads in initial and optimal configuration for optimization by simulation on host *Goedel*.

optimal node utilizations and the optimum configuration with a true bottleneck. In this simulation the ticket generator automatically chose the right amount of tickets and sent (on average) 2856 tickets per simulation cycle. In this experiment over-utilized nodes are nodes with a utilization higher than just 4%. Due to the thread-optimization, the utilization of the entire system is reduced from 23.09% in the initial configuration to 15.24% in the optimum configuration. Table 4.11 shows the optimum configuration as well as the uti-

	Initial Config.		**Optimal Config.**	
Node	**Util.**	**Threads**	**Util.**	**Threads**
Decoder	0.1047	1	0.0315	3
Converter	0.1960	1	0.0357	6
Serializer	0.0386	1	0.0107	1
Feeder (fixed)	0.5841	1	0.5314	1

Table 4.10: Utilization and number of threads in initial and optimal configuration for optimization by simulation with true bottleneck on host *Zerberus*.

lization of nodes in initial and optimal configuration on host *Zerberus* with no true bottleneck and a utilization bound of 12%. The utilization of the entire system was reduced from 23.49% in the initial configuration to 6.09% in the optimal configuration. This time 15904 tickets were sent on average in each

simulation cycle. To fully utilize host *Zerberus*, a real-time-like simulation, or

	Initial Config.		Optimal Config.	
Node	**Util.**	**Threads**	**Util.**	**Threads**
Decoder	0.0901	1	0.0521	2
Converter	0.1933	1	0.0739	3
Serializer	0.0388	1	0.0069	1
Feeder	0.6174	1	0.1107	7

Table 4.11: Utilization and number of threads in initial and optimal configuration for optimization by simulation on host *Zerberus*.

a real system, like the queueing network software is needed (see Section 4.5.4).

4.5.4 Experiments

In the previous sections, calculation, measuring the service rates of artificial nodes and the simulation of an entire queueing-network software have been used. This section describes experiments using the actual queueing network software software. Also, a ticket generator will be described, sending diameter tickets to the queueing network software.

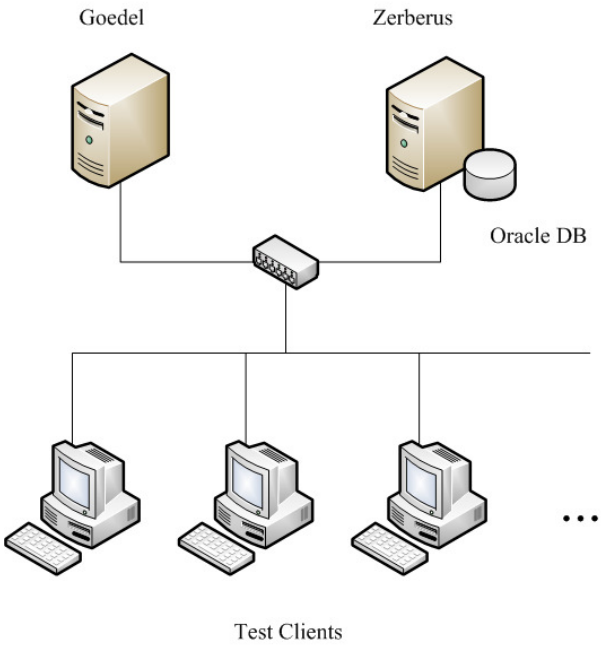


Figure 4.14: Test Environment: Testing the queueing network software.

Figure 4.14 shows the test environment used in the following experiments. The ticket generator was installed on a number of test clients. These test clients were then started and were sending Diameter tickets to the queueing network software, installed either on host *Goedel* or host *Zerberus*. The Oracle database was only installed on host *Zerberus*, so database entries from the queueing network software installed on host *Zerberus* were just local invocations, while each ticket passing through the *Feeder* node on host *Goedel* included some network involvement as well.

Ticket Generator

To test the queueing network software, a ticket generator was developed in Java. The goal of this ticket generator was to imitate mobile devices and placed VoIP calls. Therefore, for every call, a new thread was created and started, which is alive for the duration of the call. Therefore, each simulated call is represented by an individual thread, which is sending the Diameter tickets, like a real VoIP device.

Before starting the ticket generator, the tester can specify a number of parameters:

- The *mean external arrival rate* and the distribution of the arrival rate, which states the time that passes between two calls are started.
- The *mean call length* and the distribution of the call length, which states how long a call should last.
- The *mean loss rate* and the distribution of the loss rate, which states the possibility that a ticket is lost.

The following distributions can be used for the previous described rates: *Poisson* (see Listing 4.9), *Weibull*, *Pareto* (see Listing 4.10), *Geometric*, *Exponential* and *Uniform*.

```
1 public double getPoissonVariate(double mean) {
    double lambda = 1.0 / mean;
3   double v = -Math.log(next()) / lambda;
    return v;
5 }
```

Listing 4.9: Method for calculating a Poisson variate.

```
1 public double getParetoVariate(double mode, double shape) {
      double v = mode / Math.pow((1 - next()), 1 / shape);
3     return v;
    }
```

Listing 4.10: Method for calculating a Pareto variate.

Therefore, the tester can specify the mean arrival rate, the mean call length and the mean loss rate. Also, the tester can specify the mentioned distributions for each parameter, which uses the mean values as an input to generate (e.g.: Poisson arrival, see Figure 4.15).

After the ticket generator is started, it sets up an RMI connection to the queueing network software. Once the connection is established, the ticket generator starts to create threads imitating calls. Therefore the Java class *TestTool* is actually a call generator. The actual tickets are sent within the Java thread class *TicketGenerator*. Figure 4.15 shows the structure of the Java test tool.

Listing 4.11 shows the basic routine of the ticket generator. The ticket generator creates new calls as long as the simulation time has not expired. With the help of a mean call time and the previously mentioned (e.g.) Exponential arrival, the length of the next call $[t_C]$ (see Figure 4.15) is generated (variable *mct*). Given the fact that calls have to take at least 1000ms, the ticket generator prevents that from happening. After that, a new call thread is created with an *ID* and the calculated *call time*. Once a call has been started, the waiting time between two calls has to be calculated, again using the chosen distribution. A mean external arrival rate is used to generate a Exponential distributed waiting time $[t_W]$ (see Figure 4.15) until a new call can be created.

```
  double timestart = System.currentTimeMillis();
2 double timeend = System.currentTimeMillis();
  for (long i = 1; timeend<(timestart+timesimulation); i++) {
4   long mct = (long)v.getExponentialVariate(meancalltime);
    if (mct<1000) mct=1000;
6   TicketGenerator tGen = new TicketGenerator(i,mct);
    new Thread((Runnable) tGen).start();
8   Thread.sleep((long)v.getExponentialVariate(arrivalrate));
    timeend = System.currentTimeMillis();
10 }
```

Listing 4.11: Ticket Generator: Starting Calls.

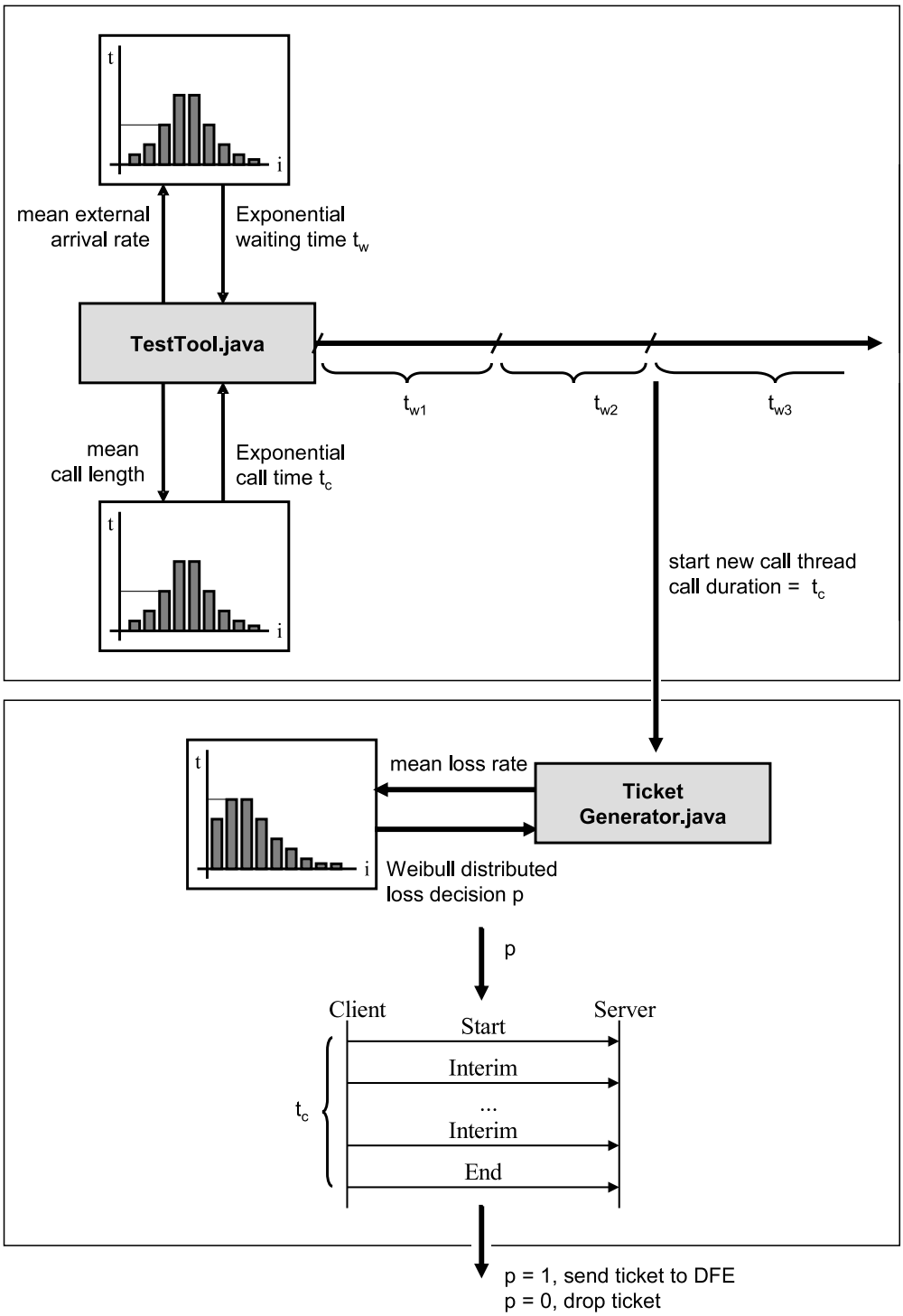


Figure 4.15: Test Scenarios: Java classes and distributions.

Within the class *TicketGenerator*, the tool first calculates how many *Interim* tickets have to be sent, using the entire *call time* and the time between two *Interim* tickets. Listing 4.12 furthermore shows that after an initial *Start* ticket has been sent, the thread knows how many *Interim* tickets have to be sent. The time between two *Interim* tickets is given by the variable *ticketinterim* [seconds].
The TicketGenerator also uses a Weibull distribution to decide whether to send each individual message, or to drop it and continue with the next ticket. The Weibull distribution therefore provides a factor p for every ticket (*Start*, every single *Interim* and *Stop*) stating if the ticket will be sent or dropped (see Figure 4.15).
After all *Interim* tickets have been sent, the ticket generator calculates the time that is remaining until the end of the call. Then a *Stop* ticket is sent and the call is successfully terminated.

```
ticketinterim = AvpCodes.ACCT_INTERIM_INTERVAL * 1000;
long interims = duration/ticketinterim;
//sending start ticket
IMessage startMessage = DFEstarter.tpd.createStartMessage(session, id);
TestProcessDiameter.streamingFeader.process("diameter", parser.
    encodeMessage(startMessage).array());
//sending interim tickets
for (int i = 0; i < (int)interims; i++) {
  Thread.sleep(ticketinterim);
  IMessage interimMessage = DFEstarter.tpd.createInterimMessage(session
      , i + 1, id);
  TestProcessDiameter.streamingFeader.process("diameter", parser.
      encodeMessage(interimMessage).array());
}
long remaining = duration - (ticketinterim*(int)interims);
Thread.sleep(remaining);
//sending stop ticket
IMessage stopMessage = DFEstarter.tpd.createStopMessage(session, i+1,
    id);
TestProcessDiameter.streamingFeader.process("diameter", parser.
    encodeMessage(stopMessage).array());
```

Listing 4.12: Ticket Generator: Call Thread and Sending Tickets.

As mentioned before, the distributions used in Figure 4.15 are just examples. The tester can individually decide which distribution to use for the arrival rate,

call time and loss rate.

Results

The first experiments were conducted with the previously described ticket generator and the actual queueing network software software installed on host *Zerberus*. As mentioned before, host *Zerberus* has an Oracle database installed locally and has the possibility to start up to 64 parallel threads.

As mentioned before, the ticket generator is installed on a number of clients and sends tickets to the queueing network software. An additional client uses a Java thread to monitor the queueing network software remotely. The queueing network software offers a number of performance metrics that can be accessed via RMI (e.g.: the number of invocations and the overall active time for every individual node thread). The monitoring thread then uses the external arrival rate and the service rate (number of invocations per busy time) to calculate a utilization for every individual node thread and suggests to extend the most over-utilized node.

Again, an over-utilization occurs if the node is utilized more than 80%. Therefore, nodes that show a utilization of over 80% will be increased by one thread.

Table 4.12 shows the service rate and the utilization of all four nodes with an external arrival rate of 1000 tickets per second. This first experiment makes it obvious that the *Feeder* node is the bottleneck of the system, only managing an average number of 66 tickets per second. Therefore the adaption tool suggests that the *Feeder* node has to be increased by one, creating two *Feeder* threads at the next experiment.

	Service Rate [tickets/s]	**Utilization [%]**
Decoder	10658	9.38[%]
Converter	12147	8.23[%]
Serializer	2061	48.52[%]
Feeder	66	100.00[%]

Table 4.12: Initial configuration (1-1-1-1) on host *Zerberus*, tested with an external arrival rate of 1000[tickets/s].

In the next few steps it became obvious that even though the *Feeder* node was recursively extended, the service rate did not increase in the same way. Table 4.13 shows that the mean service rate of one *Feeder* thread decreases with every newly added *Feeder* thread.

Of course, the total service rate of all *Feeder* nodes does not decrease, but adding new *Feeder* threads does not improve the total service rate of all *Feeder* nodes enough. Even with the maximum number of 61 threads, the *Feeder* node stays the systems bottleneck.

	Mean Service Rate	
Configuration	**Individual**	**Total**
1-1-1-1	66	66
1-1-1-2	44	88
1-1-1-3	41	123
1-1-1-4	34	136
1-1-1-10	7	70
1-1-1-20	5	100
1-1-1-61	2	122

Table 4.13: Service rates of the *Feeder* node with different configurations on host *Zerberus*.

Table 4.14 shows that with the final configuration (one *Decoder*, one *Converter*, one *Serializer* and 61 *Feeder* nodes) all other nodes are of course still under-utilized. This leads to the conclusion that the *Feeder* node should indeed be fixed to one thread per node. The solution to this problem, as mentioned before, could be to allocate a large queue to the *Feeder* node. By doing that, the node can eventually handle queued tickets when the external arrival rate decreases.

	Utilization [%]
Decoder	9.25[%]
Converter	9.67[%]
Serializer	48.95[%]
Feeder	100.00[%]

Table 4.14: Final configuration (1-1-1-61) on host *Zerberus*, tested with an external arrival rate of 1000[tickets/s].

The final experiment therefore used a *Feeder* node fixed to one thread per node. Table 4.15 shows that an initial configuration of one thread per node cannot handle an external arrival rate of 2000 tickets per second without overstepping an utilization of 80%, because the *Serializer* node is already at a utilization level of 97.04%. Increasing the number of threads per node (without taken the *Feeder* node into account) the final and optimal configuration can handle an external arrival rate of 66900 tickets per second.

	Initial Config. 2000tickets/s		**Optimal Config. 66900tickets/s**	
Node	**Util.**	**Threads**	**Util.**	**Threads**
Decoder	0.1877	1	0.7911	9
Converter	0.1646	1	0.7044	9
Serializer	0.9704	1	0.7993	45
Feeder (fixed)	1.000	1	1.000	1

Table 4.15: Initial and final configuration on host *Zerberus*, with a fixed *Feeder* node.

Table 4.15 shows that at an external arrival rate of 66900 tickets per second, all nodes are under a utilization level of 80%. Of course it should be noted, that the *Feeder* node is still highly over-utilized and can only handle about 70 tickets per second. Given the fact that the *Feeder* node and therefore the database is the natural bottleneck, it is necessary to assign a very large queue to the *Feeder* node, to minimize the loss rate. Over time and during parts of the day where not many VoIP calls are started, the *Feeder* node will then be able to eventually handle all tickets in its queue.

To compare the results and maybe find a different optimal configuration the same experiments were conducted with the queueing network software installed on host *Goedel*. With four dual-core processors, a maximum amount of 8 threads can be started. As mentioned before, host *Goedel* has no local database installed and uses the Oracle database installed on host *Zerberus*.
Table 4.16 shows the results of the first experiment. At an external arrival rate of 1000 tickets per second, the *Feeder* node is again the systems bottleneck. Table 4.16 also shows that compared to host *Zerberus*, the *Decoder*, *Converter* and *Serializer* node show a higher service rate during the initial experiment.
To start the optimization process, the *Feeder* node again has to be increased.

	Service Rate [tickets/s]	**Utilization [%]**
Decoder	32617	3.07[%]
Converter	26058	3.84[%]
Serializer	5202	19.22[%]
Feeder	105	100.00[%]

Table 4.16: Initial configuration (1-1-1-1) on host *Goedel*, tested with an external arrival rate of 1000[tickets/s].

Table 4.17 shows that on host *Goedel* the individual service rate of one *Feeder* thread does, to some extent, stay the same, which leads to the fact that the total service rate of all *Feeder* threads is indeed slowly increasing. Table 4.17 shows that one *Feeder* thread can handle 105 tickets per second, while the final configuration of 5 *Feeder* threads can handle 350 tickets per second.

	Mean Service Rate	
Configuration	**Individual**	**Total**
1-1-1-1	105	105
1-1-1-2	79	158
1-1-1-3	81	243
1-1-1-4	70	280
1-1-1-5	70	350

Table 4.17: Service rates of the *Feeder* node with different configurations on host *Goedel*.

Given the fact that the total service rate of all *Feeder* threads is increasing, it would make sense to stick to this optimization approach. Table 4.18 therefore shows an initial and optimal configuration on host *Goedel*. With an external arrival rate of 280 tickets per second the *Feeder* node of the initial configuration is over-utilized, but with the optimal configuration of 5 threads, the *Feeder* node is able to stay under the utilization threshold of 80%.

Table 4.19 shows the results of the experiments, if the software developer decides to fix the *Feeder* node to one thread per node. At an initial configuration of one thread per node and an external arrival rate of 5200 tickets per second, the *Serializer* node would exceed the utilization threshold of 80%. After the optimization process, the system is able to handle up to 15600 tickets per second with the optimal queueing network software configuration (one *Decoder* node, two *Converter* nodes, four *Serializer* nodes and one *Feeder* node), with-

	Initial Config. 280tickets/s		Optimal Config. 280tickets/s	
Node	**Util.**	**Threads**	**Util.**	**Threads**
Decoder	0.0086	1	0.0097	1
Converter	0.0107	1	0.0141	1
Serializer	0.0538	1	0.0404	1
Feeder	1.000	1	0.8000	5

Table 4.18: Initial and final configuration on host *Goedel*.

out exceeding the utilization threshold.

	Initial Config. 5200tickets/s		Optimal Config. 15600tickets/s	
Node	**Util.**	**Threads**	**Util.**	**Threads**
Decoder	0.1594	1	0.5210	1
Converter	0.1996	1	0.7502	2
Serializer	0.9996	1	0.7989	4
Feeder (fixed)	1.000	1	1.000	1

Table 4.19: Initial and final configuration on host *Goedel*, with a fixed *Feeder* node.

Verification of the Analytical Approach

To verify the analytical approach I used the average service rates for each node derived from the experiments done on host *Zerberus* and host *Goedel* (see Table 4.20) and started the calculation module one more time.

	Mean Service Rates	
Node	**Zerberus**	**Goedel**
Decoder	9766	30055
Converter	10430	18672
Serializer	1987	6146
Feeder	28	94

Table 4.20: Mean service rates of both tested hosts for each node type.

Table 4.21 shows that starting the calculation module with optimization towards throughput, and using the average service rates derived out of the exper-

iments, both, experiments and the calculation module deliver the same optimal configuration. On host *Zerberus* and host *Goedel* a normal optimization would only increase the number of *Feeder* nodes. Due to different hosts and therefore different node service rates, an optimal configuration with a fixed *Feeder* node would lead to an optimal configuration of 9 *Decoder* nodes, 9 *Converter* nodes, 45 *Serializer* nodes and 1 *Feeder* node on host *Zerberus*, and 1 *Decoder* node, 2 *Converter* nodes, 4 *Serializer* nodes and 1 *Feeder* node on host *Goedel*.

	Zerberus		**Goedel**	
Optimal Configuration	**Normal**	**Fixed Feeder**	**Normal**	**Fixed Feeder**
Experiments	1-1-1-61	9-9-45-1	1-1-1-5	1-2-4-1
Calculation	1-1-1-61	9-9-45-1	1-1-1-5	1-2-4-1

Table 4.21: Optimal configuration of threads for both hosts.

These four verifications show the importance of the service rates and if there is no possibility to derive real service rates from actual software, it is necessary to analyze the used nodes in every detail. As mentioned before, the simulation and the measurement module are using simulated nodes to derive artificial service rates. Therefore, the simulated tasks have to be as close as they can get to the actual performed tasks of the tested queueing network software.

4.5.5 Autonomic Adaption Tool

The idea behind optimizing the DFE on a specific system was, that the software will be installed on a great number of different hardware systems. Companies which install the DFE may want to decide which servers and hardware components to use, but also over the course of time, hardware components may be replaced and servers will be upgraded. Therefore, the idea was to find a quick way to determine the optimum DFE-configuration for a certain system, before the DFE was even installed on that system. The DFE would then use this determined configuration until hardware specification change and a new optimization process is started manually.

The idea behind the recursive approach, where a new thread will be added in each step or cycle, was to monitor if a newly added thread has consequences on the service rates (the service rates of the other nodes and the service rates

of the other threads from the same node type). By measuring the service rates, running simulations and especially during the extensive experiments it became obvious that the service rate for individual threads and nodes does not stay the same, but varies to a certain degree. Therefore, the step-wise approach was ideal, because service rates will be measured more often, but it also showed, that the approach of running the optimization once and leaving the DFE like that may result in the fact that the DFE would not always use the optimal configuration, due to slight service rate changes.

The idea of this section is to present a solution to the mentioned problem. Unfortunately, the current version of the queueing network software does not include the possibility to automatically change the configuration of the system during runtime. At the current software version, the tester has to manually stop the queueing network software, change the configuration (threads per node) and restart the queueing network software again.

The driving factors of DFE-like software are the external arrival rate and the service rates of the nodes. Therefore to create a system that automatically adapts to a current workload and finds the best configuration for a specific host, the following strategy may be used:

- First, the tasks of each node of the used software have to be understood. Then, the appropriate tasks have to be picked and assigned to the nodes within the simulation module.
- Second, after installing the software on a new host, the simulation module should be started.
- The simulation module does include the measurement module, measuring the service rates of the simulated tasks for each node.
- By doing that, a starting configuration for the system can be found in a relatively short period of time.
- After a initial configuration is found, the system is ready to go online.
- One important factor of the systems performance is the external arrival rate. The system must be able to handle the incoming load. Therefore

the automated tool should be able to detect significant changes of the arrival rate.

- Once a significant change of the external arrival rate has been detected, a new configuration should be calculated.
- To calculate a new configuration, the current service rate for every node is needed.
- If the used software offers some self-monitoring methods, the current service rates should be calculated from real data.
- If no performance metrics monitoring is offered, the simulation module has to once again simulate the tasks of the nodes. The measurement module should measure the service rates for each node. And the calculation module should calculate a new configuration.
- Note: Furthermore the system could be extended to storing information about what configuration to use at a specific arrival rate, to minimize the effort of continuously simulating and calculating new configurations.

As mentioned before, the DFE was in principle developed to be set up once with an initial configuration and always run with the given configuration. This scenario just offers the possibility for an offline optimization. In the previous sections simulations, measurements and calculations are used to find optimal configurations for a given host. The idea behind these individual approaches was to increase the external arrival rate as much as possible and to find the optimal configuration of nodes.
This approach would in every case use all the available hardware resources to find an optimum configuration, leading to the fact that as many threads as cores available on the system would be started.

The idea behind an online optimization is that the system not only adapts to factors on the inside of the system (e.g.: available cores or other hardware restrictions), but also on outside factors, like the external arrival rate. The advantage of such an online optimization tool is that threads or cores could be turned off when the system is under-utilized.

A possible advantage of an online optimization could be energy efficiency. The hypothesis being that a few fully utilized cores will consume less energy that a great amount of under-utilized cores.

4.5.6 Excursus: Speeding up database operations

Previous sections showed that the *Feeder* node is by far the bottleneck of the described queueing network. The *Feeder* node takes tickets out of its queue and writes the converted data into an Oracle database. Hitherto the proposed approach was to assign a large queue to the *Feeder* node and hope, that over time, the *Feeder* will be able to process all incoming tickets.

The theoretical approach proposed in this section is twofold:

- Software optimizing techniques (e.g.: installing new database drivers), along with increasing the network bandwidth between the queueing network software and the database servers, are used to speed up the *Feeder* node. This will lead to an improvement of the individual *Feeder* node service rate.
- Setting up not only one database server, but using one database server for every individual *Feeder* thread guarantees that every individual thread delivers the same service rate. This will lead to an improvement of the overall *Feeder* node service rate.

In the following experiments, the analytical model presented in Sections 4.4.3 and 4.5.1 is used with the service rates derived out of the experiments presented in Section 4.5.4 to find an optimal configuration. The two mentioned techniques are used to hypothetically speed up the *Feeder* node five, ten and fifteen times (see Table 4.22).

These service rates are now used by the calculation module to find an optimum solution for this hypothetical scenario. Tables 4.13 and 4.14 showed that even with an external arrival rate of only 1000 tickets per second and 61 *Feeder* threads, with a total service rate of just 122 tickets per second, the *Feeder* node shows at the optimal configuration utilization of 100%.

Table 4.22 shows that with the help of the two mentioned techniques, the

	***Feeder* speed-up factor**			
Node	**Normal**	**x5**	**x10**	**x15**
Decoder	10658	10658	10658	10658
Converter	12147	12147	12147	12147
Serializer	2061	2061	2061	2061
Feeder	66	330	660	990

Table 4.22: Service rates for all four nodes. Service rate of the *Feeder* improved by factor five, ten and fifteen.

service rate of an individual *Feeder* thread can be improved to 330, 660 and 990 tickets per second. The multiplied database servers are then guaranteeing that with every new *Feeder* thread, the total service rate multiplies.

Table 4.23 now shows the optimal configurations of the system with an hypothetical speed-up of the *Feeder* node of 5, 10 and 15 times. It can be seen that with a speed-up of only 5 times, and the fact that with every new *Feeder* thread, the service rate enhances, the optimal configuration of 2 *Decoder* threads, 2 *Converter* threads, 9 *Serializer* threads and 51 *Feeder* threads, the system is now able to handle an external arrival rate of 13400 tickets per second with every node under a utilization barrier of 80%.

With a *Feeder* speed-up of 10 times, the optimal configuration of the system is built with 3 *Decoder* threads, 3 *Converter* threads, 14 *Serializer* threads and 44 *Feeder* threads and is now able to handle an external arrival rate of 23000 tickets per second. The external arrival rate can even be increased to 29500 tickets per second, with a *Feeder* speed-up of 15 times.

	***Feeder* 5x**		***Feeder* 10x**		***Feeder* 15x**	
ext.arr.rate:	**13400**		**23000**		**29500**	
Node	**Threads**	**Util.[%]**	**Threads**	**Util.[%]**	**Threads**	**Util.[%]**
Decoder	2	62.86	3	71.93	4	69.20
Converter	2	55.16	3	63.12	4	60.71
Serializer	9	72.24	14	79.71	18	79.52
Feeder	51	79.62	44	79.20	38	78.42

Table 4.23: Service rates for all four nodes. Service rate of the *Feeder* improved by factor five, ten and fifteen.

In the original setup (1-1-1-61) with a total *Feeder* service rate of 122 tickets per second, the system can only handle 97 tickets per second, so that the *Feeder* node stays under a utilization barrier of 80%. By increasing the network bandwidth, installing new drivers, or installing replicated databases the system can now handle 29500 tickets per second. This represents an overall approvement of more than 300 times.

4.5.7 Excursus: Using an enhanced queueing network for further analytical analysis

The following section proposes an approach to artificially enhance the queueing network, by not only increasing the number of threads per node, but actually using more node types to get a bigger queueing network flow graph.

The basic idea is to use the original queueing network and add the exact same queueing network to the two queueing network sink nodes *Serializer* and *Feeder*. The *Serializer* and *Feeder* nodes are still writing the data they receive to a file and the Oracle database, but this time the tickets are then forwarded to another *Decoder* node.

When the tickets arrive at the new *Decoder* node, it wanders through the same queueing network again. This means that now the queueing network consists of 12 node types instead of just 4. Every node exists exactly three times, but each one has no bond to the other nodes of the same type. Also, each of the three nodes has its own queue.

The content of a ticket wandering through the new queueing network will therefore be written three times to file and into the database. The *Decoder* and *Feeder* node are theoretically decoding and converting the ticket again. Therefore, in this scenario, the service rates derived from the experiments and presented in Table 4.22 are also used for the newly multiplied nodes. The optimization techniques presented in the previous section will further be used for the optimization process as well. For the analytical optimization a *Feeder* speed-up of 10 times is assumed. Therefore, the service rate for all three *Feeder* nodes is 660 tickets per second.

Figure 4.16 shows the structure of the new queueing network, which will be used for analytical optimization. It can be seen that the system now has four sink nodes, instead of two.

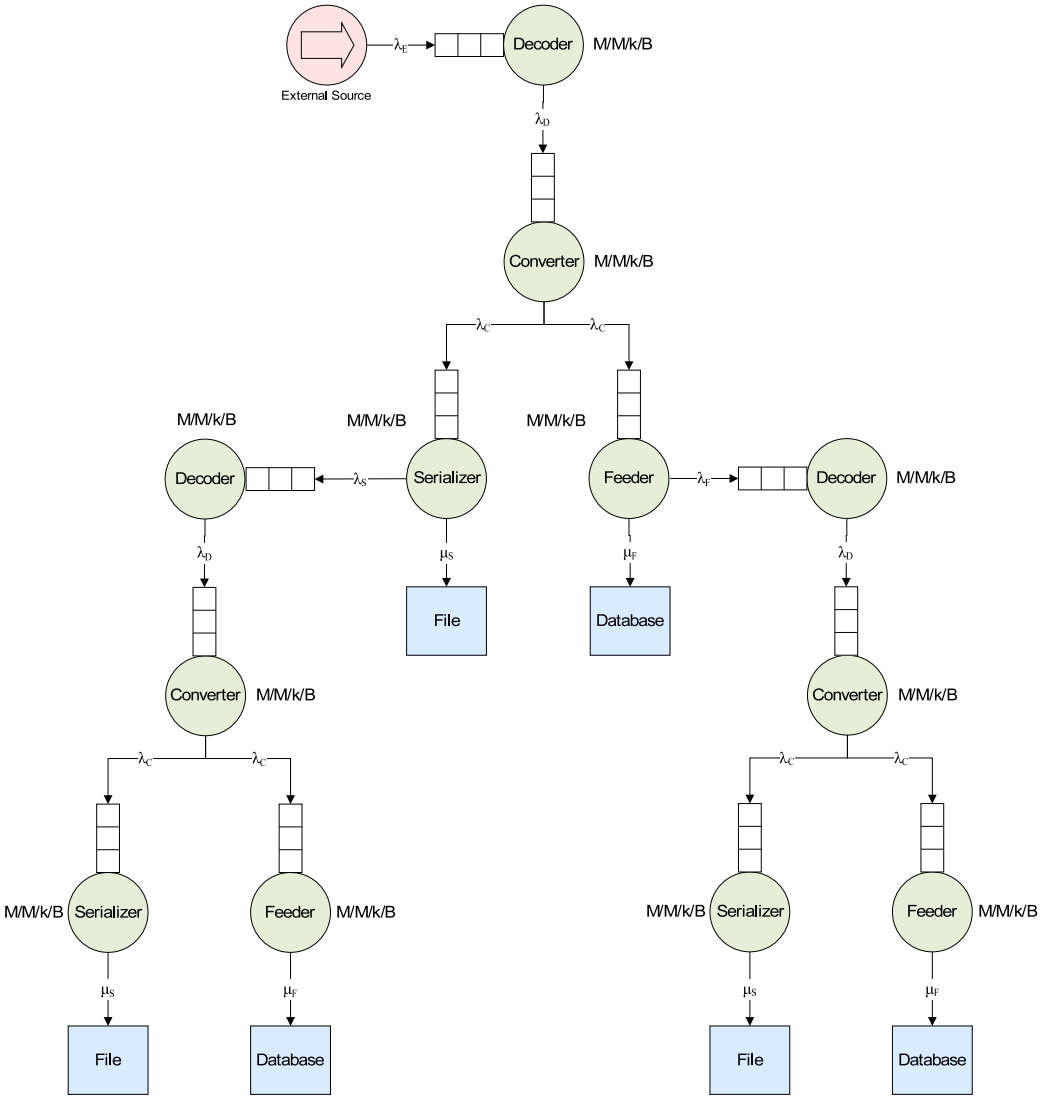


Figure 4.16: New, enlarged queueing network sctructure.

Table 4.24 shows that the optimal configuration of the enlarged queueing network system is 1-1-5-15-1-1-5-14-1-1-5-14. By taken advantage of the 64 cores of host *Zerberus*, the system can handle an external arrival rate of 7393 tickets per second with all 12 nodes under a utilization barrier of 80%.

Node	Threads	Util.[%]
Decoder	1	69.36
Converter	1	60.85
Serializer	5	72.24
Feeder	15	71.73
Decoder	1	74.67
Converter	1	60.85
Serializer	5	71.73
Feeder	14	80.00
Decoder	1	62.86
Converter	1	60.85
Serializer	5	71.73
Feeder	14	80.00

Table 4.24: Optimal configuration of the new enlarged queueing network and utilization of each node type with an external arrival rate of 7393 tickets per second.

The developed calculation module (see Section 4.5.1), implementing the analytical approach (see Section 4.4.3) can easily be changed. A property file, in which node specifications (node type, service rate, queue size and so forth) are defined, is used by the calculation module to find the optimal configuration for the specified queueing network system. Table 4.25 shows the node specifications within the property file for the Decoder node. It can be seen that the external arrival rate can be set to a specific value. The external arrival rate for all other nodes has to be set to 0.

```
node0arrivalRateExt = 1
node0serviceRate = 10658
node0maxServers = 99
node0servers = 1
node0buffers = 1000
node0memory = 0
node0diskSpace = 0
node0diskSpeed = 0
node0network = 0
```

Table 4.25: Property file: node specifications (Decoder).

After node specifications are defined, the queueing network flow-chart can eas-

ily be defined, by specifying the connection between the defined nodes. Table 4.26 shows the setup of the original queueing network.

```
Initial links:  source,destination,probability
links = 4
link0 = -1,0,1
link1 = 0,1,1
link2 = 1,2,1
link3 = 1,3,1
```

Table 4.26: Property file: transitions specifications (Decoder).

Table 4.27 shows the transitions of the extended queueing network, used in this section. By adding new nodes and the belonging transitions, new queueing network scenarios can be optimized by the developed tool in a very easy way.

```
Initial links:  source,destination,probability
links = 12
link0 = -1,0,1
link1 = 0,1,1
link2 = 1,2,1
link3 = 1,3,1
link4 = 2,4,1
link5 = 4,5,1
link6 = 5,6,1
link7 = 5,7,1
link8 = 3,8,1
link9 = 8,9,1
link10 = 9,10,1
link11 = 9,11,1
```

Table 4.27: Property file: transitions specifications (Decoder).

4.5.8 Excursus: Using artificial nodes and a fictional host for further analytical analysis

The two previously described case studies were using the nodes of actual queueing network software. This section presents an approach were fictional nodes are forming a fictional queueing network. This queueing network is then optimized on a fictional host by the analytical approach.
As mentioned in Section 4.5.2, a number of artificial tasks for artificial nodes

were described and implemented within the simulation module. The following approach uses these tasks to build a fictional queueing network. The idea is to show how a completely different queueing network can be optimized by the developed analytical model.

The tasks of the artificial nodes used in this section can be described as follows:

- The node *Random* generates 1000 random numbers for every incoming ticket.
- The node *String A* does 250 string manipulations for every passing ticket, where a string is trimmed into a shorter string.
- The node *String B* does 250 string manipulations for every passing ticket, where two strings are combined into one string.
- The node *Sort* sorts an integer array, consisting of 50000 integer values for every passing ticket.
- The node *Search* searches for an integer value within an integer array, consisting of 50000 integer values for every passing ticket.
- The node *Copy* copies an integer array, consisting of 50000 integer values for every passing ticket.
- The node *Mean* calculates the mean value of an integer array, consisting of 50000 integer values for every passing ticket.
- The node *DB Entry* writes data into an Oracle database for every passing ticket.
- The node *DB Search* searches for a specific entry in an Oracle database for every passing ticket.
- The node *File Entry* writes data to a file for every passing ticket.

Figure 4.17 shows the structure of the fictional queueing network along with the used nodes. An external source sends tickets to the *Random* node. The *Random* node executes its tasks and for every ticket, forwards one ticket to the *String A* node, the *Sort* node, the *DB Entry* node and the *File Entry* node. After tickets pass through the *Random* node, the queueing network therefore

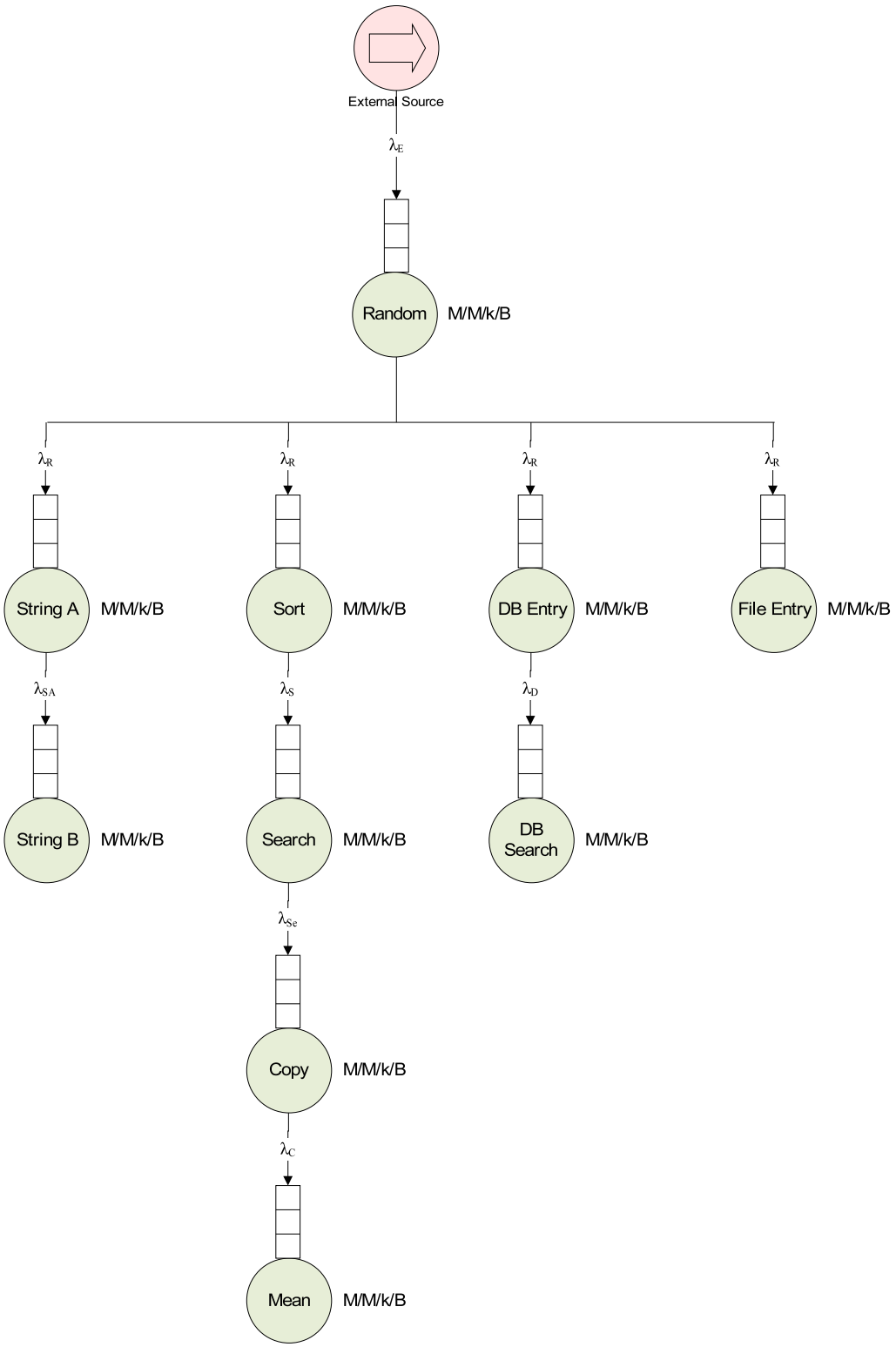


Figure 4.17: New, fictional queueing network sctructure.

consists of four branches.

The first branch deals with tasks regarding string manipulation, therefore the *String A* node forwards every ticket to the *String B* node, where the ticket essentially leaves the network.

The second branch deals with arrays, where the *Sort* node sorts an array and forwards tickets to the *Search* node, which searches within an array. The *Search* node then forwards every ticket to the *Copy* node, where an array is duplicated. Finally, the *Copy* node forwards tickets to the *Mean* node, where a mean value of an array is calculated.

The third branch deals with database operations. The *DB Entry* node writes data into a database table and forwards the ticket to the *DB Search* node, where an SQL search query is executed.

The fourth branch only consists of one node, the *File Entry* node, where data is written into a file.

In the next step, these 10 nodes were simulated on host *Zerberus*, to measure the service rates for each individual node. Table 4.28 shows the service rates of the artificial nodes, used in the following optimization process.

Node	**Service Rate**	**Threads**	**Util.**[%]
Random	388	1	25.77
String A	886	1	11.29
String B	765	1	13.07
Sort	82	1	100.00
Search	3369	1	2.97
Copy	3264	1	3.06
Mean	2807	1	3.56
DB Entry	146	1	68.49
DB Search	764	1	13.09
File Entry	27855	1	0.36

Table 4.28: Service rates and utilizations of the nodes of the fictional queueing network in the initial configuration with an external arrival rate of 100 tickets per second.

Table 4.28 shows that with an external arrival rate of only 100 tickets per second, the *Sort* node is already fully utilized and is therefore the bottleneck of the system. By using the developed analytical approach and recursively

increasing the threads of over-utilized nodes, the system should be able to handle more than 100 tickets per second.

In this hypothetical scenario a fictional host server will be used for optimization. It is assumed that host *Zerberus* now has 256 available cores and can therefore start up to 256 simultaneous threads.

Table 4.29 shows the threads per node and the node utilizations of the final and optimal configuration.

Figure 4.18 shows the service rates of the nodes of the queueing network

Node	**Service Rate**	**Threads**	**Util.**[%]
Random	388	25	79.13
String A	886	11	78.76
String B	765	13	77.18
Sort	82	117	80.00
Search	3369	3	75.95
Copy	3264	3	78.39
Mean	2807	4	68.36
DB Entry	146	66	79.66
DB Search	764	13	77.29
File Entry	27855	1	27.56

Table 4.29: Service rates and utilizations of the nodes of the fictional queueing network in the optimal configuration with an external arrival rate of 7676 tickets per second.

at the initial configuration of one thread per node. It can be seen that even though, for example, the node *File Entry* has a service rate of 27855 and can therefore handle up to 27855 tickets per second, the actual service rate is just 388 tickets per second. Tickets enter the queueing network with, for example, an external arrival rate of 10000 tickets per second. The first node *Random* however, can only handle 388 tickets per second. Therefore, the node *Random* forwards just 388 tickets per second to the *File Entry*, as well as the *String A*, the *Sort* and the *DB Entry* node.

The *File Entry* node has a service rate of 27855 and can therefore handle all 388 tickets per second coming in from the *Random* node, but has no chance to handle the 10000 tickets coming in from the external source.

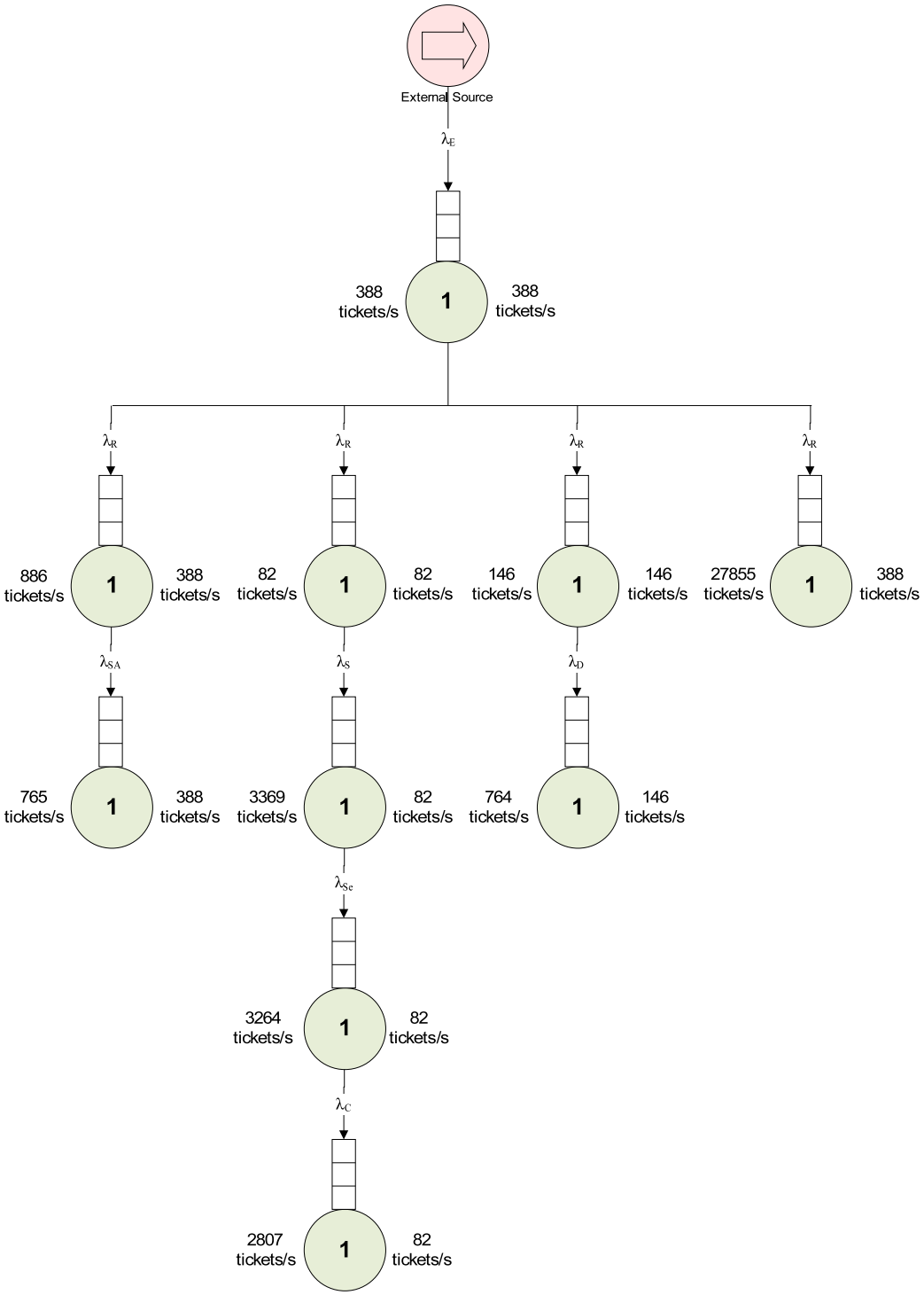


Figure 4.18: New, fictional queueing network sctructure.

From the 388 tickets per second coming into the *Sort* node, only 82 can be handled per second by the *Sort* node. Therefore, all following nodes (the *Search* node, the *Copy* node and the *Mean* node) within the dedicated branch, which all show service rates over 2800 tickets per second, can only handle 82 tickets per second.

Therefore, all four nodes (the *Sort* node, the *Search* node, the *Copy* node and the *Mean* node) within the branch benefit from increasing the *Sort* node and can all of a sudden process more tickets than before.

After the optimization, it can be seen that the system can now handle 7676 tickets per second with all 10 nodes under the utilization barrier of 80%. Table 4.29, along with the mentioned service rate situation (see Figure 4.18 and Appendix E), show, that the longer the optimization goes on and the more cores are available, the different threads per node are leading to the fact that the utilizations of the nodes are becoming more and more even and level themselves under the utilization barrier of 80%.

The problem with the actual service rates of the nodes are not a problem of the actual DFE software, because the *Feeder* node, which was the bottleneck of the entire system, was located at the end of a branch of the queueing network. The fictional setup used in this section, shows that with slow nodes at the top of the queueing network tree, the entire underlying network suffers. Faster nodes can then process only a lower percentage of the actual incoming tickets. This section therefore shows the advantage of increasing always the most over-utilized node, especially with slow nodes at the top of a queueing network. The stepwise approach balances the actual service rates of all nodes within the network, so the software can handle the most incoming tickets as possible.

With 256 available cores, the optimal configuration of the fictional queueing network, using the artificial nodes and tasks, is 25-11-13-117-3-3-4-66-13-1.

4.5.9 Conclusion

This chapter proposed an offline optimization approach, to find the best configuration for a queueing-network software on a given host. Also, an approach for an online optimization tool was presented. The optimization approach can be summarized as follows:

- *Calculation* - The calculation module represents the fundamental basis for the optimization approach. The module implements all the necessary performance metrics for a M/M/k/B queueing network, calculates over-utilized nodes and adds new threads as often as possible to reach an optimization goal.
- *Measurement* - The measurement module implements certain tasks that can be assigned to nodes of a queueing network software and measures their service rates on a given host. Given the fact that the external arrival rate, along with the service rates of the nodes, are the driving factors for the utilization of nodes, this is an important step to make the optimization more precise and suitable for a given software on a given host.
- *Simulation* - The simulation module should be used, when the software is about to be installed on a new system or if the used software does not offer possibilities to retrieve performance metrics of the real system. E.g: when the DFE is installed on a new system, and there are no incoming tickets yet, the simulation module can be started to get an initial configuration.
- *Online Optimization* - If there is a possibility to start and restart or alter the queueing network software at runtime, the online optimization should be used to guarantee an optimal software configuration at all times. Performance metrics like the service rates of all nodes should be extracted from the real system. If there is no possibility for that, the measurement module should simulate node tasks and thereby measuring artificial service rates. The calculation module should then calculate the optimal configuration of the system and restart the software. These steps should be repeated constantly, e.g.: when an increase/decrease of the external arrival rate is detected.

Chapter 5

Conclusion

The goal of this dissertation was to improve the reliability and the performance of telecommunications systems. The first part of my work focused on improving the reliability of a Voice over IP server. By working with and testing VoIP servers, I soon recognized that different SIP dialects were causing problems for VoIP devices. Different SIP dialects evolve through the openness of the SIP protocol. On the one hand, software developers use different interpretations of the protocol in their products, on the other hand, VoIP servers may not implement the full standard flawlessly right from the start. Therefore, it might be possible that two SIP devices may not be able to communicate with each other, even though they use the same protocol.

To improve this situation an autonomic and self-learning tool called Babel-SIP was developed in Java, which uses C4.5 decision trees to classify incoming SIP messages and detect headers or header parameters that may be problematic.

Experiments showed that Babel-SIP was able to improve the acceptance rate of incoming SIP messages drastically and therefore improve the problematic situation of different SIP dialects.

Furthermore, decision trees derived after the conducted experiments (or during online use in real systems) show which headers, header parameters, values or parameter-combinations are problematic. The gained knowledge can therefore improve software systems, either on the server side by finding bugs in VoIP

server software, or on the client side by finding errors within VoIP end devices.

The focus in this dissertation was improving the reliability of VoIP servers, but the presented ideas can be used within other information technology areas as well. Network protocols that are using similar text-message-based communication can also benefit from the presented self-learning approach using decision trees. Also, the ideas presented in my dissertation can lead to the development of adaptive network protocols. By taking the idea of Babel-SIP and refining the used self-learning techniques, a universal solution for communication problems between two devices using the same protocol in early stages of development may be found, which could possibly have an effect on the entire field of information technology.

The second part of my work dealt with server software used to handle incoming Diameter tickets of mobile telephone devices. The software handles these incoming tickets and stores the included data. The software itself is based on a queueing network, where each incoming ticket wanders through the queueing network and is handled by interconnected nodes. Each node has a unique task, which is executed once for each passing ticket.

Given the fact that these tasks are lightweight processes, each node can be parallelized, so that every node thread can be executed on a single core. The goal of the second part of my work was therefore to improve the performance of such a telecommunications system, by finding the optimal configuration of threads per node on a given computing platform. By parallelizing the work to a number of threads on a number of cores, the nodes total service rate can be increased and therefore the throughput of the system can be increased.

A combination of an analytical model, a measurement and a simulation approach was used to achieve this goal. Experiments along with test runs of every individual approach showed that an autonomic and self-adaptive test tool can find the optimal configuration for software based on a queueing network. By recursively adding threads to over-utilized nodes, the current bottleneck of the system can be found and potentially eliminated, leading to a new bottleneck. By doing that, the bottleneck is relocated within the software as long as hard-

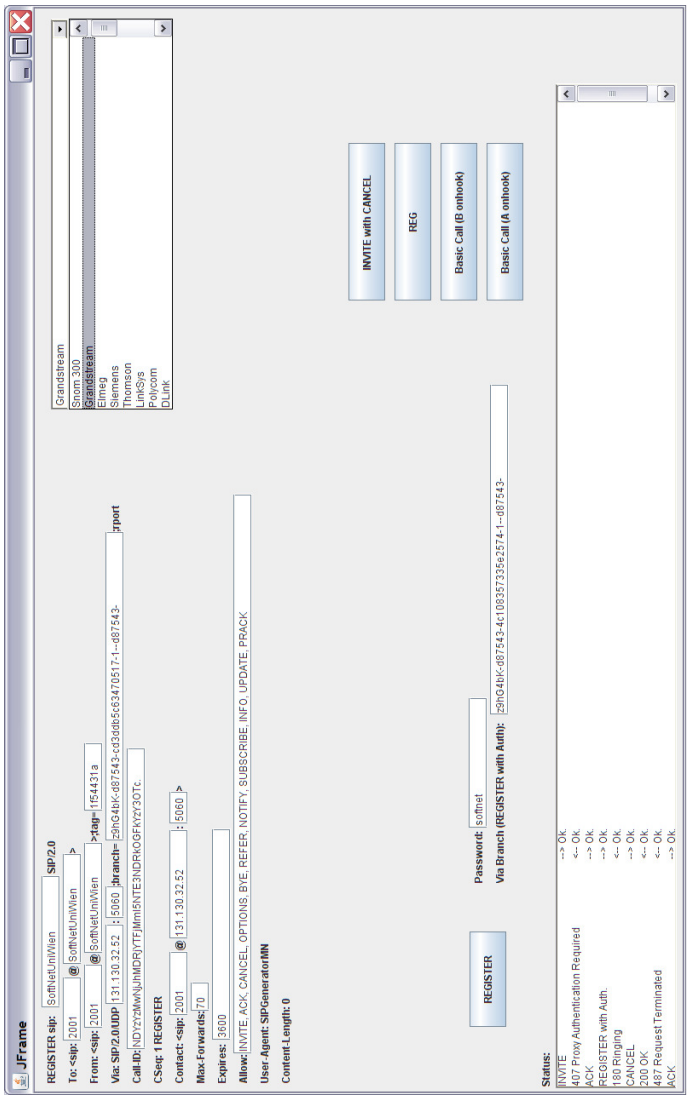
ware resources can be used or an optimization goal is reached. In that way, the number of incoming requests which can be handled by the system is increased, which improves the performance of the system.

The work on this dissertation furthermore showed me that the use of autonomic, self-adaptive and self-learning systems can improve the availability and the performance of not only VoIP systems, but these techniques are nowadays used in more and more information technology areas.

The easy-to-adapt structure of the developed optimization tool, allows it to be used for a wide range of software that is based on a multi-thread data-flow queueing network system. By simply using different node types, adding new node types and rearranging the structure of the flow graph, the optimization tool can be used for other software to find the optimal configuration on a given hardware platform.

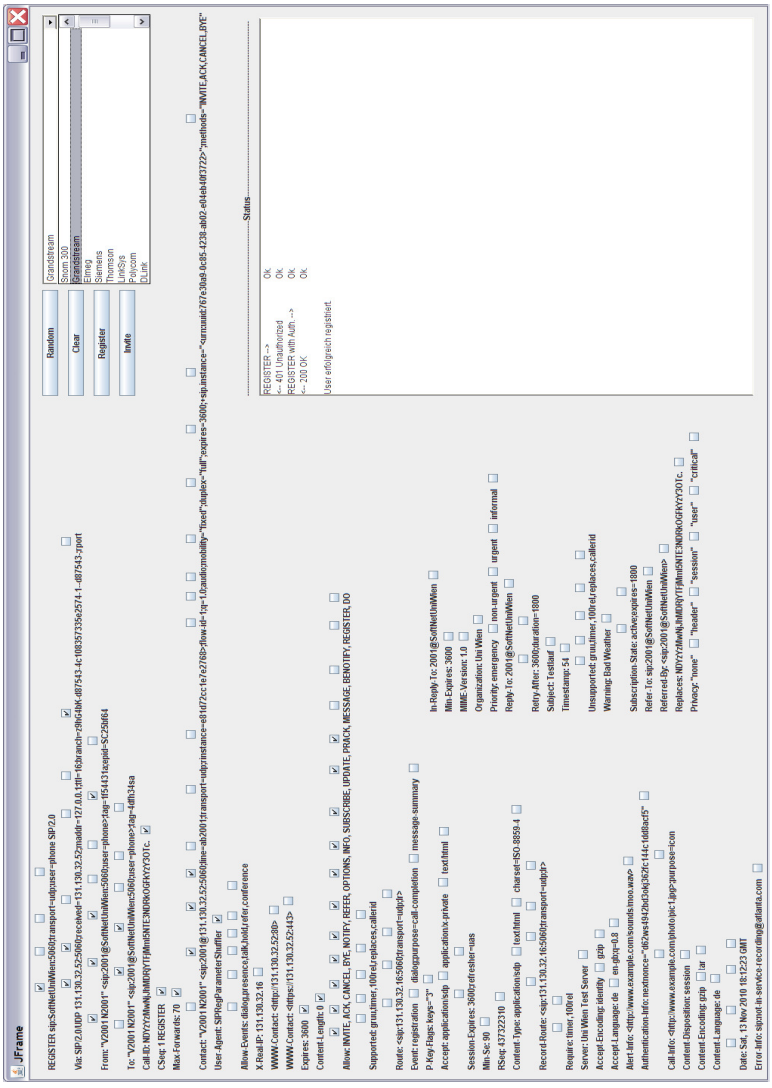
Appendix A

SIPGenerator GUI



Appendix B

SIPParameterShuffler GUI



Appendix C

C4.5 tree, REGISTER messages

```
CSeq <= 0: REJ. (104)
CSeq > 0
| Call-ID <= 0: REJ. (101)
| Call-ID > 0
| | Replaces <= 0
| | | Accept_level <= 0
| | | | To_IP <= 0
| | | | | Contact_Port <= 1111: ACC. (18/2)
| | | | | Contact_Port > 1111: REJ. (74/1)
| | | | To_IP > 0
| | | | | Accept_application/x-private <= 0
| | | | | | Contact_IP <= 0: ACC. (47/15)
| | | | | | Contact_IP > 0
| | | | | | | Min-Expires <= 20
| | | | | | | | Via_received <= 0
| | | | | | | | | Reply-To <= 0
| | | | | | | | | | Via_IP <= 0
| | | | | | | | | | | Via <= 0: REJ. (15)
| | | | | | | | | | | Via > 0: ACC. (88/1)
| | | | | | | | | | Via_IP > 0
| | | | | | | | | | | Privacy_none <= 0
| | | | | | | | | | | | Record-Route <= 0
| | | | | | | | | | | | | Allow_BENOTIFY <= 0
| | | | | | | | | | | | | | Referred-By <= 0
| | | | | | | | | | | | | | | Accept <= 0
| | | | | | | | | | | | | | | | Event_dialog <= 0
| | | | | | | | | | | | | | | | | Warning <= 0: ACC. (2054/3)
| | | | | | | | | | | | | | | | | Warning > 0
| | | | | | | | | | | | | | | | | | Accept-Language <= 0: ACC. (30)
| | | | | | | | | | | | | | | | | | Accept-Language > 0: REJ. (2)
| | | | | | | | | | | | | | | | Event_dialog > 0
| | | | | | | | | | | | | | | | | Content-Type <= 0: ACC. (28)
| | | | | | | | | | | | | | | | | Content-Type > 0: REJ. (2)
| | | | | | | | | | | | | | | | Accept > 0
| | | | | | | | | | | | | | | | | Allow-Events <= 0
```

```
| | | | | | | | | | | | | | | | | | | Request-Line_transport <= 0: ACC. (51)
| | | | | | | | | | | | | | | | | | | Request-Line_transport > 0
| | | | | | | | | | | | | | | | | | | | Accept_app/sdp <= 0: REJ. (14)
| | | | | | | | | | | | | | | | | | | | Accept_app/sdp > 0: ACC. (15)
| | | | | | | | | | | | | | | | | | | Allow-Events > 0
| | | | | | | | | | | | | | | | | | | | Contact_Port <= 1111
| | | | | | | | | | | | | | | | | | | | From_IP <= 0: REJ. (2)
| | | | | | | | | | | | | | | | | | | | From_IP > 0: ACC. (38)
| | | | | | | | | | | | | | | | | | | | Contact_Port > 1111: ACC. (338)
| | | | | | | | | | | | | | | | | | | Referred-By > 0
| | | | | | | | | | | | | | | | | | | | Content-Type_charset <= 0: ACC. (55)
| | | | | | | | | | | | | | | | | | | | Content-Type_charset > 0: REJ. (4)
| | | | | | | | | | | | | | | | | | | Allow_BENOTIFY > 0
| | | | | | | | | | | | | | | | | | | Allow_INVITE <= 0
| | | | | | | | | | | | | | | | | | | Via_rport <= 0: REJ. (11)
| | | | | | | | | | | | | | | | | | | Via_rport > 0: ACC. (30)
| | | | | | | | | | | | | | | | | | | Allow_INVITE > 0
| | | | | | | | | | | | | | | | | | | Alert-Info <= 0
| | | | | | | | | | | | | | | | | | | | Content-Type <= 0: ACC. (163/1)
| | | | | | | | | | | | | | | | | | | | Content-Type > 0
| | | | | | | | | | | | | | | | | | | | | Content-Type_charset <= 0: ACC. (20)
| | | | | | | | | | | | | | | | | | | | | Content-Type_charset > 0: REJ. (2)
| | | | | | | | | | | | | | | | | | | Alert-Info > 0
| | | | | | | | | | | | | | | | | | | | From_Name <= 0: ACC. (23)
| | | | | | | | | | | | | | | | | | | | From_Name > 0: REJ. (3)
| | | | | | | | | | | | | | | | | | Record-Route > 0
| | | | | | | | | | | | | | | | | | | Proxy-Authorization <= 0
| | | | | | | | | | | | | | | | | | | | Priority_non-urgent <= 0: ACC. (213/1)
| | | | | | | | | | | | | | | | | | | | Priority_non-urgent > 0
| | | | | | | | | | | | | | | | | | | | Contact_q <= 0: REJ. (3)
| | | | | | | | | | | | | | | | | | | | Contact_q > 0: ACC. (15)
| | | | | | | | | | | | | | | | | | | Proxy-Authorization > 0: REJ. (14)
| | | | | | | | | | | | | | | | | | Privacy_none > 0
| | | | | | | | | | | | | | | | | | | Error-Info <= 0: ACC. (91/1)
| | | | | | | | | | | | | | | | | | | Error-Info > 0: REJ. (10)
| | | | | | | | | | | | | | | | | | Reply-To > 0
| | | | | | | | | | | | | | | | | | | Contact_sip.instance <= 0
| | | | | | | | | | | | | | | | | | | Allow-Events_hold <= 0: ACC. (163)
| | | | | | | | | | | | | | | | | | | Allow-Events_hold > 0
| | | | | | | | | | | | | | | | | | | | Request-Line_user <= 0: ACC. (9)
| | | | | | | | | | | | | | | | | | | | Request-Line_user > 0: REJ. (2)
| | | | | | | | | | | | | | | | | | | Contact_sip.instance > 0: REJ. (30/1)
| | | | | | | | | | | | | | | | | | Via_received > 0
| | | | | | | | | | | | | | | | | | | Contact_q <= 0: ACC. (76)
| | | | | | | | | | | | | | | | | | | Contact_q > 0: REJ. (17)
| | | | | | | | | | | | | | | | | | Min-Expires > 20
| | | | | | | | | | | | | | | | | | | Contact_line <= 0
| | | | | | | | | | | | | | | | | | | Allow_ACK <= 0: ACC. (63)
| | | | | | | | | | | | | | | | | | | Allow_ACK > 0: REJ. (4/1)
| | | | | | | | | | | | | | | | | | | Contact_line > 0: REJ. (24)
| | | | | | | | | | | | | | | | | | Accept_application/x-private > 0
```

```
| | | | | | Contact_mobility <= 0
| | | | | | | Via_ttl <= 1
| | | | | | | | Subscription-State <= 0: ACC. (198/2)
| | | | | | | | Subscription-State > 0: REJ. (14)
| | | | | | | Via_ttl > 1: REJ. (15)
| | | | | | Contact_mobility > 0: REJ. (41/1)
| | | Accept_level > 0: REJ. (30)
| | Replaces > 0: REJ. (38)
```

Appendix D

C4.5 tree, INVITE messages

```
To_IP <= 0: REJ. (720)
To_IP > 0
| Replaces <= 0
| | Accept_application/x-private <= 0
| | | Priority_non-urgent <= 0
| | | | From_IP <= 0
| | | | | Allow-Events_conference <= 0: REJ. (310/10)
| | | | | Allow-Events_conference > 0: ACC. (100)
| | | | From_IP > 0
| | | | | Priority_urgent <= 0
| | | | | | Allow-Events_refer <= 0
| | | | | | | Timestamp <= 0
| | | | | | | | Event_purpose <= 0
| | | | | | | | | Supported <= 0
| | | | | | | | | | To_Name <= 0
| | | | | | | | | | | P-Key-Flags <= 0
| | | | | | | | | | | | Route <= 0
| | | | | | | | | | | | | Accept-Encoding <= 0
| | | | | | | | | | | | | | Require_100rel <= 0
| | | | | | | | | | | | | | | Via_branch <= 0
| | | | | | | | | | | | | | | | Expires <= 0
| | | | | | | | | | | | | | | | | Server <= 0: ACC. (290)
| | | | | | | | | | | | | | | | | Server > 0: REJ. (30/10)
| | | | | | | | | | | | | | | | Expires > 0: REJ. (30)
| | | | | | | | | | | | | | | Via_branch > 0
| | | | | | | | | | | | | | | | Allow_REGISTER <= 0
| | | | | | | | | | | | | | | | | Organization <= 0
| | | | | | | | | | | | | | | | | | Accept <= 0: ACC. (1960/30)
| | | | | | | | | | | | | | | | | | Accept > 0
| | | | | | | | | | | | | | | | | | | Content-T_text/html <= 0: ACC.(180)
| | | | | | | | | | | | | | | | | | | Content-T_text/html > 0: REJ. (20)
| | | | | | | | | | | | | | | | | Organization > 0: ACC. (200/20)
| | | | | | | | | | | | | | | | Allow_REGISTER > 0: ACC. (170/20)
| | | | | | | | | | | | | | Require_100rel > 0
| | | | | | | | | | | | | | | Contact_Port <= 0: REJ. (30)
| | | | | | | | | | | | | | | Contact_Port > 0
```

```
| | | | | | | | | | | | | | | | | Via_maddr <= 0: ACC. (240)
| | | | | | | | | | | | | | | | | Via_maddr > 0: REJ. (20)
| | | | | | | | | | | | | | | Accept-Encoding > 0
| | | | | | | | | | | | | | | | Min-SE <= 0: REJ. (50/10)
| | | | | | | | | | | | | | | | Min-SE > 0: ACC. (130)
| | | | | | | | | | | | | | | Route > 0
| | | | | | | | | | | | | | | | CSeq <= 0: REJ. (40)
| | | | | | | | | | | | | | | | CSeq > 0
| | | | | | | | | | | | | | | | Content-Type_text/html <= 0
| | | | | | | | | | | | | | | | Require_timer <= 0
| | | | | | | | | | | | | | | | Allow_UPDATE <= 0
| | | | | | | | | | | | | | | | Allow-Events_hold <= 0
| | | | | | | | | | | | | | | | Accept_text/html <= 0
| | | | | | | | | | | | | | | | Allow-Events_dialog <= 0
| | | | | | | | | | | | | | | | Content-Enc_tar <= 0: REJ. (20)
| | | | | | | | | | | | | | | | Content-Enc_tar > 0: ACC. (30)
| | | | | | | | | | | | | | | | Allow-Ev_dialog > 0: ACC. (40/10)
| | | | | | | | | | | | | | | | Accept_text/html > 0: ACC. (20)
| | | | | | | | | | | | | | | | Allow-Events_hold > 0: ACC. (40)
| | | | | | | | | | | | | | | | Allow_UPDATE > 0: ACC. (150)
| | | | | | | | | | | | | | | | Require_timer > 0: REJ. (20)
| | | | | | | | | | | | | | | | Content-Type_text/html > 0: REJ. (20)
| | | | | | | | | | | | | | | P-Key-Flags > 0
| | | | | | | | | | | | | | | | Privacy_user <= 0
| | | | | | | | | | | | | | | | In-Reply-To <= 0
| | | | | | | | | | | | | | | | Via_maddr <= 0: REJ. (90/10)
| | | | | | | | | | | | | | | | Via_maddr > 0: ACC. (20)
| | | | | | | | | | | | | | | | In-Reply-To > 0: ACC. (30)
| | | | | | | | | | | | | | | | Privacy_user > 0: ACC. (60)
| | | | | | | | | | | | | | | To_Name > 0: ACC. (1550/10)
| | | | | | | | | | | | | | | Supported > 0
| | | | | | | | | | | | | | | | Content-Type_text/html <= 0
| | | | | | | | | | | | | | | | WWW-Contact_https <= 0
| | | | | | | | | | | | | | | | Content-Type_charset <= 0
| | | | | | | | | | | | | | | | Subject <= 0
| | | | | | | | | | | | | | | | CSeq <= 0: REJ. (30)
| | | | | | | | | | | | | | | | CSeq > 0
| | | | | | | | | | | | | | | | Contact_rinstance <= 0
| | | | | | | | | | | | | | | | Allow_OPTIONS <= 0
| | | | | | | | | | | | | | | | Allow-Events_presence <= 0
| | | | | | | | | | | | | | | | Via_rport <= 0: ACC. (30/10)
| | | | | | | | | | | | | | | | Via_rport > 0
| | | | | | | | | | | | | | | | Contact_audio <= 0: REJ. (30/10)
| | | | | | | | | | | | | | | | Contact_audio > 0: ACC. (50/20)
| | | | | | | | | | | | | | | | Allow-Events_presence > 0: ACC. (50)
| | | | | | | | | | | | | | | | Allow_OPTIONS > 0: ACC. (90)
| | | | | | | | | | | | | | | | Contact_rinstance > 0: REJ. (30/10)
| | | | | | | | | | | | | | | Subject > 0: REJ. (40)
| | | | | | | | | | | | | | | Content-Type_charset > 0: REJ. (50)
| | | | | | | | | | | | | | | WWW-Contact_https > 0
| | | | | | | | | | | | | | | | Privacy_header <= 0: ACC. (580/20)
```

```
| | | | | | | | | | | | Privacy_header > 0
| | | | | | | | | | | | | Contact_sip.instance <= 0: REJ. (40/10)
| | | | | | | | | | | | | Contact_sip.instance > 0: ACC. (20)
| | | | | | | | | | Content-Type_text/html > 0: REJ. (60)
| | | | | | | | Event_purpose > 0
| | | | | | | | | To_Name <= 0
| | | | | | | | | | Contact_q <= 0
| | | | | | | | | | | Priority <= 0
| | | | | | | | | | | | Unsupported <= 0
| | | | | | | | | | | | | WWW-Contact_http <= 0: REJ. (20)
| | | | | | | | | | | | | WWW-Contact_http > 0: ACC. (40/10)
| | | | | | | | | | | | Unsupported > 0: ACC. (50)
| | | | | | | | | | | | Priority > 0: REJ. (50)
| | | | | | | | | | | Contact_q > 0: ACC. (140)
| | | | | | | | | | To_Name > 0: REJ. (30)
| | | | | | | | Timestamp > 0
| | | | | | | | | Allow-Events_hold <= 0: ACC. (220/10)
| | | | | | | | | Allow-Events_hold > 0: REJ. (100)
| | | | | | Allow-Events_refer > 0
| | | | | | | | CSeq <= 0: REJ. (50)
| | | | | | | | CSeq > 0
| | | | | | | | | Retry-After <= 0
| | | | | | | | | | Contact_flow-id <= 0
| | | | | | | | | | | Referred-By <= 0
| | | | | | | | | | | | Alert-Info <= 0
| | | | | | | | | | | | | Via_rport <= 0
| | | | | | | | | | | | | | In-Reply-To <= 0
| | | | | | | | | | | | | | | Contact_Port <= 0: ACC. (20)
| | | | | | | | | | | | | | | Contact_Port > 0: REJ. (50/20)
| | | | | | | | | | | | | | In-Reply-To > 0: REJ. (20)
| | | | | | | | | | | | | Via_rport > 0: ACC. (80/10)
| | | | | | | | | | | | Alert-Info > 0: REJ. (20)
| | | | | | | | | | | Referred-By > 0: ACC. (30)
| | | | | | | | | | Contact_flow-id > 0: ACC. (80)
| | | | | | | | | Retry-After > 0: REJ. (30)
| | | | | Priority_urgent > 0
| | | | | | Request-Line_transport <= 0: REJ. (250)
| | | | | | Request-Line_transport > 0: ACC. (90)
| | | Priority_non-urgent > 0
| | | | Via_rport <= 0
| | | | | Contact_expires <= 1: ACC. (20)
| | | | | Contact_expires > 1: REJ. (20)
| | | | Via_rport > 0: REJ. (240)
| | Accept_application/x-private > 0: REJ. (430)
| Replaces > 0: REJ. (480)
```

Appendix E

Configurations of a fictional system (see Section 4.5.8): Cumulated service rates for all nodes and the maximum external arrival rate where all nodes are utilized under 80%.

The following table shows the recursive working method of the developed optimization tool. The tool finds the most over-utilized node and adds one thread to the node. The table therefore shows each of the 256 configurations with the according service rates and the current throughput possible.

Configuration: Number of threads per node (Random-String A-String B-Sort-Search-Copy-Mean-DB Entry-DB Search-File Entry).

Threads: Number of all threads of the current configuration.

Individual Service Rates: Service rates of all ten nodes for one individual thread.

Cumulated Service Rates: Service rates of all ten nodes for all threads (cumulated service rate = individual service rate * number of threads).

Max.Ext.Arr.Rate: The maximum external arrival rate the system can cope with, so that all nodes show a utilization of under 80%.

Configuration	Threads	Random	String A	String B	Sort	Search	Copy	Mean	DB Entry	DB Search	File Entry	Max.Ext.Arr.Rate
1-1-1-1-1-1-1-1-1-1	10	388	886	765	82	3369	3264	2807	146	764	27855	65
1-1-1-2-1-1-1-1-1-1	11	388	886	765	82	3369	3264	2807	146	764	27855	116
1-1-1-2-1-1-1-2-1-1	12	388	886	765	82	3369	3264	2807	146	764	27855	131
1-1-1-3-1-1-1-2-1-1	13	388	886	765	82	3369	3264	2807	146	764	27855	196
1-1-1-4-1-1-1-2-1-1	14	388	886	765	82	3369	3264	2807	146	764	27855	233
1-1-1-4-1-1-1-3-1-1	15	388	886	765	82	3369	3264	2807	146	764	27855	262
1-1-1-5-1-1-1-3-1-1	16	388	886	765	82	3369	3264	2807	146	764	27855	310
2-1-1-5-1-1-1-3-1-1	17	388	886	765	82	3369	3264	2807	146	764	27855	328
2-1-1-6-1-1-1-3-1-1	18	388	886	765	82	3369	3264	2807	146	764	27855	350
2-1-1-6-1-1-1-4-1-1	19	388	886	765	82	3369	3264	2807	146	764	27855	393
2-1-1-7-1-1-1-4-1-1	20	388	886	765	82	3369	3264	2807	146	764	27855	459
2-1-1-8-1-1-1-4-1-1	21	388	886	765	82	3369	3264	2807	146	764	27855	467
2-1-1-8-1-1-1-5-1-1	22	388	886	765	82	3369	3264	2807	146	764	27855	524
2-1-1-9-1-1-1-5-1-1	23	388	886	765	82	3369	3264	2807	146	764	27855	584
2-1-1-9-1-1-1-6-1-1	24	388	886	765	82	3369	3264	2807	146	764	27855	590
2-1-1-10-1-1-1-6-1-1	25	388	886	765	82	3369	3264	2807	146	764	27855	611
2-1-1-10-1-1-1-6-2-1	26	388	886	765	82	3369	3264	2807	146	764	27855	612

Configuration	Threads	Random	String A	String B	Sort	Search	Copy	Mean	DB Entry	DB Search	File Entry	Random	String A	String B	Sort	Search	Copy	Mean	DB Entry	DB Search	File Entry	Max.Ext.Arr.Rate
2-1-2-10-1-1-1-6-2-1	27	388	886	765	82	3369	3264	2807	146	764	27855	776	886	1530	820	3369	3264	2807	876	1528	27855	620
3-1-2-10-1-1-1-6-2-1	28	388	886	765	82	3369	3264	2807	146	764	27855	1164	886	1530	820	3369	3264	2807	876	1528	27855	656
3-1-2-11-1-1-1-6-2-1	29	388	886	765	82	3369	3264	2807	146	764	27855	1164	886	1530	902	3369	3264	2807	876	1528	27855	700
3-1-2-11-1-1-1-7-2-1	30	388	886	765	82	3369	3264	2807	146	764	27855	1164	886	1530	902	3369	3264	2807	1022	1528	27855	708
3-2-2-11-1-1-1-7-2-1	31	388	886	765	82	3369	3264	2807	146	764	27855	1164	1772	1530	902	3369	3264	2807	1022	1528	27855	721
3-2-2-12-1-1-1-7-2-1	32	388	886	765	82	3369	3264	2807	146	764	27855	1164	1772	1530	984	3369	3264	2807	1022	1528	27855	787
3-2-2-13-1-1-1-7-2-1	33	388	886	765	82	3369	3264	2807	146	764	27855	1164	1772	1530	1066	3369	3264	2807	1022	1528	27855	817
3-2-2-13-1-1-1-8-2-1	34	388	886	765	82	3369	3264	2807	146	764	27855	1164	1772	1530	1066	3369	3264	2807	1168	1528	27855	852
3-2-2-14-1-1-1-8-2-1	35	388	886	765	82	3369	3264	2807	146	764	27855	1164	1772	1530	1148	3369	3264	2807	1168	1528	27855	918
3-2-2-15-1-1-1-8-2-1	36	388	886	765	82	3369	3264	2807	146	764	27855	1164	1772	1530	1230	3369	3264	2807	1168	1528	27855	931
4-2-2-15-1-1-1-8-2-1	37	388	886	765	82	3369	3264	2807	146	764	27855	1552	1772	1530	1230	3369	3264	2807	1168	1528	27855	934
4-2-2-15-1-1-1-9-2-1	38	388	886	765	82	3369	3264	2807	146	764	27855	1552	1772	1530	1230	3369	3264	2807	1314	1528	27855	984
4-2-2-16-1-1-1-9-2-1	39	388	886	765	82	3369	3264	2807	146	764	27855	1552	1772	1530	1312	3369	3264	2807	1314	1528	27855	1050
4-2-2-17-1-1-1-9-2-1	40	388	886	765	82	3369	3264	2807	146	764	27855	1552	1772	1530	1394	3369	3264	2807	1314	1528	27855	1051
4-2-2-17-1-1-1-10-2-1	41	388	886	765	82	3369	3264	2807	146	764	27855	1552	1772	1530	1394	3369	3264	2807	1460	1528	27855	1115
4-2-2-18-1-1-1-10-2-1	42	388	886	765	82	3369	3264	2807	146	764	27855	1552	1772	1530	1476	3369	3264	2807	1460	1528	27855	1168
4-2-2-18-1-1-1-11-2-1	43	388	886	765	82	3369	3264	2807	146	764	27855	1552	1772	1530	1476	3369	3264	2807	1606	1528	27855	1181

Configuration	Threads	Random	String A	String B	Sort	Search	Copy	Mean	DB Entry	DB Search	File Entry	Random	String A	String B	Sort	Search	Copy	Mean	DB Entry	DB Search	File Entry	Max.Ext.Arr.Rate
4-2-2-19-1-1-1-11-2-1	44	388	886	765	82	3369	3264	2807	146	764	27855	1552	1772	1530	1558	3369	3264	2807	1606	1528	27855	1222
4-2-2-19-1-1-1-11-3-1	45	388	886	765	82	3369	3264	2807	146	764	27855	1552	1772	1530	1558	3369	3264	2807	1606	2292	27855	1224
4-2-3-19-1-1-1-11-3-1	46	388	886	765	82	3369	3264	2807	146	764	27855	1552	1772	2295	1558	3369	3264	2807	1606	2292	27855	1242
5-2-3-19-1-1-1-11-3-1	47	388	886	765	82	3369	3264	2807	146	764	27855	1940	1772	2295	1558	3369	3264	2807	1606	2292	27855	1246
5-2-3-20-1-1-1-11-3-1	48	388	886	765	82	3369	3264	2807	146	764	27855	1940	1772	2295	1640	3369	3264	2807	1606	2292	27855	1285
5-2-3-20-1-1-1-12-3-1	49	388	886	765	82	3369	3264	2807	146	764	27855	1940	1772	2295	1640	3369	3264	2807	1752	2292	27855	1312
5-2-3-21-1-1-1-12-3-1	50	388	886	765	82	3369	3264	2807	146	764	27855	1940	1772	2295	1722	3369	3264	2807	1752	2292	27855	1378
5-2-3-22-1-1-1-12-3-1	51	388	886	765	82	3369	3264	2807	146	764	27855	1940	1772	2295	1804	3369	3264	2807	1752	2292	27855	1402
5-2-3-22-1-1-1-13-3-1	52	388	886	765	82	3369	3264	2807	146	764	27855	1940	1772	2295	1804	3369	3264	2807	1898	2292	27855	1418
5-3-3-22-1-1-1-13-3-1	53	388	886	765	82	3369	3264	2807	146	764	27855	1940	2658	2295	1804	3369	3264	2807	1898	2292	27855	1443
5-3-3-23-1-1-1-13-3-1	54	388	886	765	82	3369	3264	2807	146	764	27855	1940	2658	2295	1886	3369	3264	2807	1898	2292	27855	1509
5-3-3-24-1-1-1-13-3-1	55	388	886	765	82	3369	3264	2807	146	764	27855	1940	2658	2295	1968	3369	3264	2807	1898	2292	27855	1518
5-3-3-24-1-1-1-14-3-1	56	388	886	765	82	3369	3264	2807	146	764	27855	1940	2658	2295	1968	3369	3264	2807	2044	2292	27855	1552
6-3-3-24-1-1-1-14-3-1	57	388	886	765	82	3369	3264	2807	146	764	27855	2328	2658	2295	1968	3369	3264	2807	2044	2292	27855	1574
6-3-3-25-1-1-1-14-3-1	58	388	886	765	82	3369	3264	2807	146	764	27855	2328	2658	2295	2050	3369	3264	2807	2044	2292	27855	1635
6-3-3-25-1-1-1-15-3-1	59	388	886	765	82	3369	3264	2807	146	764	27855	2328	2658	2295	2050	3369	3264	2807	2190	2292	27855	1640
6-3-3-26-1-1-1-15-3-1	60	388	886	765	82	3369	3264	2807	146	764	27855	2328	2658	2295	2132	3369	3264	2807	2190	2292	27855	1706

Configuration	Threads	Random	String A	String B	Sort	Search	Copy	Mean	DB Entry	DB Search	File Entry	Random	String A	String B	Sort	Search	Copy	Mean	DB Entry	DB Search	File Entry	Max.Ext.Arr.Rate
8-4-4-34-1-1-1-20-4-1	78	388	886	765	82	3369	3264	2807	146	764	27855	3104	3544	3060	2788	3369	3264	2807	2920	3056	27855	2230
8-4-4-35-1-1-1-20-4-1	79	388	886	765	82	3369	3264	2807	146	764	27855	3104	3544	3060	2870	3369	3264	2807	2920	3056	27855	2246
8-4-4-35-1-1-2-20-4-1	80	388	886	765	82	3369	3264	2807	146	764	27855	3104	3544	3060	2870	3369	3264	5614	2920	3056	27855	2296
8-4-4-36-1-1-2-20-4-1	81	388	886	765	82	3369	3264	2807	146	764	27855	3104	3544	3060	2952	3369	3264	5614	2920	3056	27855	2336
8-4-4-36-1-1-2-21-4-1	82	388	886	765	82	3369	3264	2807	146	764	27855	3104	3544	3060	2952	3369	3264	5614	3066	3056	27855	2362
8-4-4-37-1-1-2-21-4-1	83	388	886	765	82	3369	3264	2807	146	764	27855	3104	3544	3060	3034	3369	3264	5614	3066	3056	27855	2427
8-4-4-38-1-1-2-21-4-1	84	388	886	765	82	3369	3264	2807	146	764	27855	3104	3544	3060	3116	3369	3264	5614	3066	3056	27855	2445
8-4-4-38-1-1-2-21-5-1	85	388	886	765	82	3369	3264	2807	146	764	27855	3104	3544	3060	3116	3369	3264	5614	3066	3820	27855	2448
8-4-5-38-1-1-2-21-5-1	86	388	886	765	82	3369	3264	2807	146	764	27855	3104	3544	3825	3116	3369	3264	5614	3066	3820	27855	2453
8-4-5-38-1-1-2-22-5-1	87	388	886	765	82	3369	3264	2807	146	764	27855	3104	3544	3825	3116	3369	3264	5614	3212	3820	27855	2483
9-4-5-38-1-1-2-22-5-1	88	388	886	765	82	3369	3264	2807	146	764	27855	3492	3544	3825	3116	3369	3264	5614	3212	3820	27855	2493
9-4-5-39-1-1-2-22-5-1	89	388	886	765	82	3369	3264	2807	146	764	27855	3492	3544	3825	3198	3369	3264	5614	3212	3820	27855	2558
9-4-5-40-1-1-2-22-5-1	90	388	886	765	82	3369	3264	2807	146	764	27855	3492	3544	3825	3280	3369	3264	5614	3212	3820	27855	2570
9-4-5-40-1-1-2-23-5-1	91	388	886	765	82	3369	3264	2807	146	764	27855	3492	3544	3825	3280	3369	3264	5614	3358	3820	27855	2611
9-4-5-40-1-2-2-23-5-1	92	388	886	765	82	3369	3264	2807	146	764	27855	3492	3544	3825	3280	3369	6528	5614	3358	3820	27855	2624
9-4-5-41-1-2-2-23-5-1	93	388	886	765	82	3369	3264	2807	146	764	27855	3492	3544	3825	3362	3369	6528	5614	3358	3820	27855	2686
9-4-5-41-1-2-2-24-5-1	94	388	886	765	82	3369	3264	2807	146	764	27855	3492	3544	3825	3362	3369	6528	5614	3504	3820	27855	2690

Configuration	Threads	Random	String A	String B	Sort	Search	Copy	Mean	DB Entry	DB Search	File Entry	Max.Ext.Arr.Rate
9-4-5-42-1-2-2-24-5-1	95	388	886	765	82	3369	3264	2807	146	764	27855	2695
9-4-5-42-2-2-2-24-5-1	96	388	886	765	82	3369	3264	2807	146	764	27855	2755
9-4-5-43-2-2-2-24-5-1	97	388	886	765	82	3369	3264	2807	146	764	27855	2794
10-4-5-43-2-2-2-24-5-1	98	388	886	765	82	3369	3264	2807	146	764	27855	2803
10-4-5-43-2-2-2-25-5-1	99	388	886	765	82	3369	3264	2807	146	764	27855	2821
10-4-5-44-2-2-2-25-5-1	100	388	886	765	82	3369	3264	2807	146	764	27855	2835
10-5-5-44-2-2-2-25-5-1	101	388	886	765	82	3369	3264	2807	146	764	27855	2886
10-5-5-45-2-2-2-25-5-1	102	388	886	765	82	3369	3264	2807	146	764	27855	2920
10-5-5-45-2-2-2-26-5-1	103	388	886	765	82	3369	3264	2807	146	764	27855	2952
10-5-5-46-2-2-2-26-5-1	104	388	886	765	82	3369	3264	2807	146	764	27855	3018
10-5-5-47-2-2-2-26-5-1	105	388	886	765	82	3369	3264	2807	146	764	27855	3037
10-5-5-47-2-2-2-27-5-1	106	388	886	765	82	3369	3264	2807	146	764	27855	3056
10-5-5-47-2-2-2-27-6-1	107	388	886	765	82	3369	3264	2807	146	764	27855	3060
10-5-6-47-2-2-2-27-6-1	108	388	886	765	82	3369	3264	2807	146	764	27855	3083
10-5-6-48-2-2-2-27-6-1	109	388	886	765	82	3369	3264	2807	146	764	27855	3104
11-5-6-48-2-2-2-27-6-1	110	388	886	765	82	3369	3264	2807	146	764	27855	3149
11-5-6-49-2-2-2-27-6-1	111	388	886	765	82	3369	3264	2807	146	764	27855	3154

Configuration	Threads	Random	String A	String B	Sort	Search	Copy	Mean	DB Entry	DB Search	File Entry	Max.Ext.Arr.Rate
11-5-6-49-2-2-2-28-6-1	112	388	886	765	82	3369	3264	2807	146	764	27855	3214
11-5-6-50-2-2-2-28-6-1	113	388	886	765	82	3369	3264	2807	146	764	27855	3270
11-5-6-50-2-2-2-29-6-1	114	388	886	765	82	3369	3264	2807	146	764	27855	3280
11-5-6-51-2-2-2-29-6-1	115	388	886	765	82	3369	3264	2807	146	764	27855	3346
11-5-6-52-2-2-2-29-6-1	116	388	886	765	82	3369	3264	2807	146	764	27855	3387
11-5-6-52-2-2-2-30-6-1	117	388	886	765	82	3369	3264	2807	146	764	27855	3411
11-5-6-53-2-2-2-30-6-1	118	388	886	765	82	3369	3264	2807	146	764	27855	3414
12-5-6-53-2-2-2-30-6-1	119	388	886	765	82	3369	3264	2807	146	764	27855	3477
12-5-6-54-2-2-2-30-6-1	120	388	886	765	82	3369	3264	2807	146	764	27855	3504
12-5-6-54-2-2-2-31-6-1	121	388	886	765	82	3369	3264	2807	146	764	27855	3542
12-5-6-55-2-2-2-31-6-1	122	388	886	765	82	3369	3264	2807	146	764	27855	3544
12-6-6-55-2-2-2-31-6-1	123	388	886	765	82	3369	3264	2807	146	764	27855	3608
12-6-6-56-2-2-2-31-6-1	124	388	886	765	82	3369	3264	2807	146	764	27855	3621
12-6-6-56-2-2-2-32-6-1	125	388	886	765	82	3369	3264	2807	146	764	27855	3667
12-6-6-56-2-2-2-32-7-1	126	388	886	765	82	3369	3264	2807	146	764	27855	3672
12-6-7-56-2-2-2-32-7-1	127	388	886	765	82	3369	3264	2807	146	764	27855	3674
12-6-7-57-2-2-2-32-7-1	128	388	886	765	82	3369	3264	2807	146	764	27855	3725

Configuration	Random	String A	String B	Sort	Search	Copy	Mean	DB Entry	DB Search	File Entry
11-5-6-49-2-2-2-28-6-1	4268	4430	4590	4018	6738	6528	5614	4088	4584	27855
11-5-6-50-2-2-2-28-6-1	4268	4430	4590	4100	6738	6528	5614	4088	4584	27855
11-5-6-50-2-2-2-29-6-1	4268	4430	4590	4100	6738	6528	5614	4234	4584	27855
11-5-6-51-2-2-2-29-6-1	4268	4430	4590	4182	6738	6528	5614	4234	4584	27855
11-5-6-52-2-2-2-29-6-1	4268	4430	4590	4264	6738	6528	5614	4234	4584	27855
11-5-6-52-2-2-2-30-6-1	4268	4430	4590	4264	6738	6528	5614	4380	4584	27855
11-5-6-53-2-2-2-30-6-1	4268	4430	4590	4346	6738	6528	5614	4380	4584	27855
12-5-6-53-2-2-2-30-6-1	4656	4430	4590	4346	6738	6528	5614	4380	4584	27855
12-5-6-54-2-2-2-30-6-1	4656	4430	4590	4428	6738	6528	5614	4380	4584	27855
12-5-6-54-2-2-2-31-6-1	4656	4430	4590	4428	6738	6528	5614	4526	4584	27855
12-5-6-55-2-2-2-31-6-1	4656	4430	4590	4510	6738	6528	5614	4526	4584	27855
12-6-6-55-2-2-2-31-6-1	4656	5316	4590	4510	6738	6528	5614	4526	4584	27855
12-6-6-56-2-2-2-31-6-1	4656	5316	4590	4592	6738	6528	5614	4526	4584	27855
12-6-6-56-2-2-2-32-6-1	4656	5316	4590	4592	6738	6528	5614	4672	4584	27855
12-6-6-56-2-2-2-32-7-1	4656	5316	4590	4592	6738	6528	5614	4672	5348	27855
12-6-7-56-2-2-2-32-7-1	4656	5316	5355	4592	6738	6528	5614	4672	5348	27855
12-6-7-57-2-2-2-32-7-1	4656	5316	5355	4674	6738	6528	5614	4672	5348	27855

Configuration	Threads	Random	String A	String B	Sort	Search	Copy	Mean	DB Entry	DB Search	File Entry	Max.Ext.Arr.Rate
13-6-7-57-2-2-2-32-7-1	129	388	886	765	82	3369	3264	2807	146	764	27855	3738
13-6-7-57-2-2-2-33-7-1	130	388	886	765	82	3369	3264	2807	146	764	27855	3739
13-6-7-58-2-2-2-33-7-1	131	388	886	765	82	3369	3264	2807	146	764	27855	3805
13-6-7-59-2-2-2-33-7-1	132	388	886	765	82	3369	3264	2807	146	764	27855	3854
13-6-7-59-2-2-2-34-7-1	133	388	886	765	82	3369	3264	2807	146	764	27855	3870
13-6-7-60-2-2-2-34-7-1	134	388	886	765	82	3369	3264	2807	146	764	27855	3936
13-6-7-61-2-2-2-34-7-1	135	388	886	765	82	3369	3264	2807	146	764	27855	3971
13-6-7-61-2-2-2-35-7-1	136	388	886	765	82	3369	3264	2807	146	764	27855	4002
13-6-7-62-2-2-2-35-7-1	137	388	886	765	82	3369	3264	2807	146	764	27855	4035
14-6-7-62-2-2-2-35-7-1	138	388	886	765	82	3369	3264	2807	146	764	27855	4067
14-6-7-63-2-2-2-35-7-1	139	388	886	765	82	3369	3264	2807	146	764	27855	4088
14-6-7-63-2-2-2-36-7-1	140	388	886	765	82	3369	3264	2807	146	764	27855	4133
14-6-7-64-2-2-2-36-7-1	141	388	886	765	82	3369	3264	2807	146	764	27855	4198
14-6-7-65-2-2-2-36-7-1	142	388	886	765	82	3369	3264	2807	146	764	27855	4205
14-6-7-65-2-2-2-37-7-1	143	388	886	765	82	3369	3264	2807	146	764	27855	4253
14-7-7-65-2-2-2-37-7-1	144	388	886	765	82	3369	3264	2807	146	764	27855	4264
14-7-7-66-2-2-2-37-7-1	145	388	886	765	82	3369	3264	2807	146	764	27855	4278

Configuration	Random	String A	String B	Sort	Search	Copy	Mean	DB Entry	DB Search	File Entry
13-6-7-57-2-2-2-32-7-1	5044	5316	5355	4674	6738	6528	5614	4672	5348	27855
13-6-7-57-2-2-2-33-7-1	5044	5316	5355	4674	6738	6528	5614	4818	5348	27855
13-6-7-58-2-2-2-33-7-1	5044	5316	5355	4756	6738	6528	5614	4818	5348	27855
13-6-7-59-2-2-2-33-7-1	5044	5316	5355	4838	6738	6528	5614	4818	5348	27855
13-6-7-59-2-2-2-34-7-1	5044	5316	5355	4838	6738	6528	5614	4964	5348	27855
13-6-7-60-2-2-2-34-7-1	5044	5316	5355	4920	6738	6528	5614	4964	5348	27855
13-6-7-61-2-2-2-34-7-1	5044	5316	5355	5002	6738	6528	5614	4964	5348	27855
13-6-7-61-2-2-2-35-7-1	5044	5316	5355	5002	6738	6528	5614	5110	5348	27855
13-6-7-62-2-2-2-35-7-1	5044	5316	5355	5084	6738	6528	5614	5110	5348	27855
14-6-7-62-2-2-2-35-7-1	5432	5316	5355	5084	6738	6528	5614	5110	5348	27855
14-6-7-63-2-2-2-35-7-1	5432	5316	5355	5166	6738	6528	5614	5110	5348	27855
14-6-7-63-2-2-2-36-7-1	5432	5316	5355	5166	6738	6528	5614	5256	5348	27855
14-6-7-64-2-2-2-36-7-1	5432	5316	5355	5248	6738	6528	5614	5256	5348	27855
14-6-7-65-2-2-2-36-7-1	5432	5316	5355	5330	6738	6528	5614	5256	5348	27855
14-6-7-65-2-2-2-37-7-1	5432	5316	5355	5330	6738	6528	5614	5402	5348	27855
14-7-7-65-2-2-2-37-7-1	5432	6202	5355	5330	6738	6528	5614	5402	5348	27855
14-7-7-66-2-2-2-37-7-1	5432	6202	5355	5412	6738	6528	5614	5402	5348	27855

Configuration	Threads	Random	String A	String B	Sort	Search	Copy	Mean	DB Entry	DB Search	File Entry	Max.Ext.Arr.Rate
14-7-7-66-2-2-2-37-8-1	146	388	886	765	82	3369	3264	2807	146	764	27855	4284
14-7-8-66-2-2-2-37-8-1	147	388	886	765	82	3369	3264	2807	146	764	27855	4322
14-7-8-66-2-2-2-38-8-1	148	388	886	765	82	3369	3264	2807	146	764	27855	4330
14-7-8-67-2-2-2-38-8-1	149	388	886	765	82	3369	3264	2807	146	764	27855	4346
15-7-8-67-2-2-2-38-8-1	150	388	886	765	82	3369	3264	2807	146	764	27855	4395
15-7-8-68-2-2-2-38-8-1	151	388	886	765	82	3369	3264	2807	146	764	27855	4438
15-7-8-68-2-2-2-39-8-1	152	388	886	765	82	3369	3264	2807	146	764	27855	4461
15-7-8-69-2-2-2-39-8-1	153	388	886	765	82	3369	3264	2807	146	764	27855	4491
15-7-8-69-2-2-3-39-8-1	154	388	886	765	82	3369	3264	2807	146	764	27855	4526
15-7-8-70-2-2-3-39-8-1	155	388	886	765	82	3369	3264	2807	146	764	27855	4555
15-7-8-70-2-2-3-40-8-1	156	388	886	765	82	3369	3264	2807	146	764	27855	4592
15-7-8-71-2-2-3-40-8-1	157	388	886	765	82	3369	3264	2807	146	764	27855	4656
16-7-8-71-2-2-3-40-8-1	158	388	886	765	82	3369	3264	2807	146	764	27855	4658
16-7-8-72-2-2-3-40-8-1	159	388	886	765	82	3369	3264	2807	146	764	27855	4672
16-7-8-72-2-2-3-41-8-1	160	388	886	765	82	3369	3264	2807	146	764	27855	4723
16-7-8-73-2-2-3-41-8-1	161	388	886	765	82	3369	3264	2807	146	764	27855	4789
16-7-8-74-2-2-3-41-8-1	162	388	886	765	82	3369	3264	2807	146	764	27855	4789
Configuration	Threads	Random	String A	String B	Sort	Search	Copy	Mean	DB Entry	DB Search	File Entry	Max.Ext.Arr.Rate
		5432	6202	5355	5412	6738	6528	5614	5402	6112	27855	4284
		5432	6202	6120	5412	6738	6528	5614	5402	6112	27855	4322
		5432	6202	6120	5412	6738	6528	5614	5548	6112	27855	4330
		5432	6202	6120	5494	6738	6528	5614	5548	6112	27855	4346
		5820	6202	6120	5494	6738	6528	5614	5548	6112	27855	4395
		5820	6202	6120	5576	6738	6528	5614	5548	6112	27855	4438
		5820	6202	6120	5576	6738	6528	5614	5694	6112	27855	4461
		5820	6202	6120	5658	6738	6528	5614	5694	6112	27855	4491
		5820	6202	6120	5658	6738	6528	5614	5694	6112	27855	4526
		5820	6202	6120	5740	6738	6528	8421	5694	6112	27855	4555
		5820	6202	6120	5740	6738	6528	8421	5840	6112	27855	4592
		5820	6202	6120	5822	6738	6528	8421	5840	6112	27855	4656
		6208	6202	6120	5822	6738	6528	8421	5840	6112	27855	4658
		6208	6202	6120	5904	6738	6528	8421	5840	6112	27855	4672
		6208	6202	6120	5904	6738	6528	8421	5986	6112	27855	4723
		6208	6202	6120	5986	6738	6528	8421	5986	6112	27855	4789
		6208	6202	6120	6068	6738	6528	8421	5986	6112	27855	4789

Configuration	Threads	Random	String A	String B	Sort	Search	Copy	Mean	DB Entry	DB Search	File Entry	Random	String A	String B	Sort	Search	Copy	Mean	DB Entry	DB Search	File Entry	Max.Ext.Arr.Rate
16-7-8-74-2-2-3-42-8-1	163	388	886	765	82	3369	3264	2807	146	764	27855	6208	6202	6120	6068	6738	6528	8421	6132	6112	27855	4854
16-7-8-75-2-2-3-42-8-1	164	388	886	765	82	3369	3264	2807	146	764	27855	6208	6202	6120	6150	6738	6528	8421	6132	6112	27855	4890
16-7-8-75-2-2-3-42-9-1	165	388	886	765	82	3369	3264	2807	146	764	27855	6208	6202	6120	6150	6738	6528	8421	6132	6876	27855	4896
16-7-9-75-2-2-3-42-9-1	166	388	886	765	82	3369	3264	2807	146	764	27855	6208	6202	6885	6150	6738	6528	8421	6132	6876	27855	4906
16-7-9-75-2-2-3-43-9-1	167	388	886	765	82	3369	3264	2807	146	764	27855	6208	6202	6885	6150	6738	6528	8421	6278	6876	27855	4920
16-7-9-76-2-2-3-43-9-1	168	388	886	765	82	3369	3264	2807	146	764	27855	6208	6202	6885	6232	6738	6528	8421	6278	6876	27855	4962
16-8-9-76-2-2-3-43-9-1	169	388	886	765	82	3369	3264	2807	146	764	27855	6208	7088	6885	6232	6738	6528	8421	6278	6876	27855	4966
17-8-9-76-2-2-3-43-9-1	170	388	886	765	82	3369	3264	2807	146	764	27855	6596	7088	6885	6232	6738	6528	8421	6278	6876	27855	4986
17-8-9-77-2-2-3-43-9-1	171	388	886	765	82	3369	3264	2807	146	764	27855	6596	7088	6885	6314	6738	6528	8421	6278	6876	27855	5022
17-8-9-77-2-2-3-44-9-1	172	388	886	765	82	3369	3264	2807	146	764	27855	6596	7088	6885	6314	6738	6528	8421	6424	6876	27855	5051
17-8-9-78-2-2-3-44-9-1	173	388	886	765	82	3369	3264	2807	146	764	27855	6596	7088	6885	6396	6738	6528	8421	6424	6876	27855	5117
17-8-9-79-2-2-3-44-9-1	174	388	886	765	82	3369	3264	2807	146	764	27855	6596	7088	6885	6478	6738	6528	8421	6424	6876	27855	5139
17-8-9-79-2-2-3-45-9-1	175	388	886	765	82	3369	3264	2807	146	764	27855	6596	7088	6885	6478	6738	6528	8421	6570	6876	27855	5182
17-8-9-80-2-2-3-45-9-1	176	388	886	765	82	3369	3264	2807	146	764	27855	6596	7088	6885	6560	6738	6528	8421	6570	6876	27855	5222
17-8-9-80-2-3-3-45-9-1	177	388	886	765	82	3369	3264	2807	146	764	27855	6596	7088	6885	6560	6738	9792	8421	6570	6876	27855	5248
17-8-9-81-2-3-3-45-9-1	178	388	886	765	82	3369	3264	2807	146	764	27855	6596	7088	6885	6642	6738	9792	8421	6570	6876	27855	5256
17-8-9-81-2-3-3-46-9-1	179	388	886	765	82	3369	3264	2807	146	764	27855	6596	7088	6885	6642	6738	9792	8421	6716	6876	27855	5277

Configuration	Threads	Random	String A	String B	Sort	Search	Copy	Mean	DB Entry	DB Search	File Entry	Random	String A	String B	Sort	Search	Copy	Mean	DB Entry	DB Search	File Entry	Max.Ext.Arr.Rate
18-8-9-81-2-3-3-46-9-1	180	388	886	765	82	3369	3264	2807	146	764	27855	6984	7088	6885	6642	6738	9792	8421	6716	6876	27855	5314
18-8-9-82-2-3-3-46-9-1	181	388	886	765	82	3369	3264	2807	146	764	27855	6984	7088	6885	6724	6738	9792	8421	6716	6876	27855	5373
18-8-9-82-2-3-3-47-9-1	182	388	886	765	82	3369	3264	2807	146	764	27855	6984	7088	6885	6724	6738	9792	8421	6862	6876	27855	5379
18-8-9-83-2-3-3-47-9-1	183	388	886	765	82	3369	3264	2807	146	764	27855	6984	7088	6885	6806	6738	9792	8421	6862	6876	27855	5390
18-8-9-83-3-3-3-47-9-1	184	388	886	765	82	3369	3264	2807	146	764	27855	6984	7088	6885	6806	10107	9792	8421	6862	6876	27855	5445
18-8-9-84-3-3-3-47-9-1	185	388	886	765	82	3369	3264	2807	146	764	27855	6984	7088	6885	6888	10107	9792	8421	6862	6876	27855	5490
18-8-9-84-3-3-3-48-9-1	186	388	886	765	82	3369	3264	2807	146	764	27855	6984	7088	6885	6888	10107	9792	8421	7008	6876	27855	5501
18-8-9-84-3-3-3-48-10-1	187	388	886	765	82	3369	3264	2807	146	764	27855	6984	7088	6885	6888	10107	9792	8421	7008	7640	27855	5508
18-8-10-84-3-3-3-48-10-1	188	388	886	765	82	3369	3264	2807	146	764	27855	6984	7088	7650	6888	10107	9792	8421	7008	7640	27855	5510
18-8-10-85-3-3-3-48-10-1	189	388	886	765	82	3369	3264	2807	146	764	27855	6984	7088	7650	6970	10107	9792	8421	7008	7640	27855	5576
18-8-10-86-3-3-3-48-10-1	190	388	886	765	82	3369	3264	2807	146	764	27855	6984	7088	7650	7052	10107	9792	8421	7008	7640	27855	5587
19-8-10-86-3-3-3-48-10-1	191	388	886	765	82	3369	3264	2807	146	764	27855	7372	7088	7650	7052	10107	9792	8421	7008	7640	27855	5606
19-8-10-86-3-3-3-49-10-1	192	388	886	765	82	3369	3264	2807	146	764	27855	7372	7088	7650	7052	10107	9792	8421	7154	7640	27855	5642
19-8-10-87-3-3-3-49-10-1	193	388	886	765	82	3369	3264	2807	146	764	27855	7372	7088	7650	7134	10107	9792	8421	7154	7640	27855	5670
19-9-10-87-3-3-3-49-10-1	194	388	886	765	82	3369	3264	2807	146	764	27855	7372	7974	7650	7134	10107	9792	8421	7154	7640	27855	5707
19-9-10-88-3-3-3-49-10-1	195	388	886	765	82	3369	3264	2807	146	764	27855	7372	7974	7650	7216	10107	9792	8421	7154	7640	27855	5723
19-9-10-88-3-3-3-50-10-1	196	388	886	765	82	3369	3264	2807	146	764	27855	7372	7974	7650	7216	10107	9792	8421	7300	7640	27855	5773

Configuration	Threads	Random	String A	String B	Sort	Search	Copy	Mean	DB Entry	DB Search	File Entry	Random	String A	String B	Sort	Search	Copy	Mean	DB Entry	DB Search	File Entry	Max.Ext.Arr.Rate
19-9-10-89-3-3-3-50-10-1	197	388	886	765	82	3369	3264	2807	146	764	27855	7372	7974	7650	7298	10107	9792	8421	7300	7640	27855	5838
19-9-10-90-3-3-3-50-10-1	198	388	886	765	82	3369	3264	2807	146	764	27855	7372	7974	7650	7380	10107	9792	8421	7300	7640	27855	5840
19-9-10-90-3-3-3-51-10-1	199	388	886	765	82	3369	3264	2807	146	764	27855	7372	7974	7650	7380	10107	9792	8421	7446	7640	27855	5898
20-9-10-90-3-3-3-51-10-1	200	388	886	765	82	3369	3264	2807	146	764	27855	7760	7974	7650	7380	10107	9792	8421	7446	7640	27855	5904
20-9-10-91-3-3-3-51-10-1	201	388	886	765	82	3369	3264	2807	146	764	27855	7760	7974	7650	7462	10107	9792	8421	7446	7640	27855	5957
20-9-10-91-3-3-3-52-10-1	202	388	886	765	82	3369	3264	2807	146	764	27855	7760	7974	7650	7462	10107	9792	8421	7592	7640	27855	5970
20-9-10-92-3-3-3-52-10-1	203	388	886	765	82	3369	3264	2807	146	764	27855	7760	7974	7650	7544	10107	9792	8421	7592	7640	27855	6035
20-9-10-93-3-3-3-52-10-1	204	388	886	765	82	3369	3264	2807	146	764	27855	7760	7974	7650	7626	10107	9792	8421	7592	7640	27855	6074
20-9-10-93-3-3-3-53-10-1	205	388	886	765	82	3369	3264	2807	146	764	27855	7760	7974	7650	7626	10107	9792	8421	7738	7640	27855	6101
20-9-10-94-3-3-3-53-10-1	206	388	886	765	82	3369	3264	2807	146	764	27855	7760	7974	7650	7708	10107	9792	8421	7738	7640	27855	6112
20-9-10-94-3-3-3-53-11-1	207	388	886	765	82	3369	3264	2807	146	764	27855	7760	7974	7650	7708	10107	9792	8421	7738	8404	27855	6120
20-9-11-94-3-3-3-53-11-1	208	388	886	765	82	3369	3264	2807	146	764	27855	7760	7974	8415	7708	10107	9792	8421	7738	8404	27855	6166
20-9-11-95-3-3-3-53-11-1	209	388	886	765	82	3369	3264	2807	146	764	27855	7760	7974	8415	7790	10107	9792	8421	7738	8404	27855	6190
20-9-11-95-3-3-3-54-11-1	210	388	886	765	82	3369	3264	2807	146	764	27855	7760	7974	8415	7790	10107	9792	8421	7884	8404	27855	6208
21-9-11-95-3-3-3-54-11-1	211	388	886	765	82	3369	3264	2807	146	764	27855	8148	7974	8415	7790	10107	9792	8421	7884	8404	27855	6232
21-9-11-96-3-3-3-54-11-1	212	388	886	765	82	3369	3264	2807	146	764	27855	8148	7974	8415	7872	10107	9792	8421	7884	8404	27855	6298
21-9-11-97-3-3-3-54-11-1	213	388	886	765	82	3369	3264	2807	146	764	27855	8148	7974	8415	7954	10107	9792	8421	7884	8404	27855	6307

Configuration	Threads	Random	String A	String B	Sort	Search	Copy	Mean	DB Entry	DB Search	File Entry	Max.Ext.Arr.Rate
21-9-11-97-3-3-3-55-11-1	214	388	886	765	82	3369	3264	2807	146	764	-”	6363
21-9-11-98-3-3-3-55-11-1	215	388	886	765	82	3369	3264	2807	146	764	-”	6379
21-10-11-98-3-3-3-55-11-1	216	388	886	765	82	3369	3264	2807	146	764	-”	6424
21-10-11-98-3-3-3-56-11-1	217	388	886	765	82	3369	3264	2807	146	764	-”	6429
21-10-11-99-3-3-3-56-11-1	218	388	886	765	82	3369	3264	2807	146	764	-”	6494
21-10-11-100-3-3-3-56-11-1	219	388	886	765	82	3369	3264	2807	146	764	-”	6518
22-10-11-100-3-3-3-56-11-1	220	388	886	765	82	3369	3264	2807	146	764	-”	6541
22-10-11-100-3-3-3-57-11-1	221	388	886	765	82	3369	3264	2807	146	764	-”	6560
22-10-11-101-3-3-3-57-11-1	222	388	886	765	82	3369	3264	2807	146	764	-”	6626
22-10-11-102-3-3-3-57-11-1	223	388	886	765	82	3369	3264	2807	146	764	-”	6658
22-10-11-102-3-3-3-58-11-1	224	388	886	765	82	3369	3264	2807	146	764	-”	6691
22-10-11-103-3-3-3-58-11-1	225	388	886	765	82	3369	3264	2807	146	764	-”	6723
22-10-11-103-3-3-3-58-12-1	226	388	886	765	82	3369	3264	2807	146	764	-”	6732
22-10-12-103-3-3-3-58-12-1	227	388	886	765	82	3369	3264	2807	146	764	-”	6737
22-10-12-103-3-3-4-58-12-1	228	388	886	765	82	3369	3264	2807	146	764	-”	6757
22-10-12-104-3-3-4-58-12-1	229	388	886	765	82	3369	3264	2807	146	764	-”	6774
22-10-12-104-3-3-4-59-12-1	230	388	886	765	82	3369	3264	2807	146	764	-”	6822

Configuration	Random	String A	String B	Sort	Search	Copy	Mean	DB Entry	DB Search	File Entry	Max.Ext.Arr.Rate
21-9-11-97-3-3-3-55-11-1	8148	7974	8415	7954	10107	9792	8421	8030	8404	27855	6363
21-9-11-98-3-3-3-55-11-1	8148	7974	8415	8036	10107	9792	8421	8030	8404	27855	6379
21-10-11-98-3-3-3-55-11-1	8148	8860	8415	8036	10107	9792	8421	8030	8404	27855	6424
21-10-11-98-3-3-3-56-11-1	8148	8860	8415	8036	10107	9792	8421	8176	8404	27855	6429
21-10-11-99-3-3-3-56-11-1	8148	8860	8415	8118	10107	9792	8421	8176	8404	27855	6494
21-10-11-100-3-3-3-56-11-1	8148	8860	8415	8200	10107	9792	8421	8176	8404	27855	6518
22-10-11-100-3-3-3-56-11-1	8536	8860	8415	8200	10107	9792	8421	8176	8404	27855	6541
22-10-11-100-3-3-3-57-11-1	8536	8860	8415	8200	10107	9792	8421	8322	8404	27855	6560
22-10-11-101-3-3-3-57-11-1	8536	8860	8415	8282	10107	9792	8421	8322	8404	27855	6626
22-10-11-102-3-3-3-57-11-1	8536	8860	8415	8364	10107	9792	8421	8322	8404	27855	6658
22-10-11-102-3-3-3-58-11-1	8536	8860	8415	8364	10107	9792	8421	8468	8404	27855	6691
22-10-11-103-3-3-3-58-11-1	8536	8860	8415	8446	10107	9792	8421	8468	8404	27855	6723
22-10-11-103-3-3-3-58-12-1	8536	8860	8415	8446	10107	9792	8421	8468	9168	27855	6732
22-10-12-103-3-3-3-58-12-1	8536	8860	9180	8446	10107	9792	8421	8468	9168	27855	6737
22-10-12-103-3-3-4-58-12-1	8536	8860	9180	8446	10107	9792	11228	8468	9168	27855	6757
22-10-12-104-3-3-4-58-12-1	8536	8860	9180	8528	10107	9792	11228	8468	9168	27855	6774
22-10-12-104-3-3-4-59-12-1	8536	8860	9180	8528	10107	9792	11228	8614	9168	27855	6822

Configuration	Threads	Random	String A	String B	Sort	Search	Copy	Mean	DB Entry	DB Search	File Entry	Max.Ext.Arr.Rate
22-10-12-105-3-3-4-59-12-1	231	388	886	765	82	3369	3264	2807	146	764	-”-	6829
23-10-12-105-3-3-4-59-12-1	232	388	886	765	82	3369	3264	2807	146	764	-”-	6888
23-10-12-106-3-3-4-59-12-1	233	388	886	765	82	3369	3264	2807	146	764	-”-	6891
23-10-12-106-3-3-4-60-12-1	234	388	886	765	82	3369	3264	2807	146	764	-”-	6954
23-10-12-107-3-3-4-60-12-1	235	388	886	765	82	3369	3264	2807	146	764	-”-	7008
23-10-12-107-3-3-4-61-12-1	236	388	886	765	82	3369	3264	2807	146	764	-”-	7019
23-10-12-108-3-3-4-61-12-1	237	388	886	765	82	3369	3264	2807	146	764	-”-	7085
23-10-12-109-3-3-4-61-12-1	238	388	886	765	82	3369	3264	2807	146	764	-”-	7088
23-11-12-109-3-3-4-61-12-1	239	388	886	765	82	3369	3264	2807	146	764	-”-	7125
23-11-12-109-3-3-4-62-12-1	240	388	886	765	82	3369	3264	2807	146	764	-”-	7139
24-11-12-109-3-3-4-62-12-1	241	388	886	765	82	3369	3264	2807	146	764	-”-	7150
24-11-12-110-3-3-4-62-12-1	242	388	886	765	82	3369	3264	2807	146	764	-”-	7216
24-11-12-111-3-3-4-62-12-1	243	388	886	765	82	3369	3264	2807	146	764	-”-	7242
24-11-12-111-3-3-4-63-12-1	244	388	886	765	82	3369	3264	2807	146	764	-”-	7282
24-11-12-112-3-3-4-63-12-1	245	388	886	765	82	3369	3264	2807	146	764	-”-	7334
24-11-12-112-3-3-4-63-13-1	246	388	886	765	82	3369	3264	2807	146	764	-”-	7344
24-11-13-112-3-3-4-63-13-1	247	388	886	765	82	3369	3264	2807	146	764	-”-	7347

Configuration	Threads	Random	String A	String B	Sort	Search	Copy	Mean	DB Entry	DB Search	File Entry	Max.Ext.Arr.Rate
24-11-13-113-3-3-4-63-13-1	248	388	886	765	82	3369	3264	2807	146	764	-	7358
24-11-13-113-3-3-4-64-13-1	249	388	886	765	82	3369	3264	2807	146	764	-	7413
24-11-13-114-3-3-4-64-13-1	250	388	886	765	82	3369	3264	2807	146	764	-	7450
25-11-13-114-3-3-4-64-13-1	251	388	886	765	82	3369	3264	2807	146	764	-	7475
25-11-13-114-3-3-4-65-13-1	252	388	886	765	82	3369	3264	2807	146	764	-	7478
25-11-13-115-3-3-4-65-13-1	253	388	886	765	82	3369	3264	2807	146	764	-	7544
25-11-13-116-3-3-4-65-13-1	254	388	886	765	82	3369	3264	2807	146	764	-	7592
25-11-13-116-3-3-4-66-13-1	255	388	886	765	82	3369	3264	2807	146	764	-	7610
25-11-13-117-3-3-4-66-13-1	256	388	886	765	82	3369	3264	2807	146	764	-	7676
		9312	9746	9945	9266	10107	9792	11228	9198	9932	27855	
		9312	9746	9945	9266	10107	9792	11228	9344	9932	27855	
		9312	9746	9945	9348	10107	9792	11228	9344	9932	27855	
		9700	9746	9945	9348	10107	9792	11228	9344	9932	27855	
		9700	9746	9945	9348	10107	9792	11228	9490	9932	27855	
		9700	9746	9945	9430	10107	9792	11228	9490	9932	27855	
		9700	9746	9945	9512	10107	9792	11228	9490	9932	27855	
		9700	9746	9945	9512	10107	9792	11228	9636	9932	27855	
		9700	9746	9945	9594	10107	9792	11228	9636	9932	27855	

List of Figures

List of Tables

Listings

Bibliography

[ABR04] T. Abbes, A. Bouhoula, and M. Rusinowitch. Protocol Analysis in Intrusion Detection Using Decision Trees. In *International Conference on Information Technology: Coding and Computing (ITCC'04)*, pages 404–408, 2004.

[Agr02] Alan Agresti. *Categorical Data Analysis*. Wiley-Interscience, 2nd edition, July 202.

[APWW07] B.K. Aichernig, B. Peischl, M. Weiglhofer, and F. Wotawa. Protocol Conformance Testing a SIP Registrar: an Industrial Application of Formal Methods. In *5th IEEE Int. Conference on Software Engineering and Formal Methods*, pages 215–224, 2007.

[ASF07] H.J. Abdelnur, R. State, and O. Festor. KiF: A Stateful SIP Fuzzer. In *1st Int. Conference on Principles, Systems and Applications of IP Telecommunications*. iptcomm.org, 2007.

[Aut09] Internet Assigned Numbers Authority. Session initiation protocol (sip) parameters. *http://www.iana.org/assignments/sip-parameters*, 2009.

[AWW+07a] A. Acharya, X. Wand, C. Wrigth, N. Banerjee, and B. Sengupta. Real-time Monitoring of SIP Infrastructure Using Message Classification. In *MineNet'07*, pages 45–50, 2007.

[AWW07b] Arup Acharya, Xiping Wang, and Charles Wright. A programmable message classification engine for session initiation protocol (sip). In *Proceedings of the 3rd ACM/IEEE Symposium on Architecture for networking and communications systems*, ANCS '07, pages 185–194, New York, NY, USA, 2007. ACM.

[BCDW04] Jeremy S. Bradbury, James R. Cordy, Juergen Dingel, and Michel Wermelinger. A survey of self-management in dynamic software architecture specifications. In *WOSS'04: Proceedings of the 1st ACM SIGSOFT workshop on Self-managed systems*, pages 28–33, New York, NY, USA, 2004. ACM.

[BFL+06] Andreas Bonelli, Franz Franchetti, Juergen Lorenz, Markus Püschel, and Christoph W. Ueberhuber. Automatic performance optimization of the discrete Fourier transform on distributed memory computers. In *International Symposium on Parallel and Distributed Processing and Application (ISPA)*, volume 4330 of *Lecture Notes In Computer Science*, pages 818–832. Springer, 2006.

[BGdMT06] Gunter Bolch, Stefan Greiner, Hermann de Meer, and Kishor S. Trivedi. *Queueing Networks and Markov Chains: Modeling and Performance Evaluation with Computer Science Applications*. WileyBlackwell, 2nd edition, May 2006. `http://www4.informatik.uni-erlangen.de/QNMC`.

[BGMT05] Gunter Bolch, Stefan Greiner, Hermann de Meer, and Kishor Shridharbhai Trivedi. *Queueing Networks and Markov Chains*. Wiley-Interscience, 2005.

[BPA+08] Josep L. Berral, Nicolas Poggi, Javier Alonso, Ricard Gavaldà, Jordi Torres, and Manish Parashar. Adaptive distributed mechanism against flooding network attacks based on machine learning. In *AISec '08: Proceedings of the 1st ACM workshop on Workshop on AISec*, pages 43–50, New York, NY, USA, 2008. ACM.

[BPI03] Simonetta Balsamo, Vittoria De Nitto Person, and Paola Inverardi. A review on queueing network models with finite capacity queues for software architectures performance prediction. *Performance Evaluation*, 51(2-4):269 – 288, 2003.

[CCK+11] J.J. Cochran, L.A. Cox, P. Keskinocak, J.P. Kharoufeh, and J.C. Smith. *Wiley Encyclopedia of Operations Research and Management Science, 8 Volume Set*. Wiley Encyclopedia of Operations Research and Management Science. John Wiley & Sons, 2011.

[CLG+03] P. Calhoun, J. Loughney, E. Guttman, G. Zorn, and J. Arkko. Diameter base protocol, 2003.

[Cod06] Codenomicon. Codenomicon sip test tool. *http://www.codenomicon.com/*, 2006.

[DDKM08] Dylan Dawson, Ron Desmarais, Holger M. Kienle, and Hausi A. Müller. Monitoring in adaptive systems using reflection. In *Proceedings of the 2008 international workshop on Software engineering for adaptive and self-managing systems*, SEAMS '08, pages 81–88, New York, NY, USA, 2008. ACM.

[DMSFR10] Giovanna Di Marzo Serugendo, John Fitzgerald, and Alexander Romanovsky. Metaself: an architecture and a development method for dependable self-* systems. In *Proceedings of the 2010 ACM Symposium on Applied Computing*, SAC '10, pages 457–461, New York, NY, USA, 2010. ACM.

[FNKC07] Ali Fessi, Heiko Niedermayer, Holger Kinkelin, and Georg Carle. A cooperative sip infrastructure for highly reliable telecommunication services. In *Proceedings of the 1st international conference on Principles, systems and applications of IP telecommunications*, IPTComm '07, pages 29–38, New York, NY, USA, 2007. ACM.

[Fri04] Matteo Frigo. A fast fourier transform compiler. *SIGPLAN Not.*, 39(4):642–655, 2004.

[FSJ05] Matteo Frigo, Steven, and G. Johnson. The design and implementation of fftw3. In *Proceedings of the IEEE*, volume 93, pages 216–231, 2005.

[GVAGM08] Charles Gouin-Vallerand, Bessam Abdulrazak, Sylvain Giroux, and Mounir Mokhtari. Toward autonomic pervasive computing. In *iiWAS '08: Proceedings of the 10th International Conference on Information Integration and Web-based Applications & Services*, pages 673–676, New York, NY, USA, 2008. ACM.

[HAAM08] Hlavacs Helmut, Hummel Karin Anna, Hess Andrea, and Nussbaumer Michael. Babel-sip: Self-learning sip message adaptation for increasing sip-compatibility. In *1st IEEE Workshop on Automated Network Management*

(ANM'08), in conjunction with the IEEE INFOCOM 2008, Phoenix, Arizona, 4 2008.

[HCL06] HCLT. Hclt. *http://www.hcltech.com/*, 2006.

[HdM08] Richard Holzer and Hermann de Meer. On modeling of self-organizing systems. In *Autonomics '08: Proceedings of the 2nd International Conference on Autonomic Computing and Communication Systems*, pages 1–6, ICST, Brussels, Belgium, Belgium, 2008. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering).

[HGB10] Regina Hebig, Holger Giese, and Basil Becker. Making control loops explicit when architecting self-adaptive systems. In *Proceeding of the second international workshop on Self-organizing architectures*, SOAR '10, pages 21–28, New York, NY, USA, 2010. ACM.

[HJP06] M. Handley, V. Jacobson, and C. Perkins. Sdp: Session description protocol. RFC 4566, July 2006.

[HL08] Ruan He and Marc Lacoste. Applying component-based design to self-protection of ubiquitous systems. In *SEPS '08: Proceedings of the 3rd ACM workshop on Software engineering for pervasive services*, pages 9–14, New York, NY, USA, 2008. ACM.

[HM04] S. Heisig and S. Moyle. Using Model Trees to Characterize Computer Resource Usage. In *1st ACM SIGSOFT Workshop on Self-Managed Systems*, pages 80–84, 2004.

[HM08] Markus C. Huebscher and Julie A. McCann. A survey of autonomic computing—degrees, models, and applications. *ACM Comput. Surv.*, 40(3):1–28, 2008.

[HNHH08] Andrea Hess, Michael Nussbaumer, Helmut Hlavacs, and Karin Anna Hummel. Automatic adaptation and analysis of sip headers using decision trees. pages 69–89, 2008.

[HT03] Zsuzsanna Harangozó and Katalin Tarnay. Fdts in self-adaptive protocol specification. In *Proceedings of the 2nd international conference on Self-adaptive software: applications*, IWSAS'01, pages 113–128, Berlin, Heidelberg, 2003. Springer-Verlag.

[Jai91] R. K. Jain. *The Art of Computer Systems Performance Analysis: Techniques for Experimental Design, Measurement, Simulation, and Modeling.* Wiley, April 1991. `http://www.cse.wustl.edu/~jain/books/perfbook.htm`.

[KASH05] Nagarajan Kandasamy, Sherif Abdelwahed, Gregory C. Sharp, and John P. Hayes. An Online Control Framework for Designing Self-Optimizing Computing Systems: Application to Power Management. In *Self-star Properties in Complex Information Systems*, pages 174–188. Springer Berlin, 2005.

[KC03] Jeffrey O. Kephart and David M. Chess. The vision of autonomic computing. *Computer*, 36(1):41–50, 2003.

[KKSJ10] Narges Khakpour, Ramtin Khosravi, Marjan Sirjani, and Saeed Jalili. Formal analysis of policy-based self-adaptive systems. In *Proceedings of the 2010 ACM Symposium on Applied Computing*, SAC '10, pages 2536–2543, New York, NY, USA, 2010. ACM.

[KM07] Jeff Kramer and Jeff Magee. Self-managed systems: an architectural challenge. In *FOSE '07: 2007 Future of Software Engineering*, pages 259–268, Washington, DC, USA, 2007. IEEE Computer Society.

[KM08] Kobayashi and Mark. *System Modeling And Analysis - Foundations of System Performance Evaluation*, volume 1. Prentice Hall, 1st edition, 2008. `http://www.princeton.edu/kobayashi/Book/book.html`.

[KP11] Thomas Karcher and Victor Pankratius. Auto-tuning multicore applications at run-time with a cooperative tuner. publikation, Feb 2011.

[KRG+10] Elsy Kaddoum, Claudia Raibulet, Jean-Pierre Georgé, Gauthier Picard, and Marie-Pierre Gleizes. Criteria for the evaluation of self-* systems. In *Proceedings of the 2010 ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems*, SEAMS '10, pages 29–38, New York, NY, USA, 2010. ACM.

[KZRN07] H.J. Kang, Z.L. Zhang, S. Ranjan, and A. Nucci. SIP-based VoIP Traffic Behavior Profiling and Its Applications. In *MineNet'07*, pages 39–44, 2007.

[Lim09] Skype Limited. Skype. *http://www.skype.com/intl/en/s*, 2009.

[MM10] Moreno Marzolla and Raffaela Mirandola. Performance aware reconfiguration of software systems. In *Proceedings of the 7th European performance engineering conference on Computer performance engineering*, EPEW'10, pages 51–66, Berlin, Heidelberg, 2010. Springer-Verlag.

[NH11] Michael Nussbaumer and Helmut Hlavacs. Optimization for multi-thread dataflow software. In *Proceedings of the 2011 8th European Performance Engineering Workshop*, EPEW '11, 2011.

[NRS10] Russel Nzekwa, Romain Rouvoy, and Lionel Seinturier. Modelling feedback control loops for self-adaptive systems. *ECEASST*, 28, 2010.

[OK07] Takayuki Osogami and Sei Kato. Optimizing system configurations quickly by guessing at the performance. *SIGMETRICS Perform. Eval. Rev.*, 35(1):145–156, 2007.

[Pin08] Martin Pinzger. Automated web performance analysis, with a special focus on prediction. In *iiWAS '08: Proceedings of the 10th International Conference on Information Integration and Web-based Applications & Services*, pages 539–542, New York, NY, USA, 2008. ACM.

[PMJ+05] Markus Püschel, José M. F. Moura, Jeremy Johnson, David Padua, Manuela Veloso, Bryan Singer, Jianxin Xiong, Franz Franchetti, Aca Gacic, Yevgen Voronenko, Kang Chen, Robert W. Johnson, and Nicholas Rizzolo. SPIRAL: Code generation for DSP transforms. *Proceedings of the IEEE, special issue on "Program Generation, Optimization, and Adaptation"*, 93(2):232– 275, 2005.

[Rad06] Radvision. Prolab sip test solution. *http://www.radvision.com/Products/Developer/Testing-and-Analysis-Tools/ProLab-SIP/sip.htm*, 2006.

[RN09] Z. Rusinovic and Bogunovic N. Self-healing model for sip-based services. ieee, 2009.

[RS03] J. Rosenberg and H. Schulzrinne. An extension to the session initiation protocol (sip) for symmetric response routing, 2003.

[RSC+02] J. Rosenberg, H. Schulzrinne, G. Camarillo, A. Johnston, J. Peterson, R. Sparks, M. Handley, and E. Schooler. SIP: Session Initiation Protocol. RFC 3261, June 2002.

[SCFJ03] H. Schulzrinne, S. Casner, R. Frederick, and V. Jacobson. Rtp: A transport protocol for real-time applications. RFC 3550, July 2003.

[SILM07] Bogdan Solomon, Dan Ionescu, Marin Litoiu, and Mircea Mihaescu. A real-time adaptive control of autonomic computing environments. In *Proceedings of the 2007 conference of the center for advanced studies on Collaborative research*, CASCON '07, pages 124–136, New York, NY, USA, 2007. ACM.

[SIP02] SIPsak. Sip swiss army knife. *http://sipsak.org/*, 2002.

[Sip06a] Sipient. Sipflow standard. *http://www.sipient.com/standard.html*, 2006.

[SIP06b] SIPp. Sipp. *http://sipp.sourceforge.net/*, 2006.

[Spi06] Spirent. Spirent protocol tester. *http://www.aspid.pt/files/PDF/spt.pdf*, 2006.

[ST09] Mazeiar Salehie and Ladan Tahvildari. Self-adaptive software: Landscape and research challenges. *ACM Trans. Auton. Adapt. Syst.*, 4:14:1–14:42, May 2009.

[Sub09] Venkita Subramonian. Towards automated functional testing of converged applications. In *Proceedings of the 3rd International Conference on Principles, Systems and Applications of IP Telecommunications*, IPTComm '09, pages 9:1–9:12, New York, NY, USA, 2009. ACM.

[TDB08] Stanimire Tomov, Jack Dongarra, and Marc Baboulin. Towards dense linear algebra for hybrid gpu accelerated manycore systems, 2008.

[TI09] Tech-Invite. Sip service examples. *http://www.tech-invite.com/Ti-sip-service-1.html*, 2009.

[VI09a] VoIP-Info.org. A reference guide to all things voip. *http://www.voip-info.org/*, 2009.

[VI09b] VoIP-Info.org. Voip phones. *http://www.voip-info.org/wiki/view/VOIP+Phones*, 2009.

[VR07] Peter Van Roy. Self Management and the Future of Software Design. In *Proceedings of the Third International Workshop on Formal Aspects of Component Software (FACS 2006)*, volume 182, pages 201–217, 06 2007.

[WD98] R. Clint Whaley and Jack J. Dongarra. Automatically tuned linear algebra software. In *Supercomputing '98: Proceedings of the 1998 ACM/IEEE conference on Supercomputing (CDROM)*, pages 1–27, Washington, DC, USA, 1998. IEEE Computer Society.

[WF05] Ian H. Witten and Eibe Frank. *Data mining : practical machine learning tools and techniques*. Elsevier, Morgan Kaufman, Amsterdam [u.a.], 2. ed. edition, 2005.

[Wik09a] Wikipedia. List of sip response codes. *http://en.wikipedia.org/wiki/SIP_Responses*, 2009.

[Wik09b] Wikipedia. Mobile phone. *http://en.wikipedia.org/wiki/Mobile_phone*, 2009.

[Wik10a] Wikipedia. Fuzz testing. *http://en.wikipedia.org/wiki/Fuzz_testing*, 2010.

[Wik10b] Wikipedia. Kendall's notation. *http://en.wikipedia.org/wiki/Kendall's_notation*, 2010.

[WNH10] Roman Weidlich, Michael Nussbaumer, and Helmut Hlavacs. Optimization towards consolidation or throughput for multi-thread software. In *Proceedings of the 2010 3rd International Symposium on Parallel Architectures, Algorithms and Programming*, PAAP '10, pages 161–168, Washington, DC, USA, 2010. IEEE Computer Society.

[WR05] D. Wilking and T. Röfer. Realtime Object Recognition Using Decision Tree Learning. In *RoboCup 2004: Robot World Cup VII*, pages 556–563. Springer, 2005.

[WSF10] Jon Whittle, Will Simm, and Maria-Angela Ferrario. On the role of the user in monitoring the environment in self-adaptive systems: a position paper. In *Proceedings of the 2010 ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems*, SEAMS '10, pages 69–74, New York, NY, USA, 2010. ACM.

[Xu08] Jing Xu. Rule-based automatic software performance diagnosis and improvement. In *WOSP '08: Proceedings of the 7th international workshop on Software and performance*, pages 1–12, New York, NY, USA, 2008. ACM.

[YPS05] Kamen Yotov, Keshav Pingali, and Paul Stodghill. X-ray: A tool for automatic measurement of hardware parameters. *Quantitative Evaluation of Systems, International Conference on*, 0:168–178, 2005.

[Zuk09] Moshe Zukerman. *Introduction to Queueing Theory and Stochastic Teletraffic Models*. Zukerman, 2009. `http://www.ee.cityu.edu.hk/~zukerman/classnotes.pdf`.

Abstrakt

Meine Dissertation beschaeftigt sich mit autonomen, selbst-lernenden und selbst-adaptiven Systemen. Prinzipiell muss ein autonomes und selbst-lernendes System seinen eigenen Status, sowie die externen Operationen kennen, muss Systemveraenderungen erkennen koennen und muss in der Lage sein sich selbst zu adaptieren.

Verbesserung der Zuverlaessigkeit von Multimedia Kommunikation:
Im Zuge des Testens eines kommerziellen VoIP Servers wurde deutlich, dass das SIP Protokoll, welches fuer die Initiierung von VoIP Telefonaten verwendet wird, in einem sehr offenen Standard definiert ist.
Fuer eine korrekte SIP Nachricht sind nur einige wenige Informationen notwendig. Es gibt allerdings eine enorme Anzahl an optionalen Informationen, die ebenfalls innerhalb einer SIP Nachricht verwendet werden koennen. Diese Tatsache fuehrt dazu, dass VoIP Geraete eine enorme Anzahl an unterschiedlichen SIP-Dialekten verwenden, die aus der riesigen Anzahl an unterschiedlichen Parameterkombinationen entstehen. Dies kann zu dem Problem fuehren, dass Telefone die dasselbe Protokoll verwenden, trotzdem nicht in der Lage sind miteinander zu kommunizieren.
Deshalb wird ein autonomes, selbst-lernendes SIP-Uebersetzungstool praesentiert, welches die Rate der faelschlich vom Server abgewiesenen SIP Nachrichten drastisch reduziert, indem ankommende Nachrichten analysiert und eventuell veraendert werden.

Autonome Adaption von Systemparametern, um die Systemperformance zu verbessern:
Die Performance eines kommerziellen Systems, welches Daten von unterschiedlichen mobilen Geraeten sammelt und verarbeitet, ist aufgrund des hohen ankommenden Datenaufkommens extrem wichtig.
Ankommende Datentickets wandern durch ein Warteschlangensystem, wo in jedem durchlaufenen Knoten unterschiedliche atomare Aktionen durchgefuehrt werden. Dieser Aufbau ermoeglicht es, die einzelnen Knoten zu parallelisieren, in dem mehrere Auspraegungen der Knoten auf unterschiedlichen CPU-Kernen gestartet werden.
Mit Hilfe eines Systems, welches analytische Ansaetze, Messungen und Simulationen verwendet, wird die optimale Softwarekonfiguration fuer eine bestimmte Hardware automatisiert gefunden. Dadurch passt sich die Software immer exakt an die aktuelle Hardware und an das aktuelle Datenaufkommen an. Die Performance des Gesamtsystems kann so drastisch verbessert werden.

Lebenslauf

Mag. Michael Nussbaumer, Bakk.
Obkirchergasse 16/8/12, 1190 Wien
Tel.: +43 676 911 80 80
E-Mail: michael.nussbaumer@chello.at

Ausbildung:

September 1996 - Juni 2001
Hoehere Technische Lehranstalt - Schulzentrum Ungargasse, Wien
Spezialisierung: Wirtschaftsingenieurwesen mit Schwerpunkt: Betriebsinformatik

Oktober 2001 - Maerz 2006
Universitaet Wien
Studienrichtungen: Wirtschaftsinformatik
Magisterarbeit: Performance ausgesuchter Methoden zur Spamerkennung

Juni 2006 - Dezember 2010
Universitaet Wien - Institut fuer verteilte und multimediale Systeme
Wissenschaftlicher Mitarbeiter im Projekt SoftNet Austria in Kooperation mit Kapsch CarrierCom
Themen: Formal Methods in Software Engineering of Mobile Applications - Zuverlaessigkeit und Performance von Telekommunikationssystemen

Jaenner 2011 - August 2011
Universitaet Wien - Institut fuer verteilte und multimediale Systeme
Werkvertrag Angestellter im Projekt EuroNF
Themen: Energy monitoring and its impact on individual user privacy - Messung des Energieverbrauchs von unterschiedlichen virtuellen Maschinen, Literaturrecherche

---

Publikationen:

ANM'08: Babel-SIP: Self-learning SIP Message Adaptation for Increasing SIP-Compatibility [HNHH08]

IPTComm'08: Automatic Adaptation and Analysis of SIP Headers Using Decision Trees [HAAM08]

PAAP'10: Optimization towards Consolidation or Throughput for Multi-thread Software [WNH10]

EPEW'11: Optimization for Multi-Thread Data-Flow Software [NH11]