FAKULTÄT
FÜR !NFORMATIK

Faculty of Informatics

Voice over IP Integration in Service Component Architecture

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Software Engineering & Internet Computing

eingereicht von

Michael Pickelbauer
Matrikelnummer 0425061

an der
Fakultät für Informatik der Technischen Universität Wien

Betreuung
Betreuer: O. Univ. Prof. DI Dr.techn. Dietmar Dietrich

Wien, 27.9.2011

(Unterschrift Verfasser/in) (Unterschrift Betreuer/in)

Erklärung

Michael Pickelbauer
Schulstrasse 21
7304 Großwarasdorf

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit - einschließlich Tabellen, Karten und Abbildungen -, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 27.9.2011

Michael Pickelbauer

Kurzfassung

Das Paradigma der Service-oriented Architecture (SOA) fand in den letzten Jahren eine weite Verbreitung. Diese wurde durch die Vorzüge bei der Interoperabilität und Erweiterbarkeit vorangetrieben. Ebenso hat sich Voice over IP (VoIP) zur Übertragung von Multimedia über Computernetzwerke wie das Internet etabliert.

Die vorliegende Arbeit beschäftigt sich mit der Integration von Sprachdiensten über paketorientierte Netzwerke in einem, auf einer SOA-basierenden, System. Ziel ist es, Informationen über VoIP-Anrufe den Diensten eines SOA-basierenden Systems bereitzustellen. Durch die Integration sollen die Vorteile und Möglichkeiten wie die Wiederverwendbarkeit oder die Komposition von Diensten eines SOA-basierenden Systems genützt werden.

Mit der vorliegenden Arbeit wird eine ressourcensparende Lösung angestrebt. Deshalb wird als Implementierung einer SOA das Framework Service Component Architecture (SCA) verwendet. Das weit verbreitete Session Initiation Protocol (SIP) wird zur Signalisierung für VoIP eingesetzt. Zum Auffangen und Abändern der Signalisierungsnachrichten wird ein Back-To-Back User Agent als Grundlage verwendet. Mit folgender Methode wird in der vorliegenden Arbeit vorgegangen: Zu Begin wird der Stand von Wissenschaft und Technik untersucht. Darauf aufbauend werden unterschiedliche Ansätze entwickelt, wie Informationen über einen Anruf einer SCA bereitgestellt werden kann. Die vielversprechendste Lösung wird gewählt, um einen Prototypen zu erstellen.

Der Prototyp ist SIP-konform und wird als ein SCA-Service realisiert. Es werden Beispielanwendungen mit dem Prototypen umgesetzt, die den einfachen Zugriff auf die Anrufinformationen sowie die durch SCA bereitgestellten Möglichkeiten, wie die Wiederverwendbarkeit, zeigen. Bei den Lasttests wurden wie erwartet Leistungseinbußen gegenüber spezialisierten SIP-Systemen gemessen, jedoch reicht bereits die erreichte Arbeitsgeschwindigkeit aus, um die Anforderungen an die Leistung zu erfüllen.

Abstract

The paradigm of Service-oriented Architecture (SOA) has gained ever increasing popularity due to its interoperability and extensibility. Similarly, Voice over IP (VoIP) has established for the transfer of media over computer networks, like the Internet.

The present thesis examines the integration of information from Voice over IP calls into a system based on SOA as well as the manipulation of these call information. The aim is to unveil whether services can access the call information in the same way as they would access information from commonplace services.

The focus of this work is to create a lightweight system. Therefore, Service Component Architecture (SCA) is chosen as an implementation of SOA. On the Voice over IP side the widely used Session Initiation Protocol (SIP) is chosen as the signaling protocol to interact with SCA. The interaction is based on a Back-To-Back User Agent as this type of SIP server is most flexible and offers numerous interaction opportunities. Various approaches regarding how the information of a call can be inserted into SCA are developed and the most promising option is implemented as a prototype.

The prototype follows the SIP standard and is implemented as a service in SCA. With the prototype, sample applications are built which show the expected improvement in development speed and re-usability of the created services. Compared to implementations not using SCA, the prototype reveals drawbacks in performance. The achieved performance, however, is already sufficient to fulfill the requirements.

Table of Contents

1 Introduction **1**

1.1 Motivation 1

1.2 Scope of Work 2

1.3 Methodology 4

2 State of the Art **6**

2.1 Voice over IP 6

2.1.1 Session Initiation Protocol 8

2.1.2 Signaling 10

2.1.3 Messages 13

2.1.4 Routing Information 16

2.2 Service-oriented Architecture 17

2.2.1 Concepts 18

2.2.2 Service-oriented Architecture Ingredients 20

2.2.3 Orchestration and Choreography 21

2.2.4 Web Service 21

2.2.5 Service Component Architecture 24

2.3 Integration of Voice over IP in Systems Based on Service-oriented Architectures 26

3 Concept and Model **29**

3.1 Analysis 29

3.2 Design 33

3.2.1 Session Initiation Protocol Message Interception 33

3.2.2 Physical Architecture 39

3.2.3 Integration in Service Component Architecture 45

3.3 Proposed Solution 50

3.3.1 Infrastructure 50
3.3.2 Message Processing 50

4 Proof of Concept **54**
4.1 Implementation 54
4.1.1 Used Components by the Prototype 54
4.1.2 Structure 57
4.2 Simulation and Results 63
4.2.1 Test Scenarios and Metrics 63
4.2.2 Test Cases and Setup 66
4.2.3 Test Results 68
4.3 Discussion 72

5 Conclusion and Outlook **78**
5.1 Conclusion 78
5.2 Outlook 80

Appendix: SDO Definition **82**

Appendix: SIPp Scenarios **85**

Literature **94**

Abbreviations

3GPP	3rd Generation Partnership Project
ALG	Application Layer Gateway
API	Application Programming Interface
AS	Application Server
ATM	Air Traffic Management
B2BUA	Back-to-Back User Agent
BPEL	Business Process Execution Language
BPM	Business Process Modeling
CPU	Central Processing Unit
DCCP	Datagram Congestion Control Protocol
DNS	Domain Name System
CR	SCA Requirement
DTLS	Datagram Transport Layer Security
ESB	Enterprise Service Bus
GNU	GNU's Not Unix!
GPL	General Public License
HTTP	HyperText Transfer Protocol
HTTPS	HyperText Transfer Protocol Secure
IETF	Internet Engineering Task Force
IMS	IP Multimedia Subsystem
IP	Internet Protocol
ISDN	Integrated Services Digital Network
ISO	International Organization for Standardization
LI	Lawful Interception
MDSD	Model-driven Software Development
MGCP	Media Gateway Control Protocol
NAT	Network Address Translation
NR	Non-functional Requirement
OASIS	Organization for the Advancement of Structured Information Standards
OSGi	Open Services Gateway initiative
OSI	Open Systems Interconnection
OSOA	Open Service-oriented Architecture
PC	Personal Computer
PSTN	Public Switched Telephone Network
RFC	Request For Comments
RTCP	Real-time Transport Control Protocol
RTP	Real-time Transport Protocol
SCA	Service Component Architecture
SCTP	Stream Control Transmission Protocol

SDO	Service Data Object
SDP	Session Description Protocol
SIP	Session Initiation Protocol
SIPS	Session Initiation Protocol Secure
SMTP	Simple Mail Transfer Protocol
SOA	Service-oriented Architecture
SR	SIP Requirement
TCP	Transmission Control Protocol
TLS	Transport Layer Security
UA	User Agent
UAC	User Agent Client
UAS	User Agent Server
UDDI	Universal Description Discovery and Integration
UDP	User Datagram Protocol
UMTS	Universal Mobile Telecommunication System
URI	Universal Resource Identifier
ITU-T	Telecommunication Standardization Sector of the International Telecommunication Union
VoIP	Voice over Internet Protocol
W3C	World Wide Web Consortium
WSDL	Web Service Description Language
WS	Web Service
WWW	World Wide Web
XML	Extensible Markup Language

1 Introduction

In an attempt to modernize air traffic control, various research has been done to utilize Voice over IP (VoIP). The air traffic control consists of heterogeneous computer systems, thus the idea is picked up to introduce the paradigm of Service-oriented Architecture (SOA). As with VoIP, the voice communication is an additional computer system, the benefits of SOA should also be facilitated. In the following, the motivation to merge these technologies and the scope of work will be defined.

1.1 Motivation

Since its introduction, the telephone system has been a circuit switched system. In such a system an end-to-end path between the participants of a phone call is set up [SPS04, p. 22]. This path is exclusively used by the participants and offers a reliable connection with guaranteed bandwidth. Hence, the telephone system is basically a closed system and does not share its capacities or wires with other systems.

With the growing number of users and increasing bandwidth, the Internet adapted in the capability in carrying video and audio transmission on a broad base. Around the millennium development in VoIP started to emerge. VoIP is the name for a technology used to transfer media like video, audio or text via an Internet Protocol (IP) network. Transferring media over a packet-switched network, such as the IP network, is a more complex challenge than using a traditional circuit-switched network. Due to the packetization of the data and the way how those packets are routed over a packet-switched network, the arrival for each individual package can not be predicted [SPS04, p. 23]. Therefore, the transmission of media in a reliable and qualitative manner over a network which is build for transferring data is a complex engineering challenge. It offers, however, various opportunities for computer systems, network infrastructure and the overall interaction with the telephone system. With VoIP every computer is able to work as a phone agent and every server can be a switch for VoIP calls. This enables the development of applications which are enhanced with VoIP to offer new possibilities and experiences.

The basic purpose of Air Traffic Management (ATM) is to operate aircrafts in the sky and on the ground. To enable safe and efficient operation, ATM defines the interplay of systems, people and procedures. The way how ATM is carried out has not changed since the sixties, though technology has improved. With improvements in technology, like increased resolution of the radar, the increasing air traffic is overcome. To modernize ATM researches in all areas is carried out. One target of the modernization is to harmonize the systems of ATM among countries and organizations.

The air traffic control is one part of ATM and is responsible for safely separating and guiding the aircrafts from and to airports. It consists of different systems, which together enable an efficient operation. In course of the modernization the idea is picked up to apply the paradigms of SOA to the computer systems involved in the air traffic control. SOA [PvdH07, pp. 389-391] is not an architecture, which can be applied directly to a project. It is just a paradigm which describes what the concepts of SOA are and which paradigms have to be followed in order to successfully build a system based on SOA. The very basic idea behind SOA is to create services that provide functionality. It can be used by other services or programs or these services can be orchestrated/choreographed (see Subsection 2.2.3) to accomplish a more complex task. Using VoIP as a service in such a system based on SOA would enable a variety of opportunities that increase flexibility to a computer system. The re-usability and orchestration/choreography possibilities of SOA could be applied to these created VoIP services. By enriching a VoIP call with information gathered in a system based on SOA the client application would not need to use a separate channel to query these parameters, e.g. by querying a database server. Current VoIP server only offer limited possibilities to influence the information carried in a call or the call itself. Often, the server can be influenced by settings or some even offer to create simple scripts. However, a query to a database or other systems is not possible with these scripts. In contrast the usage of application server [TW07, pp. 229-236] is rather tedious and complex as well it does not offer the flexibility and re-usability of a SOA based solution.

There are already attempts to add VoIP to systems based on SOA [LCLL04, HZ04, ESS+09], but there is no suitable implementation for the environment in which this thesis is dealing with. This environment requires a reliable implementation which is capable to perform on a system with comparatively weak hardware configuration to nowadays computer systems as described closer in the next section.

1.2 Scope of Work

In air traffic control different systems are used to safely maintain an efficient air traffic. The most important systems are still the radar system and the radio system to communicate with pilots as well as with the neighboring air traffic controls. Usually, these system landscapes grew historically, thus legacy systems are still in use and they consist of products from different manufacturers. The variety of systems and technologies make the interconnection of these systems a complex and tedious task. In the past, primary individual systems were improved to overcome the increasing requirements and challenges.

For example the radar systems resolution was increasingly improved, as a result of which routes of airplanes could be made more compact without any reduction in safety. However, it could also be desirable to see a feedback on the radar screen to which plane the air traffic manager is talking to, and this requires the collaboration of different systems.

This thesis is written in the course of the research to modernize air traffic control. One subproject is addressing the heterogeneity of the systems. In the course of the project the appliance of the paradigm of SOA is evaluated.

The communication in air traffic control is an important service. One of its crucial roles the system has is to communicate with the pilots cruising through their territory giving them instructions for the route, weather conditions and further information that might be necessary. In the course of the present thesis the voice communication system should also be integrated in a system based on SOA.

The aim of the present thesis is to find a solution how to realize an integration of the communication system into SOA. For this reason, a proper VoIP protocol need be found as well as an implementation of SOA. Central questions of the thesis are, if the integration is possible and how it can be realized. Also, how far the added abstraction layer of a system based on SOA is limiting the performance of the system.

For the air traffic controlling ordinary computer systems cannot be used due to the requirements in safety and availability [ED109a, pp. 59-65]. Therefore, special hardware was developed which fulfills these requirements. A drawback of the hardware is the weak performance compared to modern computer systems. Commonly those system's memory is less than 256 Mbyte and the size of the hard drive does not exceed 50 Mbyte. This hardware has been certified which takes a certain amount of time and thus changes are rather tedious. All applications developed for the air traffic control run on this hardware and thus the question need to be answered, if the system being developed is capable to run on this hardware with an acceptable performance, too.

To meet the requirements for safety and availability also the software needs to be certified. To verify an operating system, the used libraries and the developed application takes a lot of time. If a virtual machine would be used for executing the application, like with Java [GJSB05] or .NET [Pro02], it also needs to be verified. The complexity of a virtual machine would add a significant amount of time to the verification process. Also a virtual machine consumes notable resources, which are considered to be limited for the addressed system. Thus, implementations using a virtual machine should be avoided. Finding a considerable implementation to meet the limitations in hardware is part a of the problem covered by the present thesis.

With the integration of VoIP in a system based on SOA several tasks should be accomplished. Therefore, the information of a call should be used by services of a system based on SOA. This could reach from simple logging of calls, error or fault detection, statistics to more sophisticated tasks like displaying additional information to the air traffic controller up to mission critical decisions, like escalation of an unanswered call. But the system should not be restricted to passive reception of information. It should be possible to actively manipulate call information. In this way, call information could be enriched with additional data like flight number, origin of the flight, current flight sector. But also applications should be enabled to change the destination of the call depending on different features like geographic position or internal state of other systems. Thus the integration of the VoIP should be done at a state, where most information of the call can be extracted but where the call still can be unrestrictedly influenced. Further, information extraction and manipulation of the call should be possible over the hole duration of the call.

VoIP is being used for some time and a lot of software and hardware was developed, tested and proved the functionality and reliability in the field. To be able to use the present hardware and software, the interaction should be done in a standard conform manner. This means that the developed system needs to comply to the VoIP protocol in every aspect without the need to introduce any changes to the protocol. This, not only saves the investments already made, but it also preserves the extendability of the solution.

In the past, different efforts to integrate VoIP in a system based on SOA were undertaken, see Section 2.3. The majority of these solution are designed for large scale applications and most are implemented using the programing language Java. Also, these solutions depend on technologies and products which are are not suitable for the target environment of this work.

The aim of this work is to find a way to intercept VoIP communication in a transparent way for all participants. This means that the solution has to be compatible with present standards. It should also enable the alternation of the intercepted information. The limited resources have to be considered for all decisions made in this work.

It is obvious and thus expected that the integration of VoIP in a system based on SOA will affect the performance compared to a stand alone VoIP system. But the designated area of application does not need a high performance system. However, a processing speed of 100 calls per second is required.

1.3 Methodology

Initially, the problem domain is introduced and described. The motivation for the on-handed work is explained and the area of research is defined as well as the addressed problems and expected results. It also gives a brief outlook of the direction in which the solution is inclining.

For an overview of current research and products, a state of the art analysis is done. As two technologies should be integrated, each technology is examined on its own. The start marks VoIP where different important implementations are briefly described. The Session Initiation Protocol (SIP) is further described with all details which are needed for an integration into a system based on SOA. Next, SOA is introduced with all important aspects. As SOA is only a paradigm, concrete architectures are presented. One of the implementation of SOA will be chosen to be used in the on-handed thesis. Finally, related work in integrating VoIP into systems based on SOA is discussed. Additionally, related work is examined on how messages of the chosen VoIP protocol can be intercepted and changed. The results of the state of the art analysis can be found in Section 2.

Based on the problem description and the state of the art analysis, technical requirements are going to be defined and the design of the system will be worked out. The related work is analyzed if a present solution or product can fulfill all requirements. Subsequent the system will be developed in three steps:

1. Entire SIP message interception
2. Physical architecture
3. Integration in a system based on SOA

First, the issue on how the signaling messages can be intercepted and changed without violating the standard is addressed. For this purpose, different server defined by the protocol standard are analyzed if they are suitable for this task. Then, the physical architecture of the system and the needed parameters of the environment are defined. In the last step, different ways will be worked out how the developed server can be integrated into a system based on SOA and hand the collected information in an effective manner to the services. The concept and model can be found in Section 3.

To test the concept and model, a prototype needs to be developed and tested, see Section 4. Details on the implementation cover the used components, the structure of the prototype and faced problems. Subsequently, the capabilities and features of the prototype will be tested by implementing sample applications. Different performance tests will be conducted using two test scenarios. Next to the prototype, two additional systems are tested for reference. Finally, the prototype, the test results and the design are discussed.

With the conclusion and outlook in Section 5 the on-handed thesis is concluded and an outlook in the area as well as further development of the system is given.

2 State of the Art

SIP is one of the most used signaling protocols for VoIP. Therefore, it is contemplated to be used in the present work. An overview of major VoIP protocols will be given and the SIP protocol will be examined. The areas of application will be discussed followed by the concepts and functionality of SIP. Next to SIP, the paradigm of SOA are introduced. Giving a brief outline how SOA was established, the concepts and ideas are described. Subsequently, work will be discussed which already examined the integration of SIP in systems of SOA.

2.1 Voice over IP

VoIP is a technology to transfer audio or other media via an IP based network. The IP network is a packed-switched network. Data over the network is transfered in packets and thus the media has to be divided in pieces to be transfered on this network [SPS04, p. 23]. In contrast to circuit-switched networks [Bad09, pp. 4-6]), transmission of media over a packed-switched network pose different challenges.

In a circuit-switched network a channel has to be established to the target before any communication can happen. This channel persists for the complete session and can just be exclusively used by the participants. The main purpose of this kind of network was to deliver voice.

In packet-switched networks all data is transfered in packets and packets from different applications use the same network at the same time. The fact that different kind of data can be transfered at the same time can be seen as an advantage of a packed-switched network in contrast to a circuit-switched network as the available bandwidth can be used more efficient. For the transfer of voice or other media this fact is rather a burden. Media transfered over the network should be delivered without a noticeable delay for the users. The sharing of the bandwidth and the impossibility to directly influence the transfer of other data on the network is a challenge for VoIP.

To establish and to release a call signaling is necessary. In circuit-switched networks the signaling could be handled via the same channel where the voice is transfered, called in-band signaling, or via an own channel, called out-of-band signaling. Out-of-band signaling is used e.g. by ISDN with the D-Channel [Bad09, pp. 4-9].

In VoIP signaling is responsible for managing a media session. This includes the establishment of sessions, the negotiation of the media transport as well as modification and termination of sessions. The actual media is transfered by the media transport. It streams constantly the voice or other media between the participants. Signaling and media stream are independently transfered in VoIP and use different protocols. Therefore, VoIP is using out-of-band signaling although the same network is used.

In course of the modernization of the air traffic controlling, research is done in the area of the utilization of VoIP for ATM voice service [EK10, pp. C8-1 - C8-3]. The trend of major telecom provider, to converge voice and data into one network for better scalability at reduced costs, is picked up by the research teams. Air traffic control is responsible for organizing aircrafts on the ground and in the sky to safely and efficiently separate and guide the aircrafts to and from airports as well as the flow of traffic along airways. For this purpose, the air traffic controller use radar screens, radiotelephony systems and telephones. The radiotelephony system is used to communicate with the pilots of the aircrafts. This kind of communication is called ground to air communication. The telephones are mainly used for communication with other control centers, called ground to ground communication. In a first step the research is focused on the ground to ground communication to change it to VoIP, in a second step the research is expanded to ground to air communication. The EUROCAE workgroup 67 specified SIP to be used as the VoIP protocol [ED109a, ED109b, ED109c].

In the following, major VoIP protocols are introduced to give an overview. For the media transport itself the Real-time Transport Protocol (RTP) [SCFJ03] is widely used and is a de facto standard. For VoIP following major signaling protocols exist:

H.323: This signaling protocol [H3209] uses a binary message encoding similar to the ISDN signaling. It relies on the protocols H.225 [H2209] and H.245 [H2409] for call setup and management. As media transport RTCP [SCFJ03] is used. H.323 is a recommendation of the Telecommunication Standardization Sector of the International Telecommunication Union (ITU-T).

SIP: A text based signaling protocol is SIP [RSC$^+$02]. In 2002 the latest version was standardized by the Internet Engineering Task Force (IETF). The protocol is characterized by the comparative simple architecture and extendability [RS02b, RS02c, RS02a, Roa02].

MGCP: The Media Gateway Control Protocol (MGCP) [AF03] is a master/slave-protocol. All endpoints are slaves and the media gateway itself is controlling them as the master (Media Gateway Controller) [TW07, p. 35]. This results in a strong position of the carrier and no peer-to-peer communication is possible without including the master and thus accounting is easily possible. A drawback is that services to customers can only be offered on a gateway level.

As SIP was developed by IETF it is an open standard. Therefore, several open source SIP implementations are available as well as SIP servers and softphones. Compared to the other introduced protocols, SIP addresses only the signaling issues without defining any other codec or additional protocols. SIP uses standardized codecs and protocols instead. Therefore, and because of the similar syntax to the Simple Mail Transfer Protocol (SMTP) [Kle08], the SIP protocol is relatively simple to use and to understand. The 3rd Generation Partnership Project (3GPP) chose SIP as the signaling protocol in the Universal Mobile Telecommunication System (UMTS) [TW07, pp. 405-406]. As UMTS is widely spread, but not only therefore, SIP has evolved to a quasi standard.

2.1.1 Session Initiation Protocol

SIP is, as the name already suggests, a protocol to initialize sessions for multimedia over IP. With the help of SIP audio, video, text and pictures can be exchanged over an IP network. SIP is being developed by IETF and thus is an open standard. The first version of the SIP standard was released in 1999 as Request For Comments (RFC) 2543 [HSSR99]. The main purpose of this version was originally to establish, maintain and terminate multimedia sessions for two or more participants over IP [HSSR99, pp. 7-8]. In this release multimedia sessions with more than two participants such as conference calls was also payed attention. With the latest version of the SIP standard in the year 2002 the main focus changed to two-party unicast sessions. It was published as RFC 3261 [RSC+02].

The SIP standard does not cover all necessary aspects to establish a media session [RSC+02, pp. 9-10]. An application realizing VoIP with SIP consists in general of three components, which can also be seen in Figure 2.1:

Signaling: With the signaling a session is established, modified and terminated. For this purpose, SIP is used.

Session Description: To describe and negotiate the parameter of the audio or video communication, the session description is needed.

Media Transport: The recorded audio or video is encoded and transfered to the receiver by the third component.

SIP was originally designed to take minimal assumption on the underlaying transport protocol [HSSR99, pp. 18-20] and thus easily new protocols can be introduced. The SIP standard supports reliable, connection-oriented protocols like Transmission Control Protocol (TCP) [Pos81] or unreliable, connectionless protocols like User Datagram Protocol (UDP) [Pos80]. Due to the support of unreliable protocols, simple methods for committing and retransmission of messages are included. In latest SIP standard four transport protocols are defined [RSC+02, pp. 179-180]:

- UDP
- TCP
- Transport Layer Security (TLS) [DR08]
- Stream Control Transmission Protocol (SCTP) [SXM+00]

Note, TLS means TLS over TCP. As mentioned, *UDP* is a connectionless protocol where *TCP* and *SCTP* are connection-oriented transport protocols. For securing SIP connections *TLS* is used. TLS itself can be used on different transport protocols, but in the SIP standard, TLS means TLS over TCP. Figure 2.1 shows SIP in the ISO OSI layer with the defined transport protocols. The support for TCP and UDP need to be implemented by all SIP applications [RSC+02, p. 142], however, additional protocols defined within the RFC 3261 or other RFCs can be implemented. UDP is the preferred protocol for SIP, because it offers more control to the application for the transmission and retransmission of messages as well as it enables multicast messages.

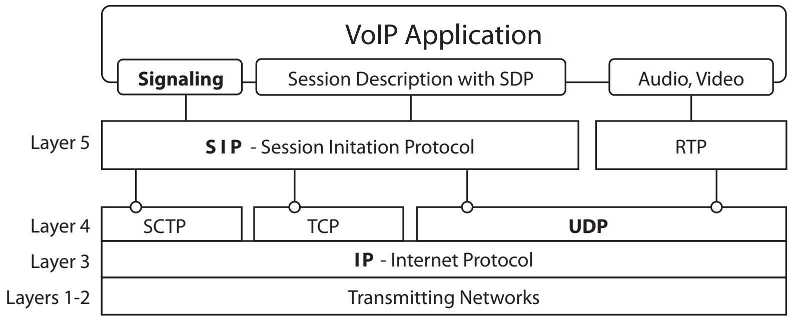

Figure 2.1: SIP in the ISO OSI layer model with different transport protocols. The component names printed in bold are in focus of the on-handed thesis.

Additional RFCs add support for more transport protocols [Bad09, pp. 274-276]. RFC 4168 [RSC05] enables SIP to use TLS on over SCTP. With the release of the Datagram Congestion Control Protocol (DCCP) [KHF06] a draft for using SIP over DCCP [JM07] was proposed in October 2007, but at the time of writing it was not approved as a standard. This draft also adds the possibility to secure the connectionless protocols UDP and DCCP with the use of Datagram TLS (DTLS) [RM06].

As mentioned the multimedia session itself cannot be described with SIP. To describe the session an additional protocol is necessary and most commonly the Session Description Protocol (SDP) [HJ98] is used to negotiate and describe the multimedia session. This splitting can also be seen in Figure 2.1. The session description via SDP is embedded in the SIP message when a session is initialized.

The last component also displayed in Figure 2.1 to successfully establish a session with SIP is the media transport. As mentioned at the beginning of this Section, RTP [SCFJ03] is widely used to transport media between the parties. The parameters of the multimedia session like encoding and network port information are exchanged using SDP.

Despite just the establishment of multimedia sessions is described in the standard, SIP is designed to be easily extended. Lots of extensions already exist, like instant messaging [CRS+02], user presence [Ros04] or conferencing [Ros06].

2.1.2 Signaling

SIP is a text based protocol similar to SMTP [Kle08] or the HyperText Transfer Protocol (HTTP) [FGM+99] and uses a request/response model. A message consists of a start-line, one ore more header fields and a content [Bad09, p. 307]. The content can be any arbitrary text, e.g. at the call setup it contains an SDP or if an instant message is sent, it contains the message. The structure and type of SIP messages will be briefly described in Subsection 2.1.3.

In SIP an user is identified with a SIP address [Goo02, p.1507]. A SIP address is an Universal Resource Identifier (URI) which is used to address a logical destination. An URI consists at least of three parts:

- Protocol
- Address
- Name

This three parts are necessary to uniquely identify and access a specific resource. Additional to those, various optional parts can be added to an URI. The *protocol* part names the network protocol that must be used to access the resource. The *address* shows the network location of the resource and the *name* uniquely identifies the resource within the network location.

An example for a SIP address is *sip:bob@biloxi.example.com*. *sip* indicates that SIP is needed as the communication protocol. The colon is following the name of the resource, in this example *bob*. In SIP the resource name is the user name. The user name can as well be the telephone number of a Public Switched Telephone Network (PSTN). In this way, PSTN numbers can be addressed within SIP. *biloxi.example.com* represents the network location. This can be either the network location of the user or the location of the domain where the user is registered. The use of the domain location in SIP addresses enables location transparency. Due to this, the SIP user can roam inside or even outside of the domain (hence having each time a different network address) but still be reachable via the same SIP address.

With SIP it is possible to establish a peer-to-peer connection directly between two SIP clients, but in a typical SIP scenario the clients do not know the current network location of each other. To find the current network location of a SIP user, a SIP proxy server is used. The location part of the SIP location points the client to the SIP proxy which knows the current temporary network location of the user. When a request is received by the SIP proxy, the current network location of the user is looked up and the request will be forwarded. Such a proxy is called *inbound proxy*. Through this procedure location transparency is achieved.

Usually, there is one more proxy between the caller and the inbound proxy of the calling domain, called *outbound proxy*. This kind of setup is called *SIP trapezoid* and a common call setup scenario with SIP.

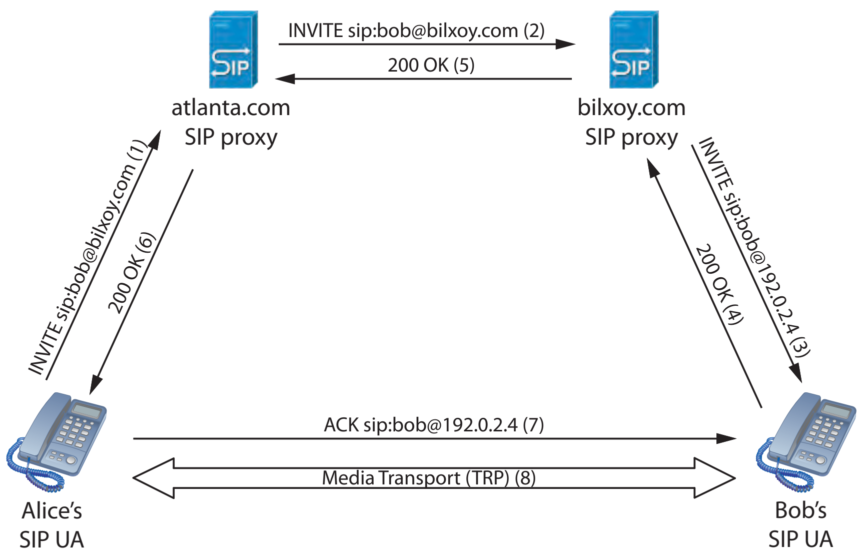
INVITE sip:bob@bilxoy.com (2)
200 OK (5)
atlanta.com SIP proxy
bilxoy.com SIP proxy
INVITE sip:bob@bilxoy.com (1)
200 OK (6)
INVITE sip:bob@192.0.2.4 (3)
200 OK (4)
ACK sip:bob@192.0.2.4 (7)
Media Transport (TRP) (8)
Alice's SIP UA
Bob's SIP UA

Figure 2.2: SIP session setup example with SIP trapezoid [RSC+02, p. 12]

Figure 2.2 shows a SIP trapezoid with an example of a message flow to establish a session. The SIP user Alice is initiating a session by sending an *INVITE* message to the corresponding outbound proxy atlanta.com (1). This proxy is discovering the inbound proxy of the called user and forwards the message to the proxy (2). The inbound proxy bilxoy.com resolves the current network address of the called user Bob and forwards the *INVITE* message to the SIP User Agent (UA) (3). As can be seen in the figure, the destination SIP address of the message changed, as the real address of the user is now known. The phone of Bob is ringing and when the phone is picked up, the SIP UA sends a *200 OK* response message to the inbound proxy server biloxy.com (4). This is done as in SIP all response messages have to travel the same path to the sender as the request used. The inbound proxy bilxoy.com sends the *200 OK* message to the outbound proxy atlanta.com (5) and finally the message is delivered to the sender Alice (6). To confirm the receipt of the *200 OK* message the UA of Alice sends a *ACK* message directly to Bob, as his current network address is now known by the sender too. After the *ACK* message, the media session is established (8).

The SIP standard defines next to endpoints, called UA, (like a softphone on a PC or a SIP phone) different types of servers, those are [RSC+02, pp. 20-26]:

- Registrar
- Location Service
- Proxy Server
    - Stateful Proxy
    - Stateless Proxy
- Redirect Server
- Back-to-Back User Agent (B2BUA)

The listed servers are in the following shortly introduced, a detailed description and discussion of the differences is done in the course of the design, see Subsection 3.2.1.

A *registrar* accepts REGISTER requests of his domain and forwards the information of those requests to the location service [TW07, pp. 181-183].

The main purpose of a *location service* is to store the current location of the SIP user of the domain [TW07, pp. 191-192]. The standard does not define how the current location of a user is stored or how the proxy or redirect server can query a location service. Nevertheless, the standard defines how bindings can be created and deleted with the REGISTER method.

The *proxy server* in the SIP standard has similar responsibilities like the namesakes in other protocols and standards. The main purpose is to take a message of a client and forward it to an entity closer to the target [TW07, pp. 183-189]. The SIP standard further distinguishes between a stateful and a stateless proxy server. A *stateful proxy* maintains the transaction of the forwarded requests during the processing [TW07, pp. 183-189]. Due to the knowledge of the state of a request the stateful proxy server can generate responses. The other kind of proxy is the *stateless proxy* [TW07, pp. 183-189]. It forwards each request without maintaining any information of the state or transaction of the request. Received requests are forwarded directly downstream and received responses upstream.

A *redirect server* is a server which is responding to a received message with a notice that the target is moved to an other network location and thus is not reachable at the used network location [TW07, pp. 189-191]. A set of alternate locations of the target is provided in the response.

A B2BUA is a special kind of SIP proxy server [TW07, pp. 209-210]. In contrary to the SIP proxies defined by the standard, it can also generate request messages next to response messages and it can alter messages in any way before they are being forwarded.

Beside to the described types of SIP servers, additional types exist. One worth to mention is the *Application Server* (AS) [Bad09, pp. 35-41]. An application server is a server which is hosting applications that can be accessed via the Internet or Intranet. It offers frameworks, utilities and a platform for execution to the applications. With the provided Application Programming Interfaces (API) telecommunication networks can be accessed.

2.1.3 Messages

The SIP standard and the available extensions define a fair amount of different SIP message types. SIP messages can be categorized in request and response messages [Bad09, pp. 303-313]. A message exchange is initiated by sending the initial request message. The communication partner will answer with an appropriate response message. As this behavior is similar to those of a server and a client, the logical parts of the SIP system which are sending requests are called User Agent Client (UAC) [RSC$^+$02, p. 26] and the parts which receive requests and generate responses are called User Agent Server (UAS) [RSC$^+$02, p. 26].

SIP messages are text-based and line oriented. The structure can be grouped into three parts [Bad09, p. 307]:

- Start-line
- Header fields
- Body (optional)

An example of an INVITE message containing all three parts is shown in Listing 2.1.

```
INVITE sip:bob@biloxi.example.com SIP/2.0
Via: SIP/2.0/TCP client.atlanta.example.com:5060;branch=z9hG4bK74bf9
Max-Forwards: 70
From: Alice <sip:alice@atlanta.example.com>;tag=9fxced76sl
To: Bob <sip:bob@biloxi.example.com>
Call-ID: 3848276298220188511@atlanta.example.com
CSeq: 1 INVITE
Contact: <sip:alice@client.atlanta.example.com;transport=tcp>
Content-Type: application/sdp
Content-Length: 151

v=0
o=alice 2890844526 2890844526 IN IP4 client.atlanta.example.com
s=-
c=IN IP4 192.0.2.101
t=0 0
m=audio 49172 RTP/AVP 0
a=rtpmap:0 PCMU/8000
```

Listing 2.1: Example of a SIP message for establishing a call

The first line is the *start-line*. It contains either the name of the SIP method or the response code. This line is used to identify the message type. Basically, the structure of all SIP messages are the same, only the format of the start-line will differ depending if the message is a request or response. In the lines 2 to 10 are the header fields of the message. A header field consists at least of the name of the header and a value separated by a semicolon. E.g. in line 10 the header field *Content-Length* contains the value *151*, it indicates the size of the body. A body is optional and is added after the header fields. The body and the header fields are separated by an empty line. In this example a body is present ranging from line 12 to 18.

Start-Line

The first word in the start-line of *request messages* is the name of the request, followed by the target SIP address and the SIP version. RFC 3261 defines six request messages [Bad09, pp. 303-304]:

- REGISTER
- INVITE
- BYE
- OPTIONS
- CANCEL
- ACK

The *REGISTER* message is used to register, unregister or update user information. With the *INVITE* message a session establishment is being started. It is the first message of the message flow and it contains information of the target user and already information to negotiate the media session. The *BYE* message is used to initialize a closing of a running session. The *OPTIONS* message is used to query an user agent or a proxy for capabilities. With the *CANCEL* message a running transaction can be canceled. Finally the *ACK* message confirms responses if required.

The start-line of *response messages* begins with the SIP version, followed by the *Status-Code* and the *Reason-Phrase* [Bad09, pp. 309-310]. The *Status-Code* is a number which is identifying the message like the responses in HTTP. The response messages are grouped into six response classes which can be distinguished by the first number [Bad09, pp. 306-307]:

- 1xx: Provisional Responses
- 2xx: Success Responses
- 3xx: Redirection Responses
- 4xx: Client Error Responses

- 5xx: Server Error Responses
- 6xx: Global Failure Responses

The *1xx Provisional Responses* inform a client of the processing of the received request, e.g. 180 Ringing. The class of *2xx Success Responses* are used to signal the successful reception or acceptation of a request, e.g. 200 OK. With the *3xx Redirection Responses* the requester is advised that further actions are needed to complete the request, e.g. 302 Moved Temporarily. *4xx Client Error Responses* are used to give notice to the client of a faulty message or that the server cannot execute the message, e.g. 401 Unauthorized. If the SIP server cannot process an apparently valid request, a *5xx Server Error Responses* is generated, e.g. 500 Internal Server Error and a *6xx Global Failure Responses* is used to signal that the request cannot be processed on any SIP Server, e.g. 600 Busy Everywhere.

Header Fields

The header fields contain different information of the SIP message and can differ depending on the message type [Bad09, p. 308]. Like new message types also new header fields can be introduced by extending standards. Header fields can depend on the message type or can be optional, but every SIP message has to contain the following mandatory header fields [Bad09, p. 313]:

- Via
- Max-Forwards
- To
- From
- Call-ID
- CSeq

The field *Via* is important for routing purposes. Every entity creating or forwarding requests on its way to the destination enters the own address in a *Via* field. The response message will then travel according to this list of entered addresses back to the issuer of the request. There are other fields for routing purposes, which will be described in Subsection 2.1.4. The *Max-Forwards* field defines the number of times the message is allowed to be forwarded and is decreased on every hop. This is used to identify loops in the routing. The *To* and *From* field state the receiver and the sender contact, respectively. The *Call-ID* is a unique number and identifies the call and *CSeq* is used to identify and order transactions.

2.1.4 Routing Information

SIP offers different possibilities to influence the route (on SIP protocol level) of a message while traveling to the destination. For this purpose, a SIP message can carry different header fields containing information of the route:

- Via
- Route
- Record-Route

In this context, the influence of the fields From, To and the SIP address in the start-line of request messages is also discussed.

As already described in Subsection 2.1.3, the *Via* header field [RSC+02, pp. 39-40] contains information on the route the message has taken to get to the examining entity. Before sending or forwarding a message, a Via header field has to be inserted with the own location. Different header field *parameter* can be added, e.g. the branch parameter contains the transaction ID of the location. This information is needed to send the response via the exact same entities back to the requester, as each entity might maintain a transaction state.

While the Via header field shows a history which entities has been passed, the *Route* header field lists to which entities this message has to be sent next on the route to the destination [RSC+02, p. 177]. As the opposite of the Via header field, the own location has to be removed from the Record header field before the message is forwarded. This list of entities can be either set by the sender, e.g. to ensure the message is passing an outbound proxy or it can be requested by previous SIP messages of this dialog.

A proxy server passed on the way to a destination can ensure to receive all messages of the dialog by inserting a *Record-Route* header field [RSC+02, p. 17]. This field is accumulated on the way to the destination. The UA of the destination saves the Record-Route header in a Record-Route set. This Record-Route set will be inserted as Route fields in any future messages sent within this dialog. In this way, the entities which entered a Record-Route header will receive the message, too. Because both participating clients need to know the list of entities which has to be passed in this dialog, the Record-Route list is being added to the response message where the sender also saves the Record-Route header in a Record-Route set.

The header fields From and To contain the name of the sending user and the user who is addressed respectively [RSC+02, pp. 36-37]. As this values contain the original full and unresolved SIP address, they are not used for the routing of the message. When sending a request, the SIP address of the destination is being placed in the start-line after the type of the request. This address can be altered on the way to the destination, e.g. when an inbound proxy resolves the current location of the client. Replies do not have a SIP address in the start-line and it is not needed, because the Via fields are used to send the message back to the requester.

2.2 Service-oriented Architecture

In recent years, the term SOA was used in many ways to promote products, systems and technologies. Even webhoster advertised to have a SOA. But in most cases it was inappropriate to use the term SOA to describe their products, but it shows how popular SOA got in the past years.

A definition of SOA cannot be found easily, because there are many aspects connected to SOA. Also, as many manufacturer use SOA in association with their products, they define SOA in their own way. The term SOA was introduced by Gartner analysts and their first reports of SOA are published by Schulte and Natis [SN96].

In a paper published by Gartner SOA is described as [Nat03, p. 2]:

> ”*... SOA is a software architecture that starts with an interface definition and builds the entire application topology as a topology of interfaces, interface implementations and interface calls. SOA would be better-named ”interface-oriented architecture.” SOA is a relationship of services and service consumers, both software modules large enough to represent a complete business function.*”

Thomas Erls wrote a couple of well-cited books of SOA and he defines it as follows [Erl05, p. 54]:

> ”*SOA is a form of technology architecture that adheres to the principles of service-orientation. When realized through the Web services technology platform, SOA establishes the potential to support and promote these principles throughout the business process and automation domains of an enterprise.*”

An other definition is given by Nicolai Josuttis [Jos08, p. 24]:

> ”*SOA is an architectural paradigm for dealing with business processes distributed over a large landscape of existing and new heterogeneous systems that are under the control of different owners.*”

And finally a definition formed by more than one person on the Wikipedia in 2011 [Wik]:

> ”*Service-oriented architecture (SOA) is a flexible set of design principles used during the phases of systems development and integration in computing. A system based on SOA will package functionality as a suite of interoperable services that can be used within multiple separate systems from several business domains.*”

These definitions show that SOA is something abstract that can be interpreted differently. But most of the definitions contain the fact that SOA uses well defined interfaces to provide services. Important, and therefore also mentioned in some definitions, is that the access to the services need to be in an interoperable way. This makes the utilization of the services possible by different systems and platforms in a heterogeneous system. The definition given by Garner is taken as a basis in the on-handed thesis. In the following all important aspects of SOA are described closer.

The most important fact about SOA is that it is a paradigm. Therefore SOA is not a concrete tool, framework or even a concrete architecture. It is a mind set, concept, draft or approach for building a practical software architecture to solve a given problem. This implies, that SOA cannot be bought as a tool or a blueprint, which just has to be applied to achieve the benefits of SOA. The paradigm of SOA has to be applied for concrete situations to make concrete decisions under certain circumstances.

Another important case of SOA is, that it is meant for large distributed software systems [Jos08, pp. 3-5]. When a company is growing, more systems and interactions with other companies are added, which have to be integrated. SOA is prepared for handling complex distributed systems, it easies the way to find and access entities which offer certain functionalities.

In large distributed software systems there are usually more than one owner of the individual systems and services. They can be owned by different departments or even services can be used from different companies. SOA is designed to cope with different owners and contains processes and practices to handle it.

The last issue of large distributed software systems is heterogeneity. When software systems grow and the time passes by, different technologies and types of systems are used. In the past, it was tried to leverage the heterogeneity by harmonizing the systems. Unfortunately, large systems cannot be completely harmonized. SOA accepts that fact and works with the heterogeneity. This approach to accept heterogeneity instead of trying to fight it, is one of the qualities of SOA and can change the way how large distributed software systems are looked at.

SOA does not have to be build from the scratch. It could be build on existing systems by adapting and wrapping applications into services [PvdH07, p. 390]. But not all existing applications can be used in services. The legacy programs still have to fulfill the requirements to services in SOA. When those applications not fulfill the requirements on services, they need to be adopted. If the adoption of all legacy applications is too time-consuming or cost-intensive, they can be recreated from the scratch as services. Of course SOA can be build from the scratch or it could be a combination of both.

2.2.1 Concepts

SOA is based on three technical concepts which help to fulfill the targets [Jos08, pp. 21-23]:

- Services

- Interoperability
- Loose Coupling and Seclusiveness

In software development, addressed problems are abstracted and transformed into rules. In this way, the reality is implemented as a software system. An abstraction can be done using different views on the problem. SOA focuses on the business aspects of the problem when the abstraction is done. The solutions for these problems are provided in *Services* [Jos08, pp. 33-45] and this fact gives also the name to SOA. The target of SOA is to build large distributed software systems in abstracting business functions and rules. In this way, clear structures are created where business aspects are put in the spotlight and the technical realization is being hidden. The hole system gets service oriented.

To establish connections between distributed systems is very crucial. The faster and easier this can be done, the better, because it is possible to quicker complete an application. *Interoperability* [Jos08, pp. 21-22] reflects the fact, that connections between distributed systems can be established easily. The desire for interoperability is not new with SOA, but here it is a very essential point. It is used to build principles to interconnect various distributed systems quickly and easily to accomplish business tasks. How interoperability is achieved depends on the implementing system. However, the usage of established standards is a way to implement interoperability.

Today, lot of different systems are used which get integrated with each other. Processes get increasingly automated over their hole life span and the systems are more globalized. If many systems work under such a condition together, there is the possibility that a small error can break the hole system and so bring the business to a stop. Also one change in a specific part could affect unexpectedly other parts in different systems. For these reasons, fault tolerance [Jos08, p. 47] is important for large distributed systems. But not just fault tolerance is important, flexibility and scalability are also very desired in such systems.

The key to achieve these goals is *loose coupling and seclusiveness* [Jos08, pp. 47-62]. Loose coupling is a concept to minimize dependencies. The reduction of dependences to a minimum ensures that changes or faulty behavior in parts of the system does not affect other parts. In this way, loose coupling and seclusivness brings fault tolerance and flexibility. Additionally, loose coupling improves scalability as tasks can be more easily parallelized. For large distributed software systems, loose coupling is not just important for technical aspects, but for organizational aspects, too. This means, that the system should enable the expertise to decide how decentralized it wants to be. Decentralization could also be a form of loose coupling. But those systems need to have a certain common base to be able to interact with each other. Loose coupling has many facets and it shows that SOA is just a paradigm and does not provide a prefabricated solution. Loose coupling is needed for SOA, but the degree and implementation depends on the decision made for concrete problems which are then reflected in the concrete architecture.

2.2.2 Service-oriented Architecture Ingredients

For a successful SOA it is not enough to implement the proposed technical concepts, more is needed. As in the introduction shortly noted, in large distributed software systems processes have to be introduced to regulate different aspects and responsibilities. Depending if SOA is build on top of a existing system or a new one, the technical concepts of SOA have to be introduced appropriately. This means that the introduction has to be done in the right speed and the right scale. Also the right size of centralism has to be determined as well as the corresponding processes have to be introduced. Important aspects covered by SOA are [Jos08, pp. 23-27]:

- Infrastructure
- Architecture
- Processes
- Governance

The *infrastructure* is the technical part of SOA which enables interoperability. Mostly, the infrastructure is based on the Enterprise-Service-Bus (ESB) [Jos08, pp. 63-79]. Basically the responsibility of the ESB is to enable server calls between heterogeneous systems. This also includes additional tasks like data transformation, routing, handling of security and reliability, service-management, monitoring and logging. All tasks done by the infrastructure have to follow the principle of loose coupling.

The concepts and standards of SOA give a lot of room for decisions. When developing a concrete *architecture* a lot of such decisions have to be made based on the concrete requirements and general conditions. Also, these decisions have to be harmonized with each other. Some decisions might be:

- Classification of the kinds of services
- Degree of loose coupling
- Allowed data types at interfaces
- Definition of policies, rules and patterns
- Clarification of roles and responsibilities of persons and systems
- Decision on infrastructure, used standards and the used versions

These are just some examples and a lot more decisions have to be made until the final concrete architecture is set up.

All large systems have *processes* for accomplishing certain tasks. For example, a process can be the deployment of a new software version. The processes might be introduced explicitly or they might develop without being obvious to the participants. Changes at applications have to pass a lot of persons, teams and departments until they end up in production. In SOA following processes can be differentiated.

In Business Process Modeling (BPM) [Jos08, pp. 103-124], a business process is divided into smaller activities or tasks. Those activities or tasks can then be implemented into services.

For the lifecycle of a service [Jos08, pp. 169-177] a own process is defined. It starts at the identification of services and covers the states over the lifetime. This covers design, implementation, deployment and defines how services have to be removed from production at the end of their lifetime.

Governance [Jos08, pp. 324-330] is a special kind of process. It is a metaprocess to control all processes in SOA and the SOA strategy itself. This process is also used to introduce SOA and the associated processes. It is usually executed in a centralized team which takes care of the infrastructure, architecture and the processes. Also, this team works on developing a common understanding of SOA among all participants in SOA. As already noted this participants can be different persons, teams, departments or even external companies. This team needs support of the management, because it takes time, resources and courage to manifest SOA in the organization.

2.2.3 Orchestration and Choreography

Orchestration is a concept for a system where a centrally controlled set of workflow logic enables the interoperation between various different applications [Erl05, pp. 200-207]. The main benefit of orchestration is to be able to merge large business processes and so connect different processes through a defined workflow. This workflow can contain rules, conditions and events. With the use of orchestration a change in the workflow does not need a change of the participating application. Also the orchestration enables interoperability between application e.g. by transforming data. In a service-oriented environment, services are predestined to be used in a workflow, also the orchestration is usually again represented as a service.

Choreography is an other way how services or applications can be composed to fulfill a more complex function. But other than orchestration, there is no single entity which is responsible and controlling the execution [Erl05, pp. 208-215], e.g. Business-to-Business interaction. It acts as a community interchange pattern for collaborative purposes, whereas a orchestration is usually a organization-specific workflow.

2.2.4 Web Service

Web Service is the most common technology to build SOA. This is because it implements a lot of SOA principles, but simply the usage of Web Services does not make a system to a SOA.

Haas et al discusses in [HB04] a Web Service as a software system which supports interoperable machine-to-machine communication over a network. The interface is described in a machine readable format (most commonly using the Web Service Description Language (WSDL) [CWMR07]) and the interaction with a Web Service happens in a manner described by the interface using SOAP[1] [GHM+07] messages.

Web Services are very popular and a lot of manufacturers support them. The main reason is because Web Services are built on Internet standards and itself is being standardized by the World Wide Web Consortium (W3C). The used standards [PvdH07, p. 390] are:

- WSDL
- SOAP
- Universal Description Discovery and Integration (UDDI) [CHvRR05]

These standards build on HTTP and Extensible Markup Language (XML) [BPM+08]. For almost any common computer language a library for calling and building Web Services exist, this enables every environment to use and build Web Services. With the existing tool support for a lot of development tools (e.g. Apache Axis2 Tools [Axi]), development of Web Services is even simpler and quicker.

Chinnic et al. specifies in [CWMR07] WSDL as a XML format which is used to describe interfaces in an abstract way which is neutral to implementing technologies and systems. For loose coupling it is important to have a service description which is independent to the service implementation in a way that it hides details on the implementation. The service description and the service implementation build a pair, where the service implementation can be changed without changing the exposed service description.

A WSDL consists of two parts, an abstract and a concrete description [Erl05, pp. 133-136]. In the abstract description, the interface characteristics are described without any reference to technologies used to host the service or how the the services can be accessed. It contains a list of operations which are grouped to an *interface*. Each of this operations define input and output messages, e.g. parameters and return values, respectively. To be able to access a service described in WSDL, information on concrete technologies and location is needed. The concrete description part contains this information. It consist of binding, endpoint and service. The *binding* defines the requirements to establish a connection to the service. In this way, the required transport technology to invoke the service for this binding is specified. It is possible to define different bindings using different transport technologies to access the same service. A binding can be defined for the entire interface or for a specific operation. The physical address to access the service via a binding is defined in an *endpoint*. The separation of bindings and endpoints enables to change the location information independent to the binding. In terms of WSDL, *service* is referred to a group of endpoints.

[1]As of version 1.2 of the SOAP specification, the word *SOAP* is no longer an acronym that stands for *Simple Object Access Protocol*. It is now considered a standalone term.

To invoke Web Services *SOAP* [GHM+07] is used. SOAP is a standardized, flexible specification of a transport protocol for sending messages. The transfered messages are XML documents over an underlaying transport protocol, which in principle can be any protocol as long as a binding is defined. The most common are:

- HTTP [FGM+99]
- HTTPS [Res00]
- SMTP [Kle08]

To lighten the coupling in SOA, the pattern of service registry (also known as service broker) can be used [PvdH07, p. 392], see Figure 2.3. A service registry maintains an index of available services. Each service provider registers itself at the service broker (1). When a service consumer (also known as service requester) needs to invoke a service, it does not keep information on the binding and thus on the physical location of the service. Instead it queries a service registry to find a concrete service implementation (2). The registry returns a binding of the desired service and the service consumer can invoke the service (3).

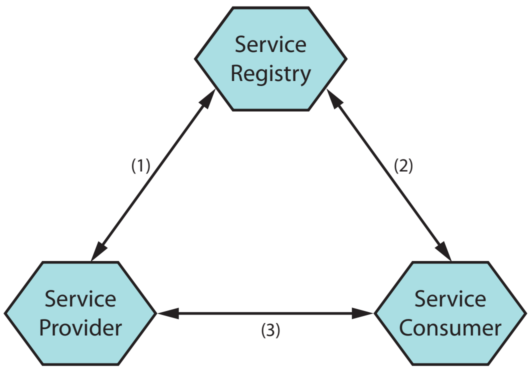

Figure 2.3: Service broker [PvdH07, p. 392]

UDDI [CHvRR05] is a specification for a service registry to implement such a functionality for Web Services [PvdH07, p. 392]. UDDI is based on a set of industry standards such as HTTP [FGM+99], XML [BPM+08], XML Schema [WF04] and SOAP [GHM+07]. A Web Service provider registers itself at a UDDI by submitting the WSDL interface description. The UDDI keeps the description and could add additional information about the service. This can be information on reliability, trustworthiness, quality of service, to name a few. When a service consumer queries the broker it receives a WSDL file which contains the physical endpoint of the service provider. The benefit of the usage of a service registry is, that the service provider can change the physical binding without breaking deployed service consumer and it can hold additional information about the service provider.

2.2.5 Service Component Architecture

Service Component Architecture (SCA) (pronounced scar) [SCA] is a specification for a model to build applications and systems using SOA. The development has been started by Open SOA (OSOA) [OSO], it is a vendor group of big players in the industry like IBM, Oracle, SAP, just to name a few. With the version 1.0 of the specification published in March 2007, the work on the formal standardization within the Organization for the Advancement of Structured Information Standards (OASIS) [OAS] began. With the work on the SCA specification, the Open SOA group also worked on the Service Data Objects (SDO) [SDO] specification. SDO are designed to simplify the way how data is handled from heterogeneous sources, e.g., relational databases or Web Services. SDO provide also additional values like the ability to track changes on data. SDO is the preferred way how complex data structures are transfered between services in a SCA.

In SCA *components* are used to build an application [Cha07, pp. 3-5]. The components in SCA can be understood as services in terms of SOA. These components can be built in any computer-language like Java, C++ or even with technologies like Business Process Execution Language (BPEL) [BPE] or the Spring Framework [Spr]. SCA uses a common assembly model to combine these components and build in this way applications or services which can be hosted as Web Services.

Elements of SCA are:

- Component
- Service
- Reference
- Composite
- Wire
- Binding
- Policy Framework

As already mentioned the *component* contains one piece of business logic and is the smallest unit in SCA [Cha07, pp. 7-8]. The visual representation with all parts of a composite can be seen in Figure 2.4. A component can offer *services* defined by an interface, which can be used by other components. If a component requires a service, it is called *reference*. A component can also have one or more properties. These properties are data values and can be configured externally to influence the behavior of the component.

In SCA components can be grouped logically together. This group of components is called *composite* (called module in earlier versions of SCA) [BBB$^+$05, pp. 9-12]. Composites can again be grouped together and so a well-defined set of abstractions can be established with which an application can be realized. The components and composites grouped in

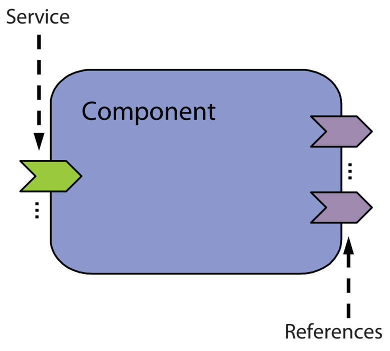

Figure 2.4: SCA Component [BBB+05, p. 10]

a composite can run in the same process, in different processes on the same machine or on different systems. Figure 2.5 shows how components and composites can be grouped together.

One reference of a component is connected to a service of an other component using a *wire* [BBB+07, pp. 36-43]. The exact communication between the components can be different. It depends on the specific runtime used, which bindings are specified and within these parameters, the wire can be configured. If no explicit wire is defined, the components are connected using the auto-wire. In this case, the runtime decides how the communication is established. Services or references defined by a component inside a composite can be made accessible from outside, this is called *promotion*. Wires and promotion can also be seen in Figure 2.5.

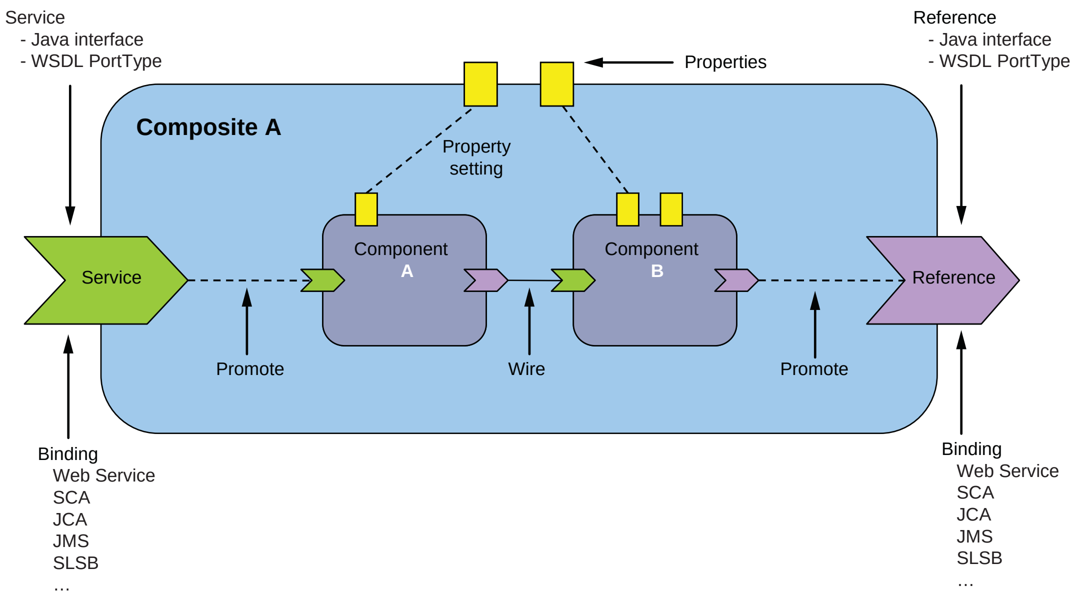

Figure 2.5: SCA Composite [BBB+07, p. 10]

When services or references are exposed to the outside world, a *binding* is used [Cha07, p. 9]. A binding specifies a particular protocol how it can be accessed. The definition of the used protocol is done independently of the definition of the service or reference. In this way, the implementation needs not to care about the protocol via which it is going to be reachable. The protocol used for the binding depends on the specific runtime.

SCA defines a own *policy framework* [BBC+07] so developers can let the container know what they intended. The framework defines two categories of policies. The *interaction policies* can be applied to services and references. They affect the way how the interaction between service provider and service client take place. For example, a policy ensures that the messages exchanged are confidential and thus have to be encrypted. The second kind of policies defined in the policy framework of SCA affects the way how components themselves behave. This kind of policy is called *implementation policies*.

A policy realization is dependent on the runtime container for the implementation policies and on the binding for the interaction policies [Cha07, pp. 18-19]. This makes it possible to realize policies as an inherent part of the container or binding. For example, a binding using HTTPS [Res00] will always send messages encrypted and thus implement the policy for confidential messaging. Other policies may be implemented by the container or the binding, or they might even be incapable to provide certain policies at all.

SCA is a new way to build SOA. It implements a lot of SOA concepts and offers a lot of possibilities to integrate into common SOA technologies. The most important difference to current SOA technologies like Web Services is, that SCA does not use a service bus like ESB. This makes this specification also interesting to be used for smaller scale projects and applications. Also the fact that it specifies a C/C++ implementation of a runtime is one of the major reasons to use SCA in the solution of the present thesis.

2.3 Integration of Voice over IP in Systems Based on Service-oriented Architectures

In the past decade, SIP was topic for a wast amount of research and development. The results can be seen in the various extensions of SIP like messaging [CRS+02], presence [Ros04], conferencing [CJ08, JL06, RSL06] and as well as the effort to integrate VoIP or in particular SIP into existing structures, e.g. the IP Multimedia Subsystem (IMS) [Bad09, pp. 41-47] or a Web Service based framework for VoIP [HZ04].

One way how concepts of SOA are used in combination with SIP or overall with VoIP is to abstract the implementing VoIP protocols or even the used infrastructure. Elsholz et al. [ESS+09] propose a framework based on Open Services Gateway initiative (OSGi) [Osg07] to hide implementation details of the communication. The framework addresses client applications and defines abstracted interface for the access of VoIP functionality. Not until the runtime an appropriate protocol and implementing component is selected to perform the call. The advantage of the abstraction is the easier and faster development. Also different VoIP protocols can be supported without the need of detailed knowledge on the specific protocols. A subsequent advantage is that new or updated protocols can be introduced by changing only small amounts of the application code.

Hillenbrand and Zhang [HZ04] introduce a framework for back-end applications to abstract VoIP protocols as well as the used infrastructure. This is accomplished with Web Services which could be hosted either in the own network, if the VoIP infrastructure is also run in-house, or at a VoIP provider. With the use of Web Services the VoIP infrastructure can be changed to/from a VoIP provider or a VoIP provider can be easily swapped. Also the used VoIP protocol can be changed without the need to change the application which is using the VoIP infrastructure. The further advantage is the simplification in development of VoIP back-end applications.

Where Hillenbrand and Zhang [HZ04] propose a complete substitution of the usage of VoIP protocols with Web Services, researches in extending VoIP protocols like SIP with WS are done, too. Chou et al. [CLL06] and Liu et al. [LCLL04] propose a dual stack solution where SIP is extended with Web Services to offer broader functionality. The dual stack approach supports regular communication via SIP. This enables present products and applications to be used straightaway with the solution. A call can also be established via Web Services, but the main purpose of them is to offer discovery, maintenance, monitoring and updating of SIP endpoints.

Different possibilities exist to intercept or interact with SIP messages. An approach to work with SIP requests in Java exists with SIP Servlets [CK08]. They offer a similar handling of SIP requests like HTTP servlets. Such SIP Servlets run within an application server which offers a framework for processing SIP requests. With this framework the development of applications can be done in a well known manner as it uses known concepts of Servlets. It offers also the flexibility to use all facets of the SIP standard. With such a SIP Servlet a SIP endpoint can be implemented, which is processing and answering incoming requests. Also the realization of a stateful proxy or a B2BUA is possible. As the SIP Servlet is running within an application server, it can also use all the advantages and features the application server offers like persistency or centralized logging.

An Application Layer Gateway (ALG) [SH99, p. 6] is usually found in routers or firewalls. An ALG is needed when Network Address Translation (NAT) is being applied. NAT is used to translate a network address from one network realm to an other. This is needed, when network addresses of a local network are not valid in an outside network. Usually, Internet applications will work without problems, but some applications face problems when NAT is applied. This happens especially when IP addresses or port information are included in the payload. For applications which cannot pass NAT cleanly, an ALG may still enable correct operation of the application. An ALG applies different, application specific operations on each message passing NAT in both directions. This operations may include modifying the payload of the message, open a port for incoming messages or what ever necessary to make the application successfully pass through NAT. SIP is an application which needs an ALG to pass a NAT [HHP+06, pp. 1650-1651]. This is because the protocol stores IP addresses and port information in the payload of the message. With such an ALG SIP messages can also be intercepted and changed for other purposes than to make the application work between two network realms.

Milanovic et al. [MSR+03] and Karpagavinayagam et al. [KSF07] describe how Lawful Interception (LI) can be accomplished in a VoIP infrastructure. Milanovic et al. [MSR+03] introduce a distributed system which shows how LI can be implemented independent to specific protocols. It is a distributed hierarchical system where different interception methods can be used at multiple points in the network to intercept VoIP messages and the media transport. For one interception method a module for a SIP proxy is proposed which alters the message during the call establishment so that all subsequent messages and media transmissions pass a device which can record the messages and the media. Also, a device is introduced which uses a network card in promiscuous mode to record the network traffic related to VoIP communication and media transmission. Karpagavinayagam et al. [KSF07] also introduce a system for LI but it is designed for the usage in an infrastructure which is based on SIP. The interception consists of three parts. The first is a module in the SIP server which changes the SDP data to redirect the media. The *RTP Mediator Module* is relaying the redirected packets and will thus act as a man in the middle. To collect finally the redirected media, a packet sniffer is used.

Acharya et al. [AWW07] describe a programmable message classification engine for SIP for the purpose of overload control. Overload control might be needed when suddenly the amount of messages unexpectedly increase to a point where the SIP server is overloaded. This may happen due to flash crowds, emergencies or denial-of-service attacks. In such situations it is important to prioritize messages and if necessary drop messages. The described implementation is not dependent on a specific SIP server as it is realized as a Linux kernel module. The code for classification and prioritization of the messages is inserted after the message is read from the network and before it is forwarded to the application. The advantage of a Linux module is, as already mentioned, the independency of the SIP server and a higher performance as the classifier does not have to change to the user mode while classifying the message and thus can completely be run in kernel mode.

3 Concept and Model

After the necessary aspects for this thesis are introduced in the previous chapters, this chapter will face the analysis and design for the implementation of the prototype. First, the requirements to the system are defined, then the related work is analyzed based on the gathered requirements. Subsequently, the partial problems of the systems are discussed and a final design is elaborated.

3.1 Analysis

The breakdown of the problem will be based on the problem description in Section 1. In combination with the insight from the state of the art analysis the requirements to the system are defined. The following vision describes briefly what the system is expected to accomplish:

> Develop a system which redirects all SIP messages sent within a network to a SCA where the message can be read and changed.

This vision expresses the central idea of the developed system. Based on the vision and the problems described in Section 1 the requirements are defined. The requirements are grouped into three categories:

- SIP requirements
- SCA requirements
- Non-functional requirements

The SIP related requirements are grouped by the *SIP requirements* and are abbreviated by *SR* whereas the SCA specific requirements are grouped by the *SCA requirements* and are abbreviated by *CR*. The non-functional requirements address the whole system and are grouped in the *non-functional requirements*, abbreviated by *NR*.

SIP requirements

SR1: SIP should be used as the VoIP protocol to initialize audio sessions. This standard should be used due to its popularity and extensibility, for more reasons to this decision see Section 2.1.

SR2: SIP messages are of interest which are sent to or from a SIP UA within the local SIP domain.

SR3: The system being developed should intercept SIP messages sent from a SIP UA before they arrive at the targeted UA. This includes request messages as well as response messages.

SR4: Incoming request and response messages of a SIP UA should be caught before they arrive at the UA.

SR5: All SIP messages sent and received by a SIP UA of the local SIP domain should be intercepted. This includes the initial message and all subsequent messages sent within as well as outside of transactions in terms of SIP.

SR6: All mentioned requirements also apply to SIP messages sent to and from an external SIP domain.

SR7: SIP messages sent within two UAs of the domain should only be intercepted once.

SR8: Intercepted SIP messages should be able to be changed freely. This includes the body as well as the header values.

SR9: The network location of he next SIP node should be able to be altered.

SR10: The system being developed should adhere to the SIP standard. Present SIP products and solutions should be able to be used in combination with the system being developed.

SR11: Unknown message types or header fields should be processed by the system as required by the standard.

SCA requirements

CR1: As a concrete SOA, SCA should be used. The reasons for this architecture are explained in Subsection 2.2.5.

CR2: All intercepted SIP messages should be handed over to a defined SCA service.

CR3: The defined SCA service should be able to read the entire SIP message. This includes, but is not restricted to, the header fields, the body if present, the message type and the network address of the SIP node where the SIP message is intended to be sent.

CR4: The SIP message should be able to be changed by the SCA service. Changes to the SIP message should not be restricted.

CR5: The altering of the SIP node to which the message is going to be forwarded should be possible.

CR6: The defined SCA service should be able to forward the SIP message to other services. These services should have the same interaction possibilities with the SIP message as the defined SCA service has. This includes reading and changing of the SIP message.

CR7: SCA services which process the SIP message should be able to be orchestrated.

CR8: When passing the SIP message from service A to service B, changes made by service A should be available in service B. At the same time changes made by service B to the SIP message should be able to be returned to service A after finishing the processing.

Non-functional requirements

NR1: A virtual machine, interpretor for scripts, just-in-time compiler or similar should not be used by the system being developed.

NR2: It should be possible to run multiple instances of the developed system in parallel, in different subnetworks or in sequence where one system is forwarding the SIP message to an other system. As the system does not have an influence on the services that run within SCA, it is the duty of the service developer to assure the service can be run on different systems at the same time.

NR3: Services developed to work with the SIP messages forwarded to SCA should be able to read and change the message in a simple way. In this way, the development of such services should be easy and fast. The services should not need to deal with details and organizational needs of SIP, like maintaining transaction state or similar.

NR4: The system being developed needs to provide a processing speed of 100 calls per second.

After the requirements to the system are defined, following related work introduced in Section 2.3 is analyzed if they can be used to build the required system:

- Application Layer Gateway
- Sniffing messages
- Message classification engine
- SIP Servlet

Due to the nature of an Application Layer Gateway (ALG) all sent SIP messages between two network realms are intercepted by the system. Also, the basic purpose of an ALG is to change SIP messages on the way to their destination. But as an ALG is part of the Network Address Translation (NAT) process, it only works between network realms. Thus, SIP messages sent within the same network realm need not to pass the NAT and thus not the ALG too. Although solutions for building an ALG for SIP exist [HHP+06] the usage of an ALG is always troublesome. For example an ALG is not aware of the existence of a client until the client sends a message which passes the ALG. Until then, the ALG cannot forward incoming SIP messages to the client, like session invitation. Also the usage of IP security bears troubles [SH99, p. 2] as it is intended to protect end to end communication. Security techniques protect the sending and destination address from modification, where modification is actually the fundamental functionality of a NAT and ALG. For these reasons an ALG is not applicable for the in the on-handed thesis examined problem.

The next examined related work is about sniffing messages directly from the network. This technique is used by the introduced LI in Section 2.3 and also at the health monitoring system. By sniffing messages from the network, all SIP messages sent in a network can be read by the system. The integration of this technique in an existing infrastructure is rather simple as it just has to be connected to the network and the SIP infrastructure does not have to be configured to explicitly include the network sniffer. As the sniffer is not an active node in the SIP infrastructure, no overhead or delay is added. The drawback of the sniffing is that it can only read messages and thus changing of SIP messages is not possible. The message sniffing also might require a redirection of SIP massages to the same network domain where the sniffing device is listening to, if the client is located in a different network domain. As sniffing of messages from the network is usually used to intrude the network, security mechanism are implemented in the network structure to suppress sniffing. Therefore, these security mechanism need to be disabled to allow network sniffing.

Acharya et al. [AWW07] introduce in their work a Linux kernel module which is developed to intercept and *classify SIP messages*. With the module it is not just possible to read a SIP message, it can also alter it. Due to the way how the Linux kernel module is implemented it is working independently to a specific SIP server. A drawback of this solution is, that the interception and the SIP server are running on the same machine and thus they share the resources. Further it is not guaranteed that a SIP proxy will receive all SIP messages, see example in [RSC+02, pp. 12-17]. Usually, a SIP proxy only receives the initial request and the subsequent response, the remaining messages of the dialog are sent directly between the partners. Further, the proposed module makes changes to the Linux kernel necessary. Therefore, these security mechanism need to be disabled to allow network sniffing.

A *SIP Servlet* [CK08] is a component which is running within an application server or a Servlet container. With a SIP Servlet, a variety of applications can be realized as the container of the SIP message provides a rich and flexible set on interfaces. Therefore, development of applications is easy but still flexible. With a Servlet, applications like an endpoint for SIP requests or a SIP server can be realized. Also a SIP proxy can be implemented and all received SIP messages can be read and changed. An advantage of a SIP Servelt is that it can utilize the rich features brought by application server, for example persistency or even a SCA runtime (e.g. IBM Websphere 7.0 [IBM]). The drawback of the SIP Servlet is the need of vast resources to run the application server or the Servlet container. Also the SIP Servlet is running within a virtual machine and cannot be run natively on the machine.

The described related work can fulfill basic requirements, but non of them can satisfy all requirements. The SIP Servlet comes closest to the specified requirements and can also offer different additional features, but fails in the essential requirement to be able to run on a system with limited resources. For this reason, a new approach is developed which meets all defined requirements.

3.2 Design

The defined requirements can be separated into two sub problems. The first one is primarily dealing solely in the area of SIP. This covers the ability to receive all SIP messages sent from the clients on the way to the destination, parse, change and finally forward them, see Subsection 3.2.1. The Subsection 3.2.2 is dealing with the physical architecture and the required settings in the environment. After solving the problem of intercepting SIP messages, the second subproblem deals in Subsection 3.2.3 with the procedure how to forward and receive the SIP message to and from SCA.

3.2.1 Session Initiation Protocol Message Interception

The subproblem of providing a system to intercept and change SIP messages is based on the requirements defined in the *SIP requirements* group.

A central requirement is to behave SIP conform. To comply with this, first a look at the SIP standard is taken if it already describes a kind of server on which the system could be build on. By using a kind of server which is described in the standard the system will meet the standard. In Subsection 2.1.2 the SIP servers introduced by the standard are briefly mentioned. Following servers are subsequently discussed in detail relating to a possible solution for intercepting SIP messages:

- Registrar
- Location Service
- Stateless Proxy
- Stateful Proxy
- Redirect Server
- B2BUA

Registrar

A registrar is a special kind of SIP server which handles *REGISTER* messages. It uses a location service to store the information of the *REGISTER* messages. A registrar may be co-located in a proxy server or be implemented as a stand-alone server. As the registrar only handles one type of SIP messages, it is not suitable for message interception.

Location Service

A location service stores the current location of all registered SIP user of the SIP domain. It is used by the registrar and proxy server to maintain the users state and resolves the destination for the routing respectively. As the location service is not directly involved in the processing of SIP messages, the location service is not suitable for SIP message interception.

Stateless Proxy

A SIP proxy routes SIP messages to their destination, this may be done via several other SIP proxies. The decision where the message is forwarded can be based on a rule set and can include information retrieved from a location service. The proxy can also be used to enforce certain policies to the users (e.g. if the user is allowed to make the call). A received message is being examined by the proxy and parts of the message can be rewritten before the message is forwarded.

A stateless proxy acts as a simple forwarding element without maintaining and being aware of the state of a SIP message [RSC+02, pp. 91-92, pp. 116-118]. Information about a message is discarded when it has been forwarded, also no SIP messages are generated by a stateless proxy server

The stateless proxy server is suitable to receive a SIP message on the way to the destination, when it is included in the routing path of the SIP message. This can be achieved by configuring an outgoing proxy server at the client or configuring other SIP proxy server. A typical example for a message flow involving a stateless proxy can be seen in Figure 3.1.

To establish a session an *INVITE* message is sent by the SIP User Agent (UA) of Alice to the stateless SIP proxy (1). The server calculates the next hop e.g. by querying a registrar server and then forwards the *INVITE* message to the SIP user Bob (2). The SIP client of Bob could confirm the receipt of the *INVITE* message with a *100 Trying* reply as originally displayed in the example of Trick and Weber [TW07, p. 195], but as provisional responses are only informational [RSC+02, pp. 182-183] a *180 Ringing* could be sent immediately by the SIP client back to the stateless server (3). This message is forwarded by the server upstream to the SIP client of Alice (4). When the phone is picked up by Bob, the *200 OK* message is sent to the stateless SIP proxy (5). The *200 OK* response is again forwarded to the SIP UA of Alice (6). As by default, only the request and responses as well as provisional responses are sent via a proxy server, the final *ACK* message is sent directly

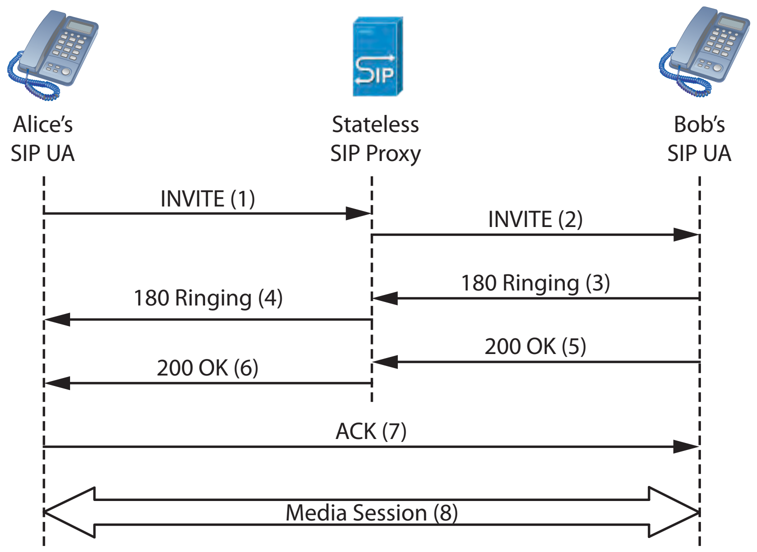

Figure 3.1: Invite message flow via a stateless proxy [TW07, p. 195]

from the SIP UA of Alice to the SIP UA of Bob (7). With the receipt of the *ACK* message the media session is established (8).

The stateless proxy server would be able to fulfill the first requirement in receiving at least the first message and the corresponding response of each session initiation. Future messages sent between the participants will usually bypass the proxy if no *Record-Route* header is added. The behavior of *Record-Route* header is described in Subsection 2.1.4. The stateless proxy needs to change parts of the message before the message can be forwarded. As no information of received messages are stored retransmitted messages cannot be recognized. Therefore, the stateless proxy has to assure that the exactly same changes to a message are applied to all retransmitted messages, too. The next SIP node is calculated by the proxy and thus can be influenced. But again as retransmitted messages cannot be recognized, the chosen destination has to be the same for each retransmitted message.

Stateful Proxy

In contrast to the stateless proxy, the stateful proxy maintains the transaction state of a incoming message. With this the stateful proxy is able to recognize re-transmitted messages, fork messages or generate a *100 Trying* response [RSC+02, pp. 92-93]. Otherwise the duties and overall behavior is similar to a stateless proxy as can be seen in the typical message flow in Figure 3.2.

By sending a *INVITE* request to the stateful SIP proxy (1) a session initialization is triggered. As the stateful SIP proxy maintains the state of transactions it can generate provisional responses. By immediately replying with *100 Trying* (2) the receipt of a message is confirmed and a retransmission by the client is avoided. Subsequently, the proxy

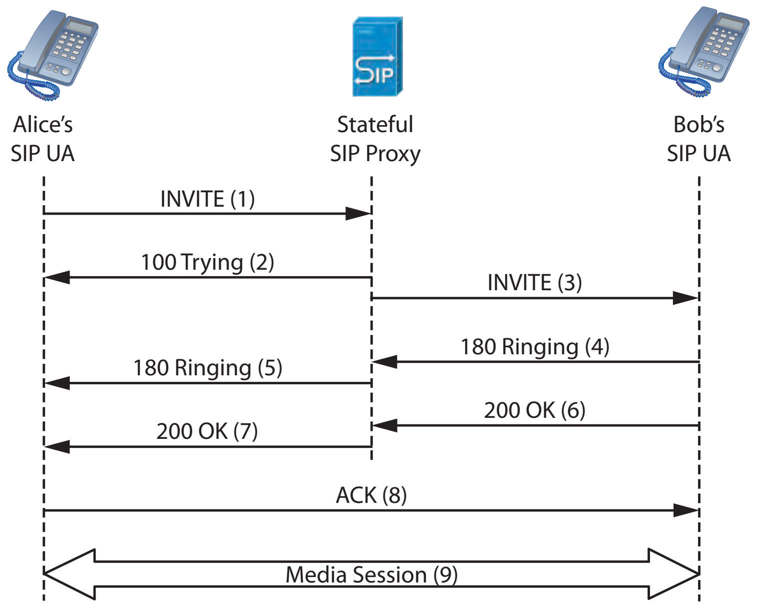

Figure 3.2: Invite message flow via a stateful proxy [RSC+02, p. 12]

calculates the next hop e.g. by using a registrar service and then forwards the *INVITE* request to the SIP client of Bob (3). The SIP UA of Bob is responding with a *180 Ringing* message to the proxy indicating the receipt of the request (4). The proxy will forward the *180 Ringing* response to the UA of Alice (5). When the call is accepted by Bob a *200 OK* message is sent to the proxy server by the SIP UA of Bob (6) which is again forwarded to the UA of Alice (7). The UA of Alice is confirming the receipt of the response by sending a *ACK* message directly to the UA of Bob (8). After this message the media session is established (9).

The first requirement to receive a SIP message sent by a SIP client can be fulfilled by the stateful proxy server. Also, responses to requests are received by the stateful proxy. To receive further SIP messages than the initial request and response message, the stateful proxy server has to add a *Record-Route* header, for details see Subsection 2.1.4. With the use of a *Record-Route* header all subsequent messages between the participants of the SIP dialog will be received also by the SIP proxy. Before a message can be forwarded, changes have to be applied to the received message. As transaction state is maintained by the proxy, retransmitted messages can be recognized. Also, the requirement to choose the SIP node where the message is going to be forwarded can be fulfilled by the stateful proxy, as the next node has to be calculated by the proxy.

Redirect Server

The redirect server is a simple server that responses each request with a *3xx Redirection Response*. This kind of server is used to redirect a request to the temporary address of a

SIP user e.g. when the called SIP user sets up a call forwarding. The information from the response is used by the requesting SIP client as new destination to send the initial request. In Figure 3.3 a typical message flow involving a redirect server is shown.

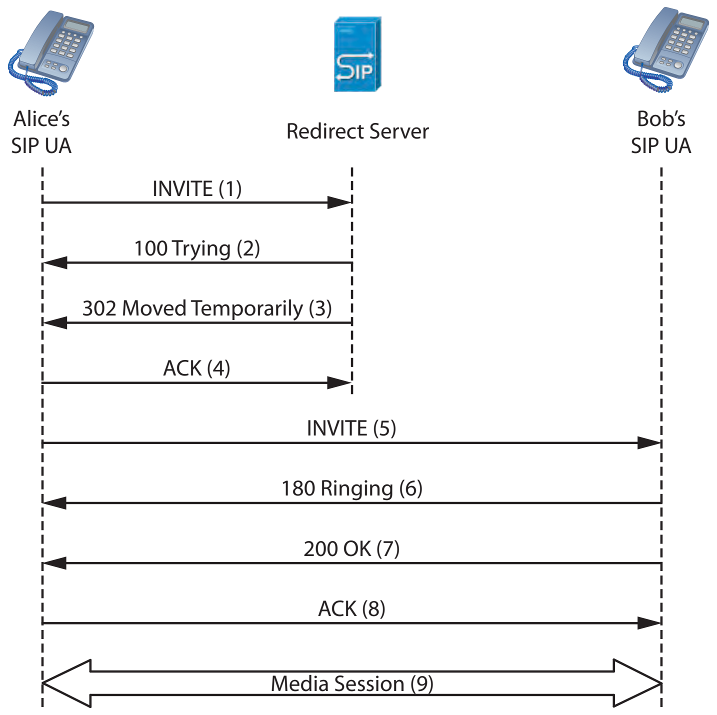

Figure 3.3: Invite message flow via a redirect server [JDS+03, p. 54]

For establishing a session using a redirect server the *INVITE* message from the SIP UA of Alice is received by the redirect server (1). Immediately, a *100 Trying* response is sent back as confirmation of delivery (2). The server looks up the current location of the user and forwards this information with a *302 Moved Temporarily* message to the UA of Alice (3). An *ACK* message is replied for confirmation by the SIP UA of Alice (4). With the information received from the redirect server, the SIP UA of Alice is sending an *INVITE* message directly to the UA of Bob (5). A *180 Ringing* message (6) followed by a *200 OK* message (7) when the phone is picked up is replied by the UA of Bob. The UA of Alice confirms with an *ACK* message (8) and a media session is established (9).

The redirect server fulfills the first requirement in receiving the request of a SIP client, but as the SIP client resends the request to the temporary location the response as well as any further SIP messages will not be received by the redirect server. The alternation of the SIP message is not possible, as a 3xx message is generated. Also, the next hop of the SIP message cannot be influenced but the final destination of the initial request can be changed with the redirect message.

B2BUA

The B2BUA is a special kind of SIP proxy server, which is mentioned in the SIP standard but not further standardized. From the outside the B2BUA behaves similar to a proxy, but internally it is different. It consists of a user agent server to receive requests and a user agent client to forward the message, see Figure 3.4.

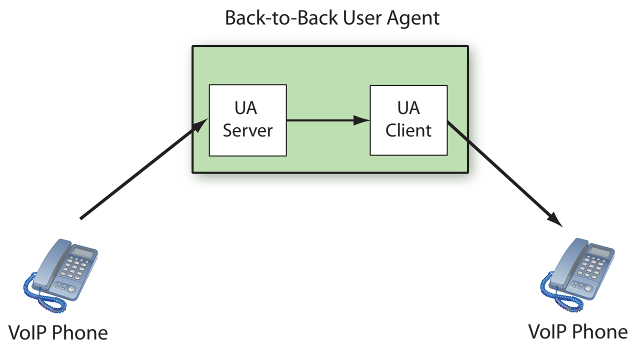

Figure 3.4: Structure of a B2BUA

Due to this the B2BUA is an active node in the message flow and will terminate SIP transactions between the calling and the called client. This makes it possible to change the messages freely and generate response as well as request messages. A typical message flow involving a B2BUA can be seen in Figure 3.5.

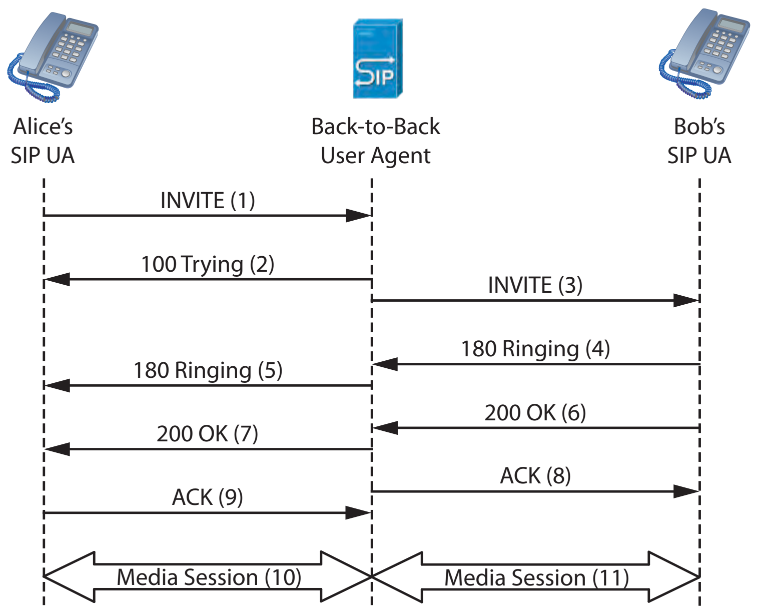

Figure 3.5: Invite message flow via a B2BUA

To establish a session between Alice and Bob, an *INVITE* message is sent by the SIP UA of Alice to the B2BUA (1). Like a stateful SIP proxy the B2BUA answers the request with a *100 Trying* message (2). Then an *INVITE* message is generated and sent to the SIP UA of Bob (3). The UA of Bob replies with a *180 Ringing* message (4). Again, the B2BUA generates a *180 Ringing* message and sends to the SIP UA of Alice (5). When the phone is picked up by Bob, the *200 OK* message is sent to the B2BUA (6). A *200 OK* message is generated by the B2BUA and sent to the UA of Alice (7). Now, in most cases, the UA of Alice would send a *ACK* message directly to the SIP UA of Bob. As the B2BUA is a active node, it will immediately send a *ACK* message to the UA of Bob (8) after the *200 OK* message is forwarded to the UA of Alice (7). The UA of Alice is sending a *ACK* message to the B2BUA (9). Finally a media session is established between the UA of Alice and the B2BUA (10) and a separate media session is established between the B2BUA and the UA of Bob (11).

Like a SIP proxy server, the B2BUA has to be included in the routing path of the SIP message. In this way, the first requirement is met to receive the first message. As the B2BUA is terminating the SIP transaction, all subsequent messages from the requester and the destination will be sent to the B2BUA. The messages sent by the B2BUA can be altered freely and the destination of the SIP message can be changed, too. The B2BUA itself is not standardized but it is acting standard conform. It is being used in production and can be found most likely in a working SIP infrastructure.

From the analyzed alternatives, the stateful proxy and the B2BUA look most promising. The stateful proxy lacks in receiving all messages of a session out of the box. But when adding a Record-Route header entry (see Subsection 2.1.4) this behavior can be achieved. Also, the standard defines rules which parts of a SIP message need to be changed and the procedure how e.g. the new values are determined [RSC+02, pp. 91-118]. Some freedom is granted to the proxy, for example it can decide to add a Record-Route header or additional header fields. Further changes in addition to those described by the standard might result in an unexpected or faulty behavior of other SIP nodes involved.

The B2BUA is more complicated to implement than a stateful proxy, but it supports most of the requirements already out of the box. Further, the B2BUA offers the possibility to generate requests which cannot be done by the stateful proxy. This is not an requirement, but it can be useful for further extensions.

A stateful proxy server can fulfill a wide range of requirements, but does not completely support the unrestricted changing of messages. The B2BUA fulfills lot of the requirements out of the box and offers additional capabilities, e.g. the generation of requests, which can be used in future extensions. Therefore, a B2BUA will be used to intercept and forward SIP messages.

3.2.2 Physical Architecture

After a suitable concept was chosen to intercept, process and forward a SIP message, the physical architecture and configuration of the participating SIP nodes are discussed in this section.

To ensure all messages are sent via the B2BUA, three scenarios on the initial request have to be be considered:

1. Requests sent within the network
2. Outgoing requests sent to a node outside of the network
3. Incoming requests received from outside of the network

The scenarios for sending requests within the network and for outgoing requests to a node outside of the local network can be grouped together and addressed at once by figuring out how outgoing requests can be redirected to the B2BUA. Usually, the SIP client tries to resolve the domain part of the SIP address and send the request directly there. When the SIP address is managed by a foreign domain, the request is going to be sent directly to the foreign SIP server like illustrated in Figure 3.6.

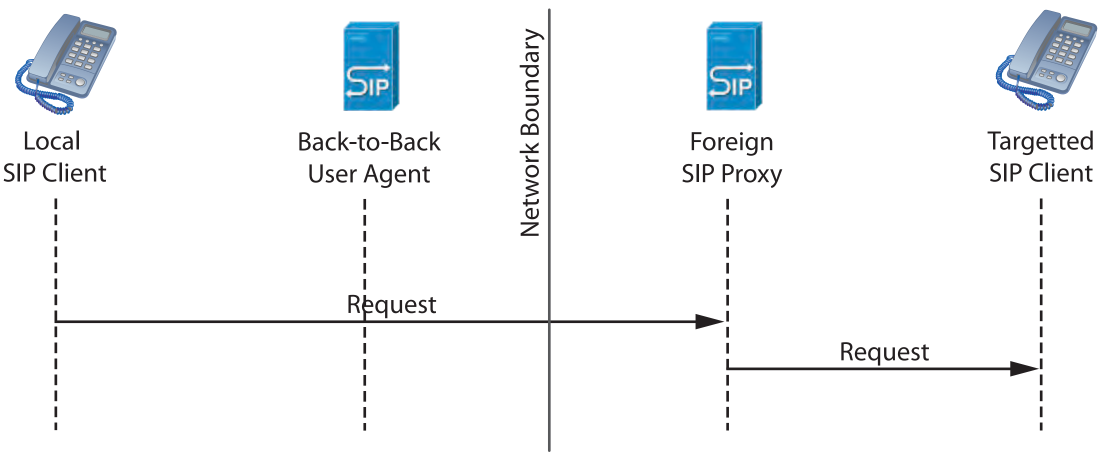

Figure 3.6: Request message without a configured outgoing proxy

To get the client to sent the SIP message to a proxy in the local domain instead of sending it directly to the foreign proxy server, an *outgoing proxy* has to be configured. In this way, all SIP messages are forwarded to the configured outgoing proxy instead of trying to find the destination on their own, see Figure 3.7.

The solution to receive all requests within the network as well all requests addressing SIP addresses in foreign network, is to configure an outgoing proxy at all clients of the local network, where the SIP messages should be intercepted. The configured address for the outgoing proxy should be the network address of the B2BUA.

To ensure that the B2BUA retrieves all incoming messages, it should be ensured that the B2BUA is contacted first from a foreign client or proxy. The foreign client or proxy server determines with the help of the Domain Name Service (DNS) where the SIP message should be sent.

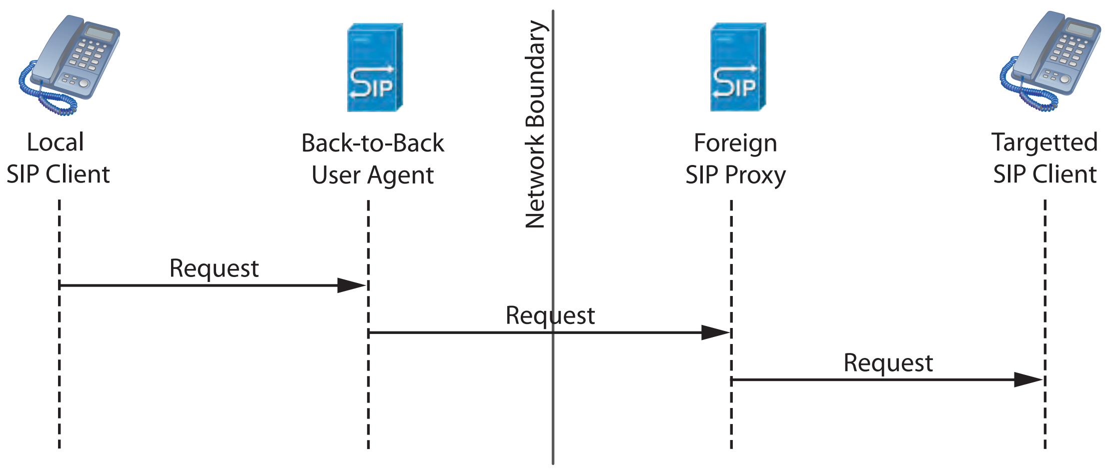

Figure 3.7: Request message with a configured outgoing proxy

The following DNS records are used to find the network location of a foreign SIP domain [RS02c, p. 2]:

- NAPTR [MD00]
- SRV [GVE00]

The first entry is the *NAPTR* field. This entry provides a mapping of the supported SIP protocol (e.g. SIP via UDP, SIP via TCP, SIPS via TCP, and other) to a SRV entry of the domain. The *SRV* entry contains a list of server including port information. In this way, first the desired protocol is selected in the NAPTR entry and then the list of server which support the desired protocol is retrieved.

To assure that all incoming messages are sent at first to the B2BUA, the NAPTR and SRV entries have to be configured to point to the B2BUA.

With this described measures the B2BUA will receive all incoming and outgoing SIP messages at first. Usually, the first message is accepted by a SIP proxy and using a location service the message would be forwarded to the corresponding user or in case of an outgoing SIP message the DNS system would be queried to find the SIP proxy server in charge. In many cases the location service is also implemented by the SIP proxy.

The implementation of these functionalities are not trivial but necessary for a working SIP infrastructure but they are out of the scope of this thesis. Therefore, an existing SIP proxy server is going to be added to facilitate the necessary functionality. In particular, the most important tasks are to process and store the user registration, maintain user state, routing of SIP messages to foreign domains, which includes the lookup of DNS entries as described above, and resolving of the temporary network address of local SIP users using a location server. Using a SIP proxy, the B2BUA can focus on the requirements and hand over these tasks to the SIP proxy server. Taking the additional SIP proxy server into account, the simplest physical architecture consist of the caller, the callee, the B2BUA and an existing SIP proxy server.

The inclusion of the SIP proxy server in the SIP message flow can be realized in different ways:

1. Placing the proxy server before the B2BUA
2. Placing the proxy server after the B2BUA
3. Intermediate proxy server

Placing the proxy server before the B2BUA

In the first option, shown in Figure 3.8, the client sends the SIP message first to the proxy server (1). Then, the proxy is processing the message and instead of sending the message to the next computed target, like the usual behavior of the proxy would be, the proxy sends the message to the B2BUA (2). The B2BUA can then send the SIP message to SCA and forward the processed message to the computed target of the SIP proxy (3).

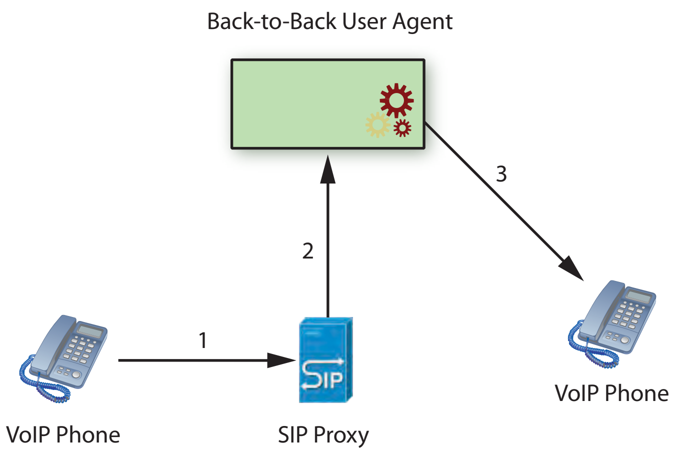

Figure 3.8: Routing option where SIP proxy is placed before the B2BUA (The numbers represent the chronological order)

With this configuration the B2BUA, and so the SIP message forwarded to SCA, will have all possible information which the proxy might add to the SIP message initially sent by the SIP client. Information the proxy adds could be the temporary address of a user, if the user state is maintained by the proxy, or information of the next hop, when the message is sent to a foreign domain. Also, a SIP message is sent to the B2BUA only once compared to the third option, which saves resources.

A drawback of this configuration is that the proxy server needs a special configuration to change the default behavior, to not send the message to the next computed SIP node than to send it in any case to the SIP component. This needs also a proxy server which allows the adjustment of the routing behavior. Especially for messages which would terminate at the proxy server, like REGISTER messages, pose a problem. With this configuration, the terminating messages have to be processed but then be forwarded to the B2BUA so that all messages of the client are forwarded to SCA. Those messages can also not be influence by the B2BUA as they have been already processed by the proxy server. An other drawback is that the B2BUA cannot control which proxy server will be used. This is because the outgoing proxy is configured at each SIP client and can usually only be changed by hand.

Placing the proxy server after the B2BUA

As shown in Figure 3.9, in the second option the B2BUA will receive the SIP message initially (1), send it to SCA and then forward it to the proxy server (2). The proxy server will process the message, calculate the next hop and send it to the target (3). This behavior of the proxy server is the same like usually SIP proxy server behave and therefore no special configuration of the routing is needed. In this configuration also terminating messages at the proxy server like REGISTER messages do not need a special treatment. A drawback of this configuration is that the information are missing which the proxy server adds to the message, like the next hop or the temporary address of a SIP user.

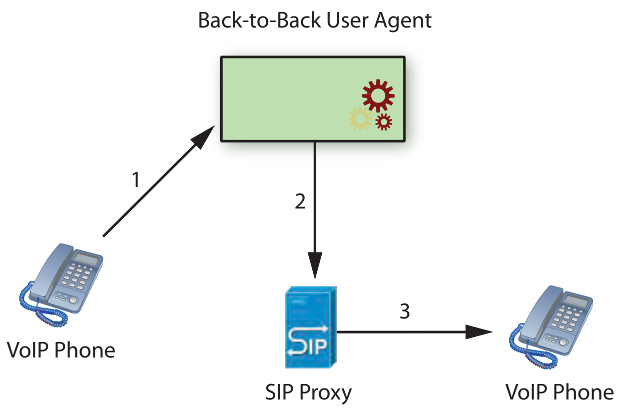

Figure 3.9: Routing option where SIP proxy is placed after the B2BUA (The numbers represent the chronological order)

Intermediate proxy server

The third option is the most complex and laborious one. It tries to consolidate the advantages of the other two options. The procedure of this option is shown in Figure 3.10. Like in the second configuration the SIP client first sends the SIP message to the B2BUA (1). This forwards the message to SCA. The returned message by SCA is forwarded to the SIP proxy (2) where the message is processed. The SIP proxy determines the next hop and like in the first option, the SIP proxy server does not forward the computed message directly to the next hop, it returns the message to the B2BUA (3). This forwards the processed SIP message a second time to SCA and forwards the result to the next hop (4) calculated by the SIP proxy server.

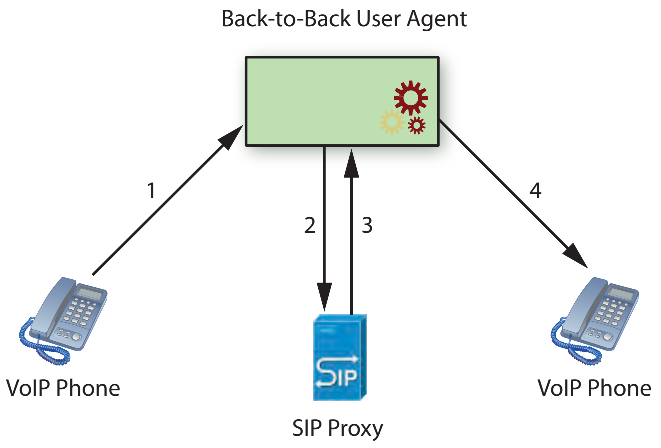

Figure 3.10: Routing option with intermediate proxy server (The numbers represent the chronological order)

This solution overcomes most shortcomings of the other options by delivering the SIP message twice to the B2BUA. In this way, the system can decide to which proxy server the message is forwarded and it receives also all altered and added information of the SIP proxy server.

This option shares the drawback of the specially configured SIP proxy server of the first option. But, as in this configuration the B2BUA can decide to which proxy server the message is forwarded, all proxy server have to be configured to send the message back to the B2BUA. As a result, the SIP proxy server can only be used with the B2BUA as they will send all received messages to the B2BUA after processing or a more complicated routing algorithm has to be installed. As an other drawback of this configuration can be seen, that the same message is going to be forwarded twice to SCA by the B2BUA. Although SCA receives more information in this way, all services run within SCA have to be aware of this behavior and consider it in the implementation. Additionally, the double reception of the same SIP message doubles the processing effort and reduce significantly the amount of messages that can be processed.

After considering all advantages and disadvantages of all options, the second configuration seems to be the best choice. It offers the flexibility in altering the message before it is sent to the SIP proxy server as well as the possibility of choosing the used proxy server. As the routing behavior of the proxy server does not need to be altered, like in the other options, the requirements to the used proxy server are less restrictive. Finally, the added and altered information by the proxy server does not contain crucial or important information and in consideration of the advantages the missed information is to be disregarded.

3.2.3 Integration in Service Component Architecture

The previous Subsections 3.2.1 and 3.2.2 focused on the SIP part of the examined problem. The second subproblem is dealing with the question how the intercepted SIP message can be forwarded to SCA and there efficiently processed by the services. Changes to the message should be reflected to the SIP message which is then sent to the next target by the system.

The *SCA requirements* group is covered in this section. Additionally, following non-functional requirements are addressed: NR2, NR3

Until now, the B2BUA was assumed to run as a own process. SCA offers different ways to interact with their services. External interfaces, like Web Services or Java Messaging Services, can be used. An other option is to use internal interfaces. For this option, the B2BUA needs to be implemented as a SCA service. The later one offers various advantages. In this way, the call of SCA services is very simple and can be done in a known way. Also in this way the B2BUA service can be easily reused and composed with other services. This is fully compliant to the idea of SOA. Figure 3.11 shows the B2BUA as a service of a SCA and the changes in the structures of the B2BUA to forward the messages to SCA services.

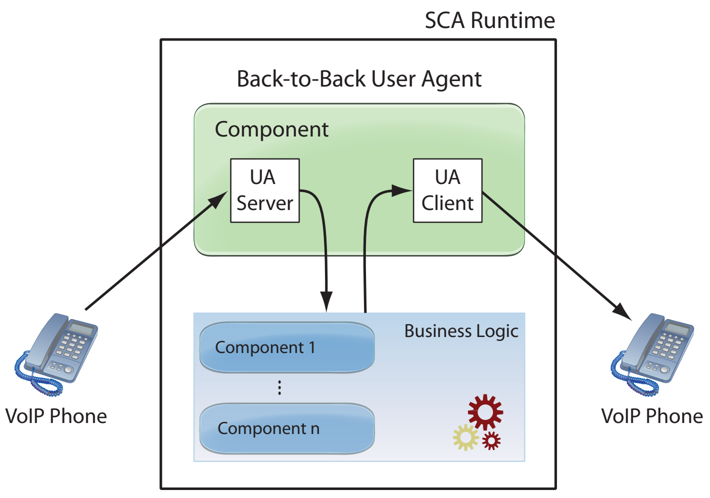

Figure 3.11: B2BUA implementation as a service

The SIP component containing the B2BUA needs to be initialized to be able to accept SIP messages. After a message is received by the user agent server of the SIP component, the message should be processed and prepared to be forwarded to the next SIP hop.

All necessary changes should be applied to the message before it is forwarded to the SCA services. In this way, the computed information like the next SIP node or changed header fields are available to the SCA services. This prepared message is forwarded to the SCA components. These can read and change the SIP message. Also, the SIP message can be forwarded to other SCA components and thus the SIP message can pass multiple components which all can change the message.

When the SIP message is returned to the SIP component it can be sent right away. To send the fully processed SIP message to the SCA services also prevents possible lost changes made by SCA components, if the B2BUA would compute and apply the necessary changes to the SIP message after it was returned by the SCA components.

To forward a SIP message to other SCA services, the message has to be serialized so it can be passed via method parameters. Different methods where developed how the SIP message can be serialized:

1. Serialize the whole SIP message as text
2. Pass each important header as a own method parameter
3. As a complex SDO object

In the first method, the SIP message is serialized by the SIP component to a text stream and stored in a string. This string has the same format as the SIP message sent via the network. This has the advantage, that the message can be read from the network and directly forwarded to the SCA services. Also, existing SIP parser can be used to parse the message and create a response, see Figure 3.12.

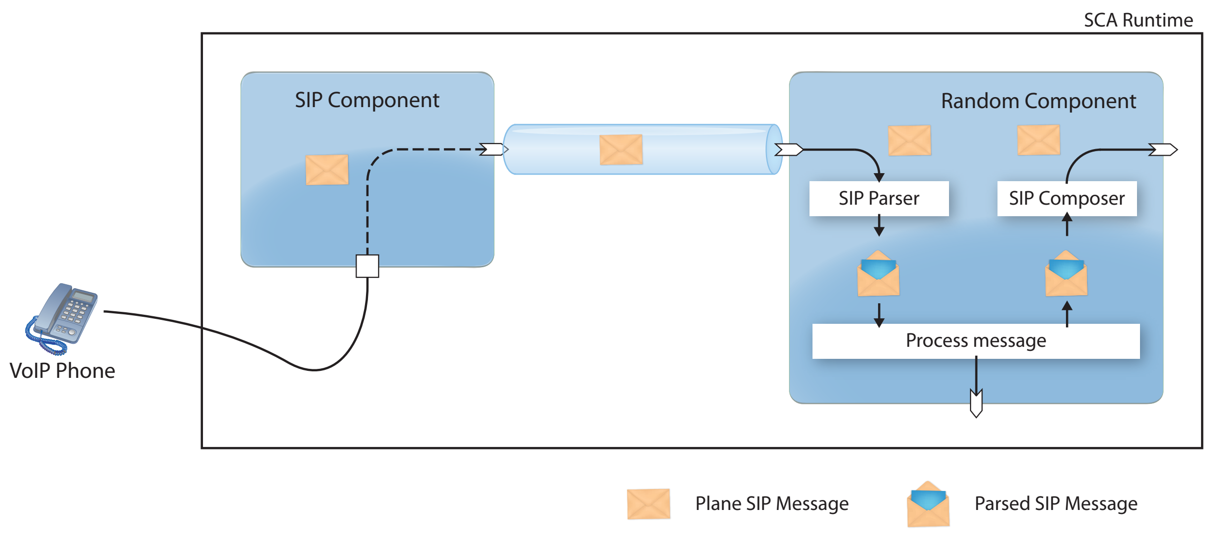

Figure 3.12: Serializing option: Whole SIP message as text

As the message is transfered as it is received from the network, all information and headers of the message will be preserved when it is transfered to the SCA services. In this way, no changes need to be done at the SIP component when additional headers are needed by the services or if a SIP extension should be supported. The needed information can be extracted directly by the services. An other advantage of this method is that the SIP message could be read from the network by the SIP component and directly forwarded to the SCA services without parsing it. This would make the implementation of the SCA component simple and at the same time flexible to changes. But as the message has to be changed by the SIP component before it can be forwarded, the message has to be parsed anyway.

A drawback of this method is that each service has to parse the SIP message on their own, as can be seen at the *Random Component* in Figure 3.12. This is not a trivial task and will usually need tool support. As well the support for particular SIP extensions need to be implemented in all services. This adds an additional layer of complexity to each service, which is not the idea of SOA.

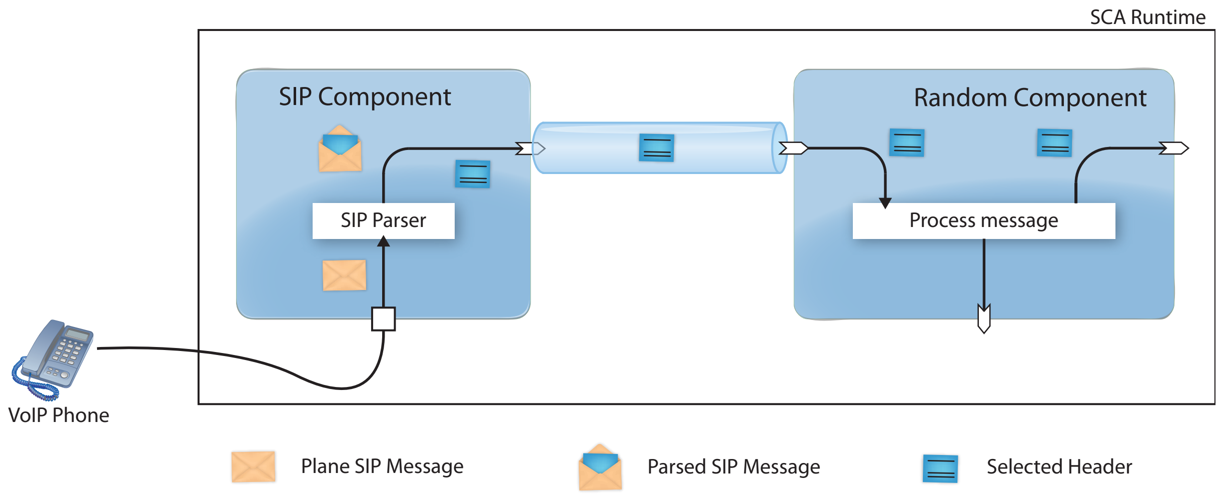

Figure 3.13: Serializing option: Pass each important header as a own method parameter

As the SIP message has to be parsed by the B2BUA anyway, an other method is developed where the most important header fields are passed as method parameter to the SCA services, see Figure 3.13. In this way, the services receive the header messages in a clean and structured way.

The SIP standard defines 44 different header fields where 17 out of them are mandatory in at least one message type [RSC$^+$02, pp. 159-163]. Header fields can also occur more than once in a SIP message, for example a VIA header field has to be added to the SIP message at each hop, see explanation in Subsection 2.1.4. Due to the amount of different SIP header and the possibility for multiple header values, the approach to pass SIP header as separate method parameter is not a promising approach.

By continuing the thought of the second idea, this third method was developed. In the third method, see Figure 3.14, the SIP message itself is stored in a complex structure.

This structure is passed to other services via a method parameter. When a SIP message is received by the SIP component, the message is parsed, needed changes are applied and finally all values of the SIP message are stored in the structure. In this structure, each header value and the content is stored in an own field. This structure is passed to SCA services which can read and change the values of the SIP message easily by simply accessing the fields in the structure. The altered structure can then be passed to other SCA services before it is returned to the SIP component. At the end of the processing the SIP component receives the structure from the SCA services and composes a SIP message based on the values of the structure.

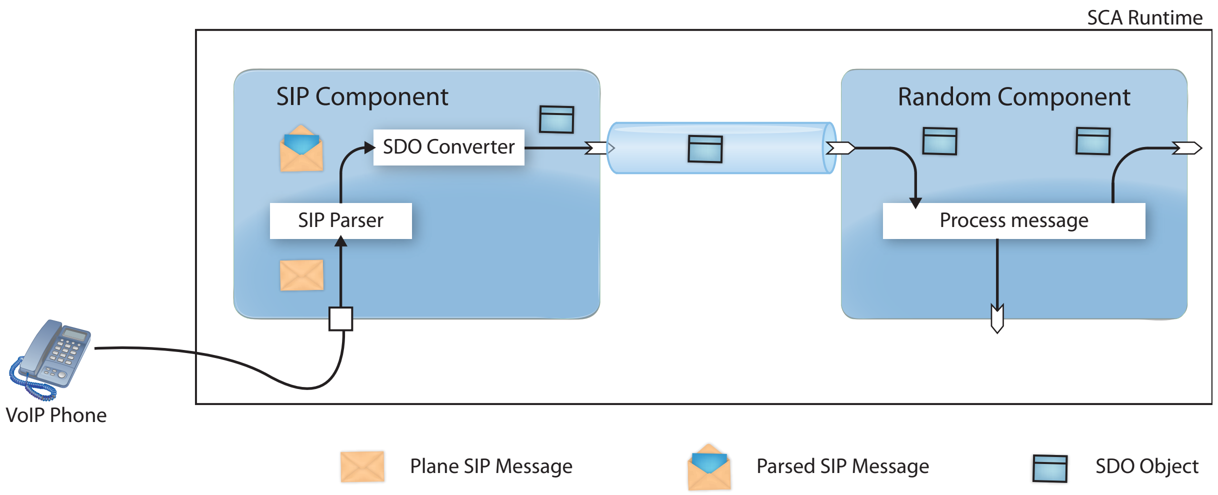

Figure 3.14: Serializing option: As a complex SDO object

SCA is designed to work with different programming languages. This requires to define for each component a language natural interface. Where primitive types like integer and string can be mapped in each language, each programming language has its own way to handle complex structures. Due to this a direct mapping of a complex structure, like a class to a complex structure in an other programming language, is not possible. Therefore, the developer of SCA introduced with SDO [SDO] a standard to handle complex structures. With SDO it is possible to pass complex structures between services.

With this solution the SIP component and the structure of the SDO object has to be changed each time a new header field should be supported. But due to the structure and the way how SDO objects are implemented, changes in the SDO structure does not necessarily mean the SCA services consuming the SDO object need to be changed, too. This is only necessary if changed or removed fields in the SDO structure are used by the service.

For the implementation developed within this thesis, the decision is made to not implement all existing SIP header fields. The first reason is simply to reduce the implementation effort and the second reason is to reduce processing and network resource consumption to create the SDO object.

Header field	ACK	BYE	CAN	INV	OPT	REG
Accept		r		r	r	r
Accept-Encoding		r		r	r	r
Accept-Language		r		r	r	r
Allow		r		r	r	r
Call-ID	r	r	r	r	r	r
Contact				r		
Content-Length	r	r	r	r	r	r
Content-Type	r	r		r	r	r
CSeq	r	r	r	r	r	r
From	r	r	r	r	r	r
Max-Forwards	r	r	r	r	r	r
Min-Expires						r
(Proxy-Authenticate)		r		r	r	r
Require		r		r	r	r
Route	r	r	r	r	r	r
Supported				i	i	
To	r	r	r	r	r	r
Unsupported		r		r	r	r
Via	r	r	r	r	r	r
(WWW-Authenticate)		r		r	r	r

Table 3.1: Listing of header fields which are required in at least one situation, distinct by method (r: required, i: header field should be included, but not mandatory, ACK: ACKNOWLEDGMENT, CAN: CANCEL, INV: INVITE, OPT: OPTIONS, REG: REGISTER)

The decision on which fields are going to be used is made on the importance of the header fields. Therefore, all header fields which are not defined as optional by the SIP standard [RSC+02, pp. 159-163] are included in the SDO object. Table 3.1 contains the list of header fields which are transfered to the SCA services. The table also contains information about the header field in relation to SIP methods. A field is seen as required by a method if there is at least one situation where the field is mandatory.

Despite the header fields *Proxy-Authenticate* and *WWW-Authenticate* are not optional, they are not included in the structure as they are used for authentication. Authentication is not in the scope of the prototype, therefore these fields are excluded.

Additionally to those 18 header fields, the message type, the address of the calculated next SIP node and the body of the message are included in the SDO object.

3.3 Proposed Solution

All parts of the problem have been discussed earlier in this chapter, different options are proposed and finally the most promising is chosen to be used. In the following the proposed solutions of the discussed subproblems are put together to formed an overall design for the addressed problem.

3.3.1 Infrastructure

It is assumed that the SIP domain is run within a company or organization which is also responsible for the maintenance of the VoIP system. This VoIP system is supposed to be used mainly for internal calls but the reception of voice calls from outside of the domain should be possible and considered. This does not necessarily mean that a connection to the PSTN network or the Internet with public SIP provider will be present. But it is assumed that the network can be organized in multiple SIP domains.

SIP clients using the SIP domain are expected to be standard conform. A differentiation between softphone and hardware phone clients is not made. The only requirements to the clients, next to the compliance with the standard, is the ability to configure an outgoing SIP proxy.

For the system to work, one SCA server running the SIP component and at least one SIP proxy server is necessary. It is assumed that the proxy server can handle all needed SIP functionality of a domain like registrar or location server. If the proxy cannot cover all required SIP functions, additional server have to be added to provide the missing functionality. Like defined in the requirements, more SCA server running a SIP component can be installed as well as multiple SIP server, if needed. A network topology with the essential SIP components the system needs is displayed in Figure 3.15.

3.3.2 Message Processing

To describe the exact interplay of the solutions introduced in Section 3.2 exemplary a SIP message is observed while being processed by the system, visualized in Figure 3.16. Examined is a SIP message sent by a SIP client of the local domain to an other SIP client of the same local domain.

As mentioned in Subsection 3.2.2, an outgoing proxy server has to be configured at each SIP client of the domain. As the outgoing proxy server, the network address of the server running the SIP component in the SCA runtime has to be used. Due to this configuration every initial request message sent by the clients of the domain will be first sent to the server running the SIP/SCA component. This is visualized in the Figure 3.16 with the message named *request* (1).

The addressed server runs a SCA runtime where the SIP/SCA component is running as a service. The SIP component listens directly on the network for incoming SIP messages

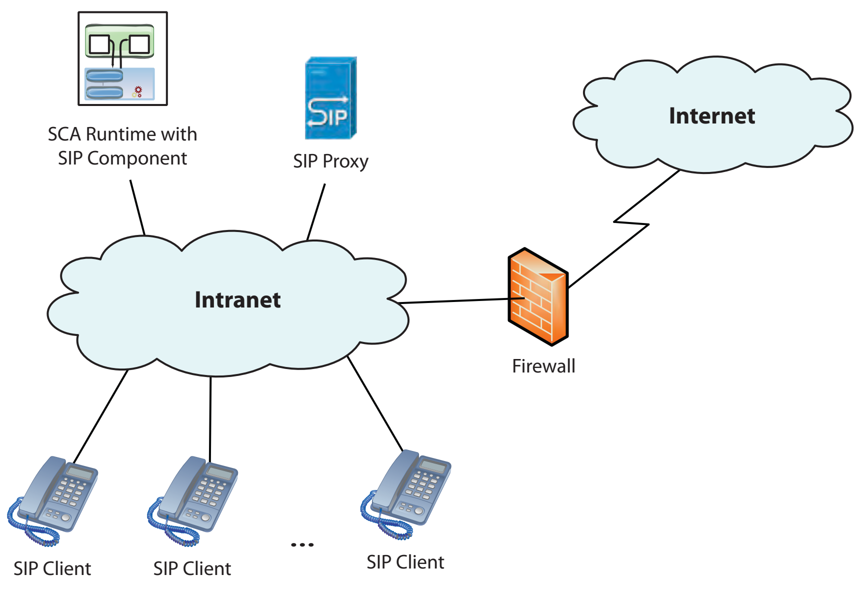

Figure 3.15: Required components by the system

and does not use any functionality a SCA environment might offer for the reception of the SIP messages from the network. The SIP/SCA component is separated into two subcomponents:

- SIP/SCA proxy
- SIP/SCA wrapper

The *SIP/SCA proxy* is handling the SIP responsibilities and the *SIP/SCA wrapper* is responsible for the SCA specific functionality.

The request sent by the SIP client is accepted within the SIP/SCA proxy subcomponent. This subcomponent implements the B2BUA which handles the processing of SIP messages as required by the SIP standard. The processing includes the needed changes of the message and the calculation of the SIP node to which the message should be forwarded (2). After finishing all processing, the SIP message ready for delivery is handed to the SIP/SCA wrapper subcomponent (3). This subcomponent handles the SCA specific part. It transforms the SIP message into a SDO object. This SDO object is being forwarded to the reference defined by the SIP/SCA component (4).

The SCA runtime invokes the service wired up with the reference of the SIP/SCA component. This service can use the SDO object containing the values of the SIP message and process it. Other services can be used to support the service (5, 6). The SDO object can also be passed to other services where it can also be altered. At the end of the processing, the changed SDO object is returned to the SIP/SCA component (7).

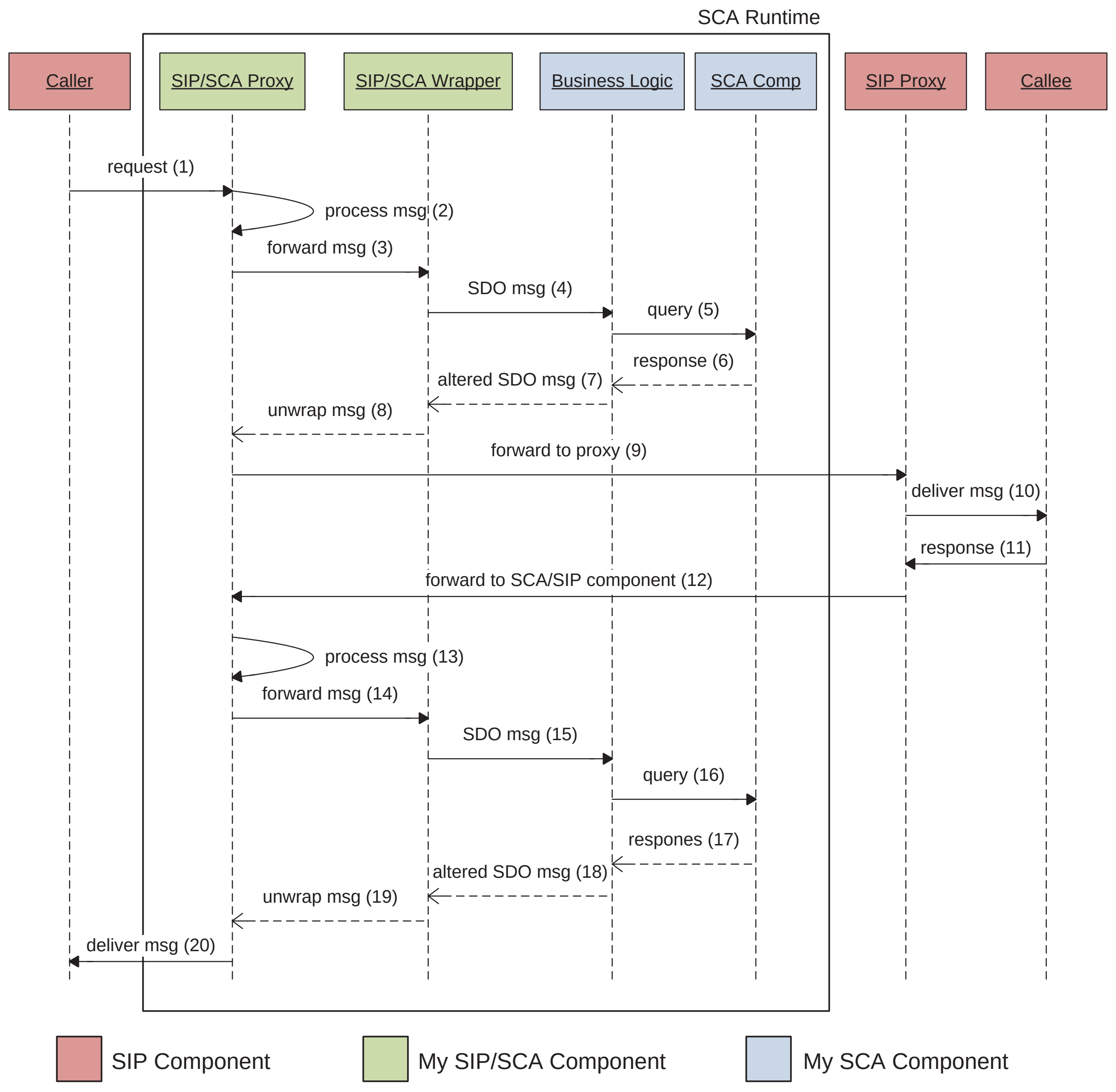

Figure 3.16: SIP message flow through the system

The returned SDO object is received by the SIP/SCA wrapper which unpacks the SDO objects. The changes to the SDO object made by the SCA services are applied to the SIP message and the altered SIP message is returned to the SIP/SCA proxy subcomponent (8). It is forwarding the message processed by the SCA services to the address defined in the SIP message (9). If the temporary address of the calling SIP user could not be resolved, the message has to be forwarded to a proxy server capable of resolving the temporary address of the calling SIP user or to a SIP server which is able to find such a proxy server. SCA services changing the forwarding address of the message have to be aware of this. Usually, the destination of the message will be a SIP proxy server of the domain.

The SIP proxy server processes the message and the decision is made to which SIP node the message is going to be forwarded. For the calculation the location service might be facilitated to find the temporary address of the called SIP client. Finally the SIP message is forwarded by the SIP proxy and is accepted by the targeted SIP client (10).

The response message is traveling the same way backwards through the system like a ordinary SIP response. The SIP client sends the response message to the SIP proxy (11) and the proxy forwards the message to the SIP/SCA component (12). There the SIP/SCA proxy subcomponent receives and processes the message (13). As all messages are forwarded to SCA, also this response is forwarded to the SIP/SCA wrapper subcomponent (14), same as initially the request message. The response message is converted to a SDO object and is forwarded to the wired SCA service (15).

The SCA service is the same as which is used at the request. Thus, the services connected to the SIP/SCA component need to be aware which kind of SIP message they receive. The service processes the SDO object and queries an other SCA service (16, 17). After the service is finished it returns the altered SDO object again to the SIP/SCA wrapper subcomponent (18).

The SDO object is converted to a SIP message and passed to the SIP/SCA proxy subcomponent (19) which is forwarding the message to the calling SIP client (20). As also the reply can be freely changed by the SCA services, the SIP message does not need to be delivered to the calling SIP client. The SIP/SCA proxy will forward the message to the address defined as the forwarding address. But not delivering the response message to the sending client would make the client to retransmit the request message. When the SCA services change the SDO object, they must be aware to comply to the SIP standard so that the participating SIP components like proxy servers and SIP clients work properly.

As can be seen in Figure 3.16, the SIP message is spending a significant amount of time within SCA. So the SIP message is not just being processed by the B2BUA residing in the SIP/SCA component, also the message is passed to SCA services which on their own take some time to execute. Because the message is being processed by the B2BUA and the SCA services, it is expected that the performance of the system is going to be worse than if an usual proxy would be used in place of the SIP/SCA component. The final performance also depends on the time each called SCA services takes to process the message. The expected drawback in performance is accepted as this system offers various advantages compared to an ordinary SIP system.

4 Proof of Concept

Based on the analysis and the chosen design a prototype is implemented. The libraries and applications used by the prototype are discussed. To evaluate the prototype different tests are made. The performance of the prototype is measured by conducting load tests. The evaluation compares the performance of the prototype with conventional SIP systems. Subsequently, the results of the tests are discussed as well as the prototype in general.

4.1 Implementation

In Section 3 different components are identified to build the system. The implementations of these required components need to fulfill the requirements raised at the beginning of the previous section. Subsequently, the implementation of the required components for the prototype is discussed.

The aim of the prototype is to prove the functionality. It should show that the designed system can be implemented and provide all functionalities as required. By implementing sample applications the expected advantages for application development should be proven. The prototype is examined for optimizations, but building a high-performance system is not the primary target of the work.

As defined in Section 3.1, the application should run natively on the system and thus should not use interpreters or virtual machines. Due to this, the system should be developed in the computer language C/C++. As it is planed to use an embedded Linux, the Ubuntu Linux [Ubu] distribution is chosen as the development platform.

4.1.1 Used Components by the Prototype

For each required component a suitable implementation is needed. Existing products can be used for some components like the additional SIP proxy. Also for the components which need to be implemented, libraries are used to ease and accelerate the development. In the following, different alternatives to each component are described as well as the reasoning for selecting one of these components.

Service Component Architecture Implementation

As described in Subsection 2.2.5, SCA is a concrete implementation of SOA. For this concrete implementation, different realizations are available. A lot of those realizations base on Java but there are also some developed with C/C++. The implementations of SCA can be categorized into stand-alone solutions and integrated solutions. Stand-alone solutions are designed to run on its own and does not need any other components to work. These solutions suit best for fields of applications where only the SCA environment is needed. Integrated solutions have SCA integrated into other products or components. In this way, SCA can be provided within a Java application server or similar. This is useful in situations where next to SCA different other technologies and products are used. Additionally the implementations can be grouped into commercial implementations like *IBM WebSphere Application Server V7.0 Feature Pack for SCA* [IBM] or *Oracle SOA Suite* [Ora] and open source implementations like *Fabic3* [Fab], *SCOrWare* [SCO] or *Apache Tuscany* [Tus].

Apache Tuscany was chosen for the prototype as it is a stand-alone implementation and it offers two different versions. One version is implemented in Java and the other one in C++, referred as native. For the prototype the native implementation of Tuscany is taken. Apache Tuscany is open source and licensed under the Apache License in the version 2.0 [Apa]. The native version supports SCA components which are implemented either in C++ or Python.

Session Initiation Protocol Library

SIP is a quite complex protocol and every aspect is defined to the least detail. This reaches from the overall structure of messages down to the possible options of a parameter, the exact spelling and the meaning of it. SIP also offers some freedom in generating SIP messages like the order of the SIP header is free or the implementation can decide if multiple header values are generated as multiple header each containing one value or as a list of header values in a single header. To correctly implement the SIP specification all aspects need to be implemented according to the defined rules and thus the correct parsing, interpretation and generation of a SIP message is laborious and error prone. Therefore, a library should be used to create and parse SIP messages and so support the developer in obeying the rules defined by the SIP standard.

Requirements to the library are as follows:

- The library has to be implemented in C/C++.
- As the target operating system will be Linux it has to be a library for that operating system.
- It has to be reliable and proven. Thus, it should already be used in applications which are in use. It is also desirable that the development is still in progress and thus the library will be maintained in case of found errors or bugs.

- The processing of SIP messages by a SIP server is different than by a SIP client, therefore server functionality should also be supported by the library.
- Finally, due to the limited resources a library with a minor need of resources should be preferred.

The decision for the library used in this work is made between PJSIP [PJSa] and eXosip2 [eXoa].

eXosip2 is a library which offers a high-level Application Programming Interfaces (API) for SIP. This library is build using osip [OSI], a SIP library from the same author which offers a low-level API. With the high-level API of eXosip2 common SIP interactions can be realized in a simple way, but it does not allow to have an influence on the exact message creation. With the second low-level API influence to all aspects of the SIP communication is possible. With this API also server can be realized. The eXosip2 library can be licensed with the GNU's Not Unix! (GNU) General Public License (GPL) version 2 [GPL] or with a commercial license. Dual-licensing is preferred for this work as it provides the freedom to decide in a later stage to either use the GNU GPL and publish the source code or to purchase a commercial license where the source code does not need to be published.

The eXosip2 library does not provide a special interface for creating servers, although a server can be built by the low-level API. But using this approach all aspects of the SIP server need to be implemented by the developer and thus, no support for obeying the SIP standard is given. Additionally, the library does not assure that the received SIP message is completely valid according to the SIP standard [eXob]. Some check still have to be done by the developer.

The second library, PJSIP, is a rich SIP library which has a small footprint but offers functionality for SIP UAs and SIP servers. It is a portable library so it is available for different operating systems as well as various mobile systems. The library is offering different APIs representing different level of abstraction, as displayed in Figure 4.1

The API offering the highest abstraction is *PJSUA-LIB* and full functional SIP clients can be build easily with some method calls. Different APIs with decreasing abstraction (*PJSIP-UA*, *PJSIP-SIMPLE*, *PJSIP*, *PJLIB-UTIL*) are also available. The *PJLIB* API represents the base framework on which all other APIs of this library base. Support for media negotiation, media stack (*PJMEDIA*) and different codecs (*PJMEDIA-CODEC*) is also offered by the library.

A rich documentation is offered by this library and also server functionality via APIs on a higher level is supported. For this purpose, mainly the *PJSIP* API can be used, but also some functionality from the *PJLIB-UTIL* and *PJLIB* API is needed. The usage of higher level APIs does not require the developer to do the error prone implementation of the SIP server behavior according to the SIP standard. Additionally, the library is available via a GPL license or optional via a commercial license.

Due to the support of server functionality via an API and the better validation of the SIP messages PJSIP was selected to be used.

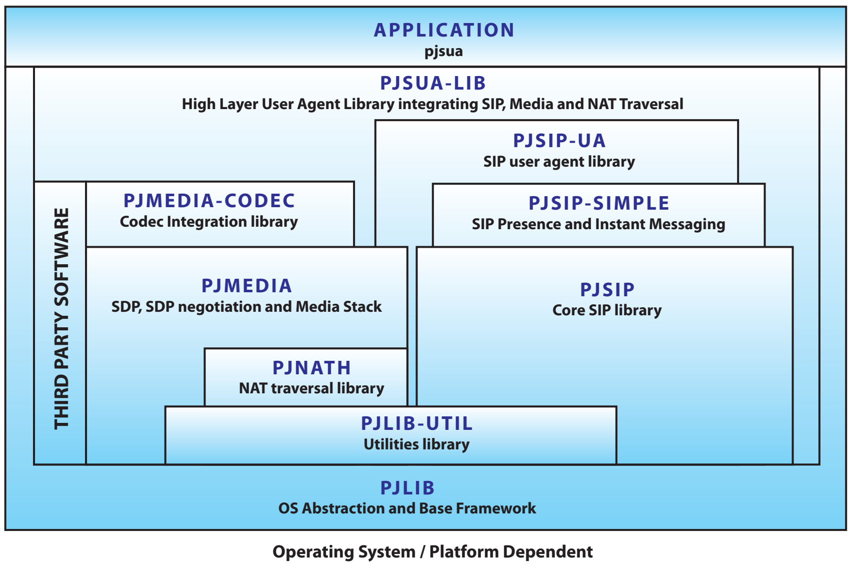
APPLICATION
pjsua
PJSUA-LIB
High Layer User Agent Library integrating SIP, Media and NAT Traversal
PJSIP-UA
SIP user agent library
THIRD PARTY SOFTWARE
PJMEDIA-CODEC
Codec Integration library
PJSIP-SIMPLE
SIP Presence and Instant Messaging
PJMEDIA
SDP, SDP negotiation and Media Stack
PJSIP
Core SIP library
PJNATH
NAT traversal library
PJLIB-UTIL
Utilities library
PJLIB
OS Abstraction and Base Framework
Operating System / Platform Dependent

Figure 4.1: Architecture of PJSIP with different APIs [PJSb]

Session Initiation Protocol Proxy Server

As described in Subsection 3.2.2, an additional SIP proxy server is required for the system to handle the registrations of the SIP user and to find routes for the SIP messages.

For the additional SIP proxy server, *Kamailio* [Kam] will be used. Kamailio is a well-cited open source SIP proxy for Linux/Unix platforms. It is a high-performance SIP proxy with a lot of features. The server can act as a registrar and supports dynamic routing. Finally, the proxy is implemented in C and thus complies to the requirements to the system.

4.1.2 Structure

The structure of the implementation is rather simple as the used SIP library took off a lot of development effort. The class diagram of the prototype can be seen in Figure 4.2, where the classes with green background belong to the developed system. The remaining classes are stubs which are necessary to run the system. Those stubs will be replaced by an own implementation of the user of the system.

The *SipProxyClientApp* is the application which uses the developed SIP/SCA component. As the server part of the component can not be started and stopped directly by the framework, the *SipProxyClientApp* is needed to take over those tasks. This application is only a stub and the accomplished tasks would be implemented by the user of the system. For starting and stopping the server a method *startServer* respectively *stopServer* is provided by the interface. This small application simply starts the server when the application is loaded and stops the server after a random user input. This simple behavior is sufficient

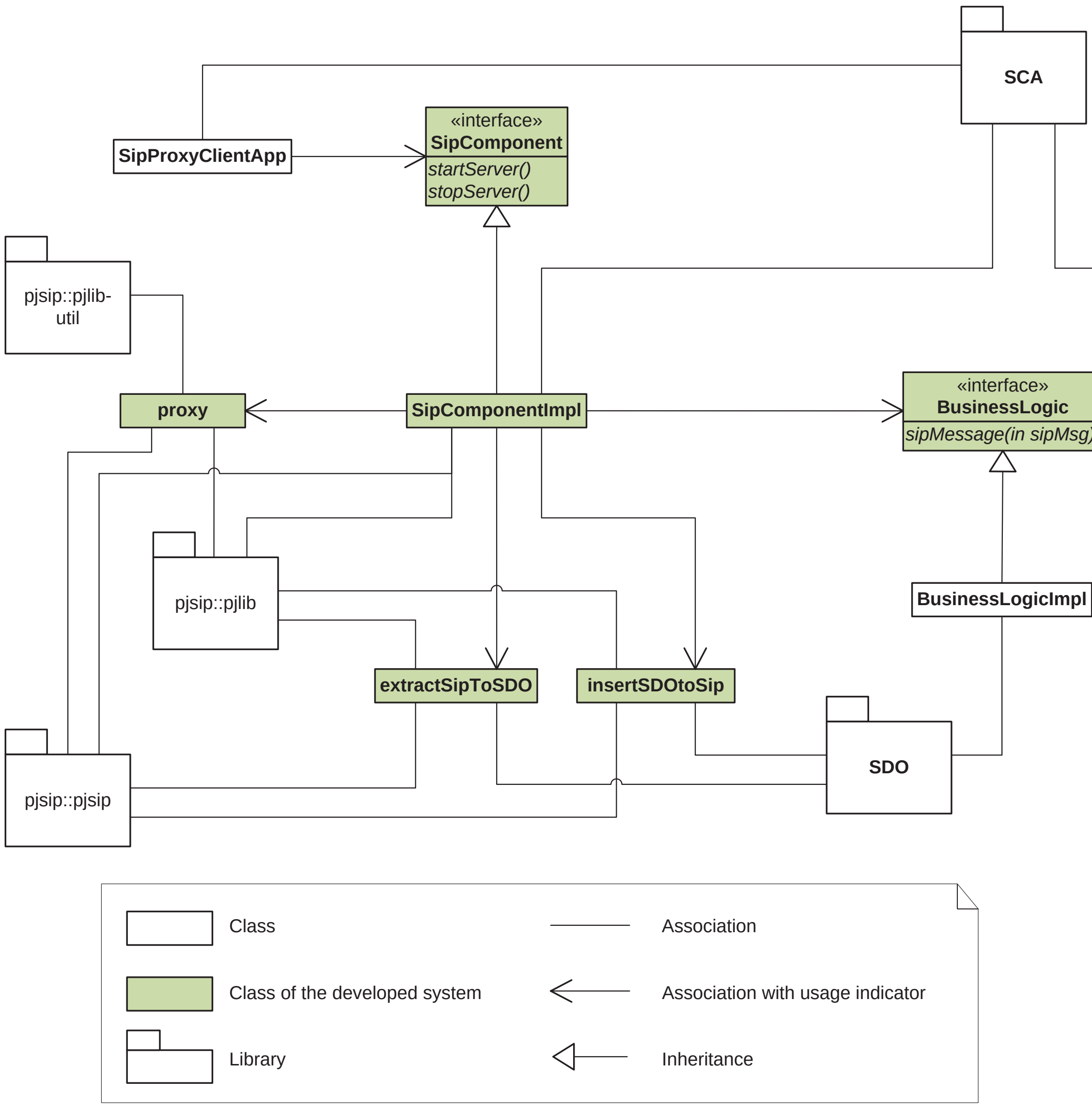

Figure 4.2: Class diagram of the implementation

for the prototype. To retrieve an instance of the *SipComponent* and invoke the start and stop operations the *SCA* library is needed.

To pass the SIP message to other SCA components, subsequently referred as *business logic*, the SIP message has to be converted to an SDO. As the PJSIP library already provides a clean data structure for the SIP message, this structure is taken as a pattern to define the structure of SDO. The SDO definition following the design decisions in Subsection 3.2.3 can be found in the Appendix: SDO Definition.

SipComponent is the interface which is exposed to the framework and through which the component can be used. This interface is defined by the developed system and is implemented by *SipComponentImpl*. This component is the core of the system. It implements the functionality of the B2BUA and communicates with the business logic. The *SipComponentImpl* is realized as a SCA component, in Listing 4.1 the composite definition is shown.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<componentType xmlns="http://www.osoa.org/xmlns/sca/1.0">
  <service name="SipComponentService">
    <interface.cpp header="SipComponent.h"/>
  </service>

  <reference name="BusinessLogicService">
    <interface.cpp header="BusinessLogic.h"/>
  </reference>
</componentType>
```

Listing 4.1: SCA component definition of the developed *SipComponent*

The component definition offers a service called *SipComponentService*. The interface of this service is defined as a C++ header file and the name of the file is called *SipComponent.h*. This referenced interface is the *SipComponent* interface in the class diagram. After the service, a reference is defined with the name *BusinessLogicService*. This reference is used to forward the SIP message to an other SCA component. The interface is again defined as a C++ header file and can be found in the *BusinessLogic.h* file. In the class diagram this interface is named *BusinessLogic*.

The PJSIP library provides different hooks where applications can register methods. These registered methods will be executed on the occurrence of certain events. This functionality is used by the *SipComponent* to process the SIP message, forward it to the business logic and finally send it. Following hooks are used:

- on_rx_request
- on_rx_response
- on_tsx_state

The first hook *on_rx_request* is executed when a request is received. This is the usual entry point for a SIP server, as clients direct their initial request message to a SIP proxy. This request is processed within the method and if necessary a transaction is initialized. The *on_rx_response* hook is used to receive response messages which are not sent within a transaction. On changes in the transaction state the *on_tsx_state* hook is called. Changes may be triggered by receiving responses, requests or other state changes like a timeout. In every hook, the SIP message is forwarded to the business logic after the message is processed and before the message is sent.

The *SipComponentImpl* component uses *pjlib* and *pjsip* from the PJSIP library as well as the *SCA* library. The *pjsip* library is used mainly for the SIP part where also some functionality from the *pjlib* library is needed. The *SCA* library is needed for SCA related functionality.

To accomplish the described tasks of the *SipComponentImpl* component following helper classes are used:

- proxy
- extractSipToSDO
- insertSDOtoSIP

The *proxy* class contains functionality for the SIP server. It is used as a collection of utilities to support the *SipComponentImpl* e.g. verification of a message or initialization of the SIP stack. This class uses *pjlib*, *pjlib-util* and *pjsip* from the PJSIP library.

The conversion of a SIP message to a SDO as well as the reverse operation to convert a SDO object to a SIP message are encapsulated in the classes *extractSipToSDO* and *insertSDOtoSIP*, respectively. These utility classes are used by the *SipComponentImpl*. The generated SDO object is forwarded to the business logic and with the returned SDO object the *insertSDOtoSIP* class updates the SIP messages. Both classes use the *SDO* library to build the SDO object as well as the *pjlib* and *pjsip* from the PJSIP library to process the SIP message structure.

The *BusinessLogic* defines an interface to the SCA framework. This interface is used as a SCA reference for the *SipComponentImpl*. Using this SCA reference the SDO containing the SIP message is sent via the *sipMessage* method. This interface is defined by the developed system and has to be implemented by the business logic to receive the SIP messages.

The *BusinessLogicImpl* class realizes the business logic. It is an own SCA component which implements the *BusinessLogic* interface. This class is not part of the developed system and will be created by the user of the system. Different business logic implementations are realized in the course of the prototype to test the capabilities of the system and confirm the non-functional requirements to the system. As the *BusinessLogicImpl* class is a SCA component, it needs the *SCA* library and for the processing of the passed SDO object the *SDO* library is needed. In Listing 4.2, the composite definition of the *BusinessLogic* is shown.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<componentType xmlns="http://www.osoa.org/xmlns/sca/1.0">
  <service name="BusinessLogicService">
    <interface.cpp header="BusinessLogic.h"/>
  </service>
</componentType>
```

Listing 4.2: SCA component definition of the *BusinessLogic*

This is a rather simple composite with only one service definition, named *BusinessLogic-Service*. The interface definition of this service is found in the file *BusinessLogic.h* which is a C++ header file.

The two SCA components defined in the prototype, the *SipComponent* and the *Business-Logic* component, need to be combined to a composite to wire up the reference of the *SipComponent* and tell the SCA framework to invoke the service at the developed *BusinessLogic*. The composite definition is displayed in Listing 4.3.

```
<?xml version="1.0" encoding="UTF-8"?>
<composite xmlns="http://www.osoa.org/xmlns/sca/1.0" name="SipServer">
  <component name="SipComponentComponent">
    <implementation.cpp library="SipComponent"
        header="SipComponentImpl.h" />
      <reference name="businessLogicService">
        BusinessLogicComponent/BusinessLogicService
      </reference>
  </component>

  <component name="BusinessLogicComponent">
    <implementation.cpp library="BusinessLogic"
      header="BusinessLogicImpl.h" />
  </component>
</composite>
```

Listing 4.3: SCA composite definition of the *SipComponent* and *BusinessLogic*

The name *SipServer* is given to the composite and following to that the components are listed which are part of the composite. The first component is named *SipComponentComponent* and it refers to the developed SIP/SCA component. The following line defines that the implementation is done in C++ and can be found in the library *SipComponent*. The definition of the implementation language is necessary as in the SCA system components can be implemented using different programming languages, as mentioned in Subsection 4.1.1. As the implementation of this component is compiled to a Linux library, the file has to be named *SipComponent.so*. Subsequently, the name of the header file of the implementation is mentioned: *SipComponentImpl.h*. This header file should not be mixed up with the header file of the interface, compare Listing 4.1. Subsequently, the reference of the component is wired up with the service within the business logic component. The second composite defined is named *BusinessLogicComponent* and defines the *BusinessLogic* component. The implementation is realized in C++ and can be found in the library named *BusinessLogic*. The header file of the implementation is located in the *BusinessLogicImpl.h* file.

This composite is used by the *SipProxyClientApp* class and is like the class not part of the developed system. It will be defined by the user of the system and can involve more than the two displayed components. The composite definition in Listing 4.3 is the minimum configuration necessary for the system.

Encountered Problems

At the development of the prototype a notable problem appeared. Usually, the invocation of a reference from a SCA component does not require the creation of an own context, but with the prototype this is not possible. This is because the PJSIP library uses different worker threads to handle the incoming messages. Within these threads, the context for invoking the reference is not present.

This problem is solved in the prototype by creating a new context, retrieving the SCA component and invoking a dedicated method which forwards the SIP message to the business logic. This procedure adds additional cost to the execution as the retrieving of a context is rather laborious. In the course of the prototype, no other solution could be found, but as can be seen in Subsection 4.2.3, this solution works quite well and the measured performance of the developed solution meets the requirements. As the primary target of the prototype is the proof of functionality, no further in-deep inspections are carried out.

Optimizations

Due to the low performance measured at the beginning of the test process, the prototype is analyzed for optimization opportunities. For this purpose, the SIP/SCA component is profiled by separating the application into chunks and measuring the execution time of each chunk. Due to this, it is noted that the creation of the SDO from the SIP message is consuming a significant amount of time in the magnitude of 1 to 5 ms. By analyzing the method for creating a SDO, it turned out that the instantiation of the needed helper classes used most of the time. Further investigation shows that the creation of the helper classes does not need to be done each time a message is converted.

As an optimization, these helper classes are made static and are only instanced the first time a SIP message is converted. This optimization can also be seen in the test results as a peak in the response time at the beginning of the test, see Subsection 4.2.3.

Example Business Logic Implementation

For the demonstration of the capabilities of the system an business logic is implemented which addresses a real-life problem.

The example implementation deals with the problem of finding a free air controller when a pilot is contacting the tower. A scenario is assumed where couple of air controller are working in a tower but the number can change during the day. The business logic should keep track of all logged in air controller as well as their status. If a pilot is contacting the tower, a free air controller should be selected and connected with the pilot.

The business logic is designed that way, that it keeps track of all REGISTER messages. Those are used to determine which air controller is logged in at the system and which logged off again. When a pilot wants to contact the tower he uses a well-known user name, e.g. *tower*. When the business logic receives an INVITE message with the well-known user name *tower*, it looks up the next free air controller in the list of logged in users. The original contacted user name *tower* is replaced in the INVITE message by the user name of the free air controller and the altered SIP message is sent back to the SIP/SCA component. Also, the used air controller is marked as busy. When the call is terminated, the BYE messages is recorded and the air controller is marked as free again.

This kind of use case is a very common problem and can also be found in other areas like a call center. This example shows how easy the interplay of the SIP message with the service is. This presented business logic was implemented within one hour and has been successfully tested with existing unaltered SIP clients. Changing values of the message is as easy as changing a value in a data structure. In a future development this service can be taken and orchestrated with an other services. After processing the changed SIP message would be handed to the other service which could enrich the SIP message with further informations from the radar system. The presented service does not need to be changed, not even recompiled.

4.2 Simulation and Results

As mentioned in Subsection 3.3.2, it is expected that the performance of the prototype will be worse than the performance of a SIP system with a common configuration. To seize the difference in performance the following tests focus on the measurement of the performance of the prototype. Two additional SIP scenarios are tested along with the prototype to act as a referee.

4.2.1 Test Scenarios and Metrics

To measure the performance of the prototype, different SIP load tests are conducted. SIP load tests are chosen as the overall purpose of the developed system is to handle SIP messages and the system as a whole is being tested with these SIP load tests. During the load tests, different parameters are monitored which are used to determine the performance of the system.

For the SIP load tests two different test scenarios are defined:

1. Transaction oriented test scenario
2. User registration oriented test scenario

The *transaction oriented test scenario* focus on the capability of the system to process SIP transactions. Transactions are used in SIP when multiple SIP messages are exchanged between caller and callee within the same context. Intermediate SIP nodes like the developed system need to maintain the transaction state efficiently to be able to process a high amount of SIP messages between different callers and callees. Multiple SIP messages within the same context occur at the most used method of SIP, the initialization of a session. At the same time, this is the core method of SIP. Therefore the *transaction oriented test scenario* uses the scenario of establishing a session.

The *user registration oriented test scenario* focuses on the capability of the system to maintain the registration of users. SIP user need to register its current location at the location service to be able to receive calls by other SIP user. If the location of the user changes or the user logs off the system, the registration has to be updated. To allow a quick adding, querying and updating of user information, the system has to store the user registrations in an efficient way. In the *user registration oriented test scenario*, the registration of SIP users is tested. This will highlight the performance of the registrar and the location server of the system.

To measure the performance of the system two test metrics are used:

1. Throughput
2. Message Delay

The *throughput* metric shows how many messages can be processed by the system within a certain amount of time. This is an indicator to see how many user or rather calls the system can handle at the same time. The unit of throughput is messages per second.

The *message delay* metric indicates how long a message takes to be processed and thus shows how fast the system can process a request. Specifically, it shows how long it takes to receive a response on a sent request. This also means, it measures the delay of the overall system including network latency as well as processing speed of each intermediate SIP node.

To measure the actual performance of the developed system, external influence on the system is reduced. E.g. the test infrastructure uses exclusively a high-speed local network to reduce the network latency. The unit of message delay is millisecond.

The test procedure how the introduced test metrics are collected is the same for each test scenario:

1. Find the maximum throughput.
2. Measure the message delay.

To find the maximum throughput, the rate of sent messages per second is increased until the system is overloaded. To indicate when the system is overloaded, the number of processed messages per second is monitored. When the rate of processed messages drop although the number of sent requests is increased the system is overloaded.

All tests started with the same message per second rate of 50. The rate is increased every 10 seconds by 5 messages per second. When a rate of 1000 messages per second is passed, the rate is increased by 50 messages per second each step. The average amount of processed messages per second over a 10 second interval is logged to a file. The maximum throughput is defined as the highest value of processed messages per second found in the log file.

During the execution of the tests, a problem occurred when conducting the *user registration oriented test scenario* in a specific test case. The setup of the test case was so fast that the system could process a huge amount of requests without any problem. But as a fact that in this test scenario all requests have to be stored the system seemed to have issues in storing the user registration information. Due to this, the responses generated by the system varied very much starting from a certain rate. This resulted in varying transfer rates where the transfer rate could drop, as seen in an overloading system, but then recover and increase again. As a result, repeated executions of the test resulted in significantly different results. Thus, the determination of the maximum throughput was not possible with the above defined rules. Due to this problem a rule was added where the delay of 99% of all messages must be below 50 ms. 50 ms was choosen as in all other test cases this response time was never exceeded.

For the second metric, the message delay, results from two situations are of interest. Once when the system is under load to show how the performance is at the limit. The second situation of interest, is with the same transfer rate for all test cases. This is to make the test results equally comparable between all test cases.

As results get significantly worse when the system is overloaded, this state should be avoided during the tests as the results are skewed. To be sure the system is not going to be overloaded but still under load, the message rate for gathering the message delay of the first situation will be 90% of the measured maximum throughput. In this way, the response time of the system is being tested while being under load but the state of overloading is being avoided.

For all test cases in the second situation, the message rate will be 90% of the worst measured maximum throughput per test scenario.

The message delay is measured over a period of 2 minutes. Before a test is started all used applications and components are terminated. 10 seconds before the test is starting all used applications are started. In this way, all tests are conducted under the same conditions. During the tests, the delay between the sending of a request and the corresponding response is measured and logged as message delay into a file.

During all tests additionally the Central Processing Unit (CPU) and memory load of each participating computer is logged. This information gives a clue which component is the limiting part of the system. Additionally, it shows how much memory is needed by the system.

4.2.2 Test Cases and Setup

In the previous section the test scenarios and the measured metrics are defined. In this section, the test setup is described by defining the used hardware as well as software and description of the used test cases.

The tests are conducted in the lab of the Institute of Computer Technology at Vienna University of Technology [ICT]. For the tests, three computers with equal configurations are used exclusively. Also, a dedicated private network prevents tempering of the test results by third party traffic.

The configuration of the computers is as follows:

- Ubuntu Linux 10.04 [Ubu] as operating system
- Intel Core 2 Duo processor at 2.80 GHz with two cores
- 1 Gbyte memory
- 100 Mbit dedicated network

For the test four components are needed:

- Caller
- Callee
- SIP/SCA component
- Kamailio [Kam] as SIP proxy server

As only three computers are available for testing, all components can not be run on a dedicated machine. As the caller and callee do not use very much resources and are also not topic of measurement, the same computer is used for those two components. For the SIP/SCA component as well as the SIP proxy server a dedicated machine is used. As in Subsection 4.1.1 decided, Kamailio is used as a SIP proxy server.

For the generation of the requests, *SIPp* [SIP] is used. SIPp is a flexible traffic generator for the SIP protocol. With this tool custom scenarios can be defined and the statistics of the tests are dumped into files. For the statistics the response message of each request is analyzed. With the dumped statistics, the metrics of the tests are calculated. The definitions of the tests used for the test scenarios can be found in Appendix: SIPp Scenarios. An example of a test execution is shown here:

```
sipp -r 196 -m 23520 -fd 1s -recv_timeout 5000 -sf uac_register_v1.0.xml -inf
sip_user300k.txt -i 172.16.1.2 -rsa 172.16.1.5:5065 172.16.1.1:5060 -trace_rtt -
trace_screen -trace_stat
```

The most important parameter will be explained here, for a detailed description refer to the technical documentation of SIPp. With the *-r* parameter the message rate is defined, in this case 196 messages per second. The *-m* parameter defines the amount of messages which are going to be sent in total. The *-sf* parameter points to the file where the scenario is defined. With *-trace_rtt*, *-trace_screen* and *-trace_stat* the recording of statistics and the dump to a file is activated.

In the *registration oriented test scenarios*, no callee is needed as at the registration only the client and the server is involved. For the *transaction oriented test scenarios*, an additional Kamailio SIP proxy is used to simulate the called client. It is configured to answer all INVITE requests with a 200 OK message, indicating to accept the call initialization or the call tear down.

Three different test cases are used to compare the performance of the prototype with present SIP systems:

1. One SIP proxy
2. One SIP proxy and one PJSIP server
3. The SIP/SCA component with one SIP proxy

The first test case represents a common SIP setup. Like shown in Figure 4.3, this test case consists of an ordinary SIP proxy, the Kamailio server, and two SIP clients. As this test case does not contain any self developed components and the used components are fully grown and high-performance products, the results of this test case can be seen as the best value which can be reached with the used hardware. The purpose of this test case is to give a reference to the performance of a conventional SIP setup. Because in the other test cases an additional SIP node is added and self developed components are used, it is expected that the results of those test cases are worse than the results of this test case.

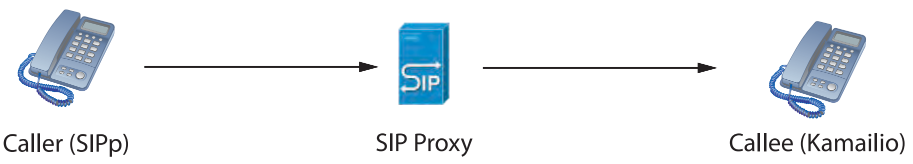

Figure 4.3: Test Case Kamailio

Figure 4.4 shows the second test case. It consists of total four components, namely the two SIP clients, the Kamailio SIP proxy server and an additional SIP server implemented with the PJSIP SIP library. The setup of this case is similar to the system setup of the prototype, but instead of the developed SIP/SCA component, a self-implemented SIP server is used. The implementation of the SIP server is similar to the SIP/SCA component, but does not contain any SCA functionality.

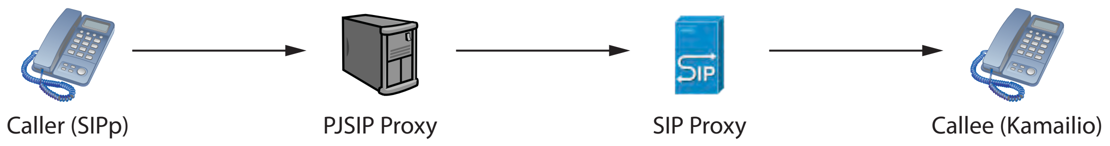

Figure 4.4: Test case PJSIP

This test case is used to show the reduction in performance compared to the first test case when an additional SIP node is added to the call flow. Also, the performance of the PJSIP library compared to the Kamailio proxy server can be seen. It also gives a clue of the impact the SCA library has on the overall performance.

In the third test case, the developed prototype is tested in the configuration as described in Section 3.2.2, see Figure 4.5. It consists of two SIP clients, the Kamailio SIP proxy and the developed SIP/SCA component. For the tests, a dummy business logic is used. As the business logic can contain any kind of code the actual runtime can vary depending on the implementation. This can reach from a couple lines of code until complex calculation where different services and/or a database is queried. As no assumptions on the business logic and thus the amount of time spent in business logic can be made, the dummy logic does not perform any calculation at all and immediately returns to the SIP/SCA component. Therefore, it has to be kept in mind, that the delay caused by the calculations in the business logic has to be added to the results of the tests.

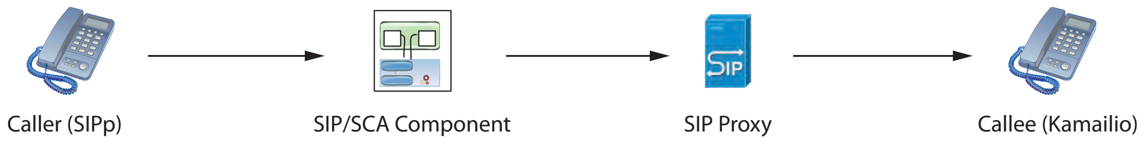

Figure 4.5: Test case SCA SIP Proxy

4.2.3 Test Results

The following test results base on the definition described in the Subsections 4.2.1 and 4.2.2. During the execution of the tests, issues occurred which required the adoption of the test sequence.

Test Case with One SIP Proxy

The measured maximum throughputs of the Kamailio SIP proxy can be seen in Figure 4.6. This results already respect the adopted test procedure, where less than 1% of the test messages can have a response time over 50 ms.

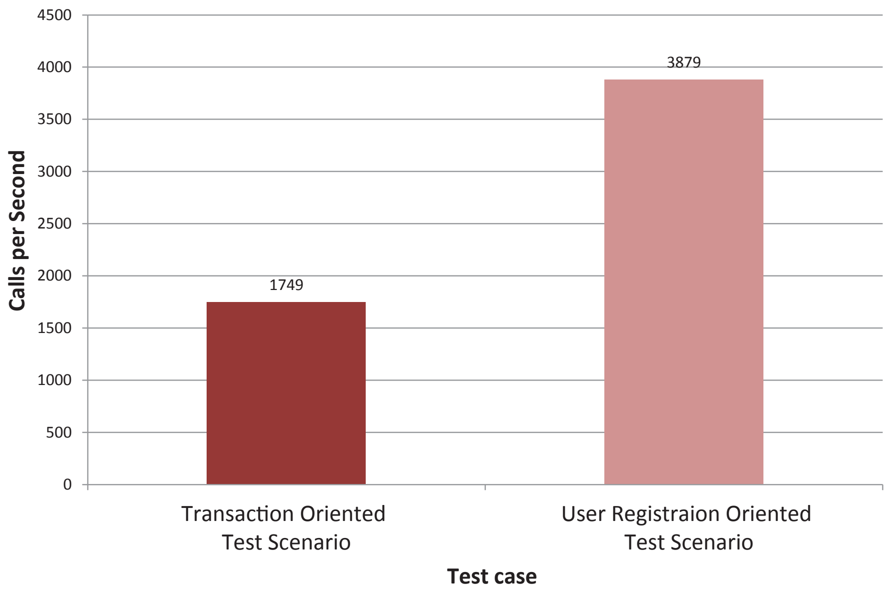

Figure 4.6: Maximum measured throughput of the Kamailio server

Figure 4.7 depicts the measured *message delays*.

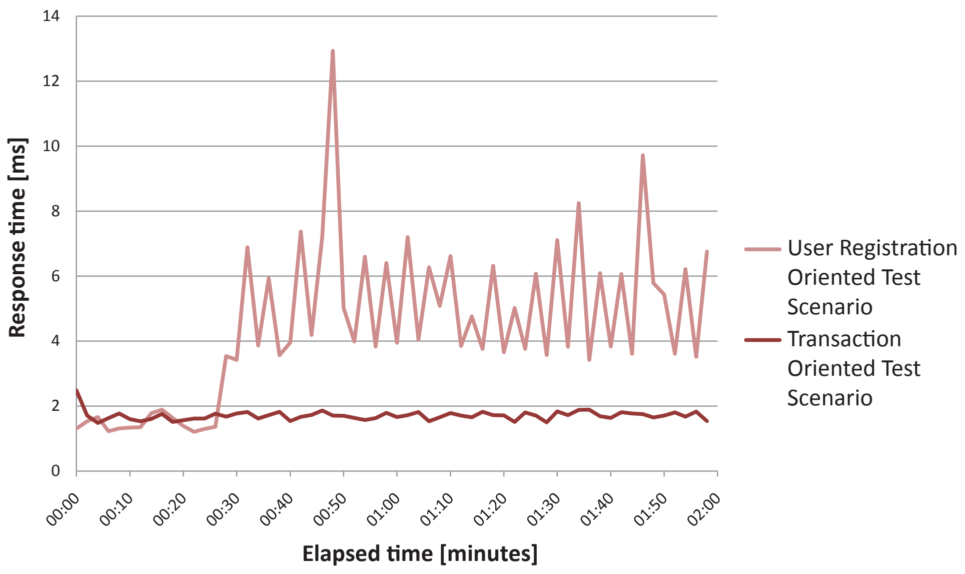

Figure 4.7: Message response time of the Kamailio server under load

As mentioned in Subsection 4.2.1 problems occurred during the *user registration oriented test scenario* while running this test case. When measuring the maximum throughput the behavior of the server was unexpected. The test procedure stipulates that the data rate is gradually increased until the rate of the processed responses drop again due to overloading of the system. In the state of overloading, the response rate should never raise again, but exactly this happened. Further investigations hypothesized that the way how the user registrations are management is the reasons for this behavior. But at this point it need to be mentioned that about 700.000 registrations within two minutes are used to get the described behavior. At this message rate also the response messages got more unreliable, this resulted in significant amounts of requests being dropped or the response delay was very high. Therefore, the additional rule is added where not more than 1% of the messages are allowed to have a response time over 50 ms.

When looking at the results of the *message delays* during the server is under load, peaks can still be seen for the *user registration oriented test scenario.* In the first 30 s the response time is at the same level as the *transaction oriented test scenario.* Starting from this moment, the response times get worse until the end of the test. The *transaction oriented test scenario*, in contrast, show reliable and constant response times with a low latency below 2 ms.

Test Case with One SIP Proxy and One PJSIP Server

In the second test case, no problems are encountered. The measured calls per second are reasonable as can be seen in Figure 4.8.

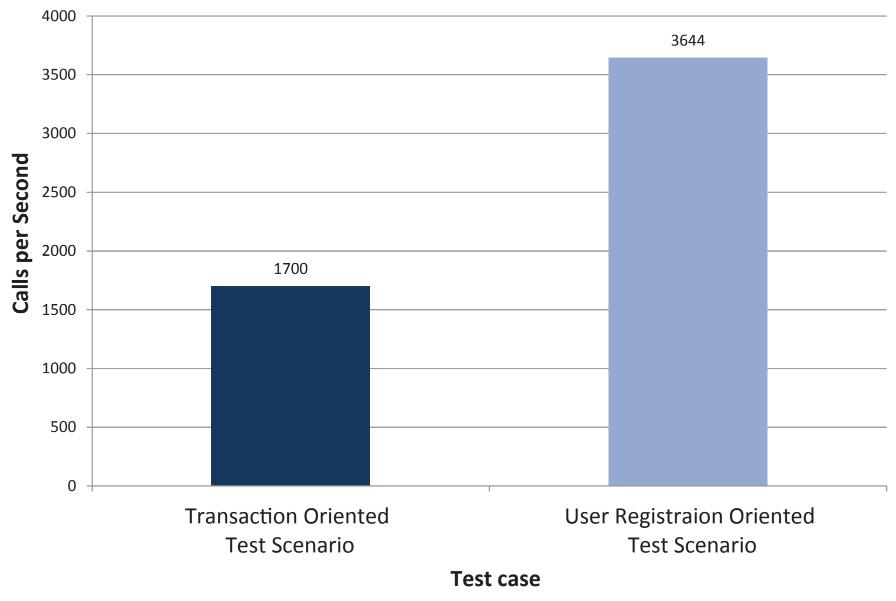

Figure 4.8: Maximum measured throughput of the second test case

The message rate of this test case is almost as high as the test case with the full grown SIP server. Taking into account that this test case has one more entity than the previous one, the performance of the SIP library is respectable.

Figure 4.9 shows the message delay under load.

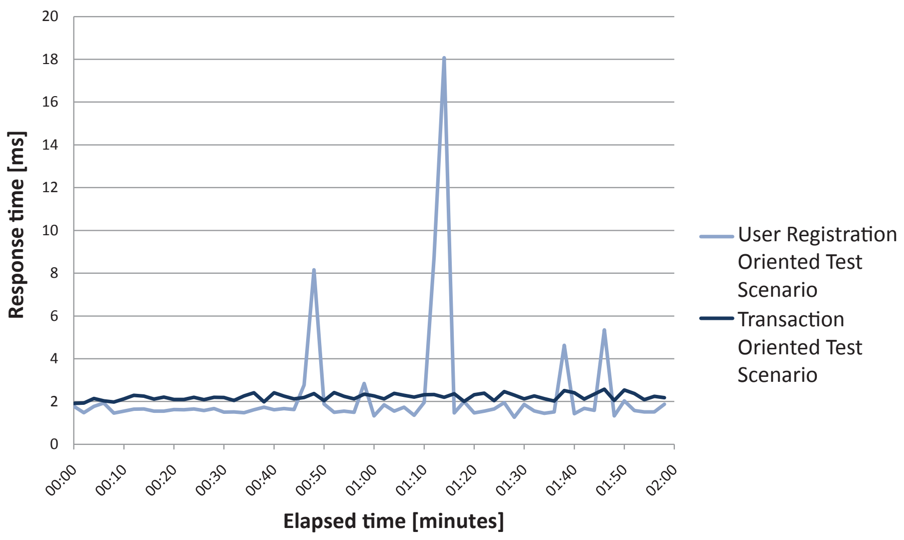

Figure 4.9: Message response time of the second test case under load

Overall the message delay is constant with low delay, only in the *user registration oriented test scenario* peaks can be seen. The reason for the peaks can be found in the Kamalio SIP server which processes the registrations. For a closer discussion see Section 4.3.

Test Case with the Designed SIP/SCA Component

Figure 4.10 shows the measured calls per second of the developed system.

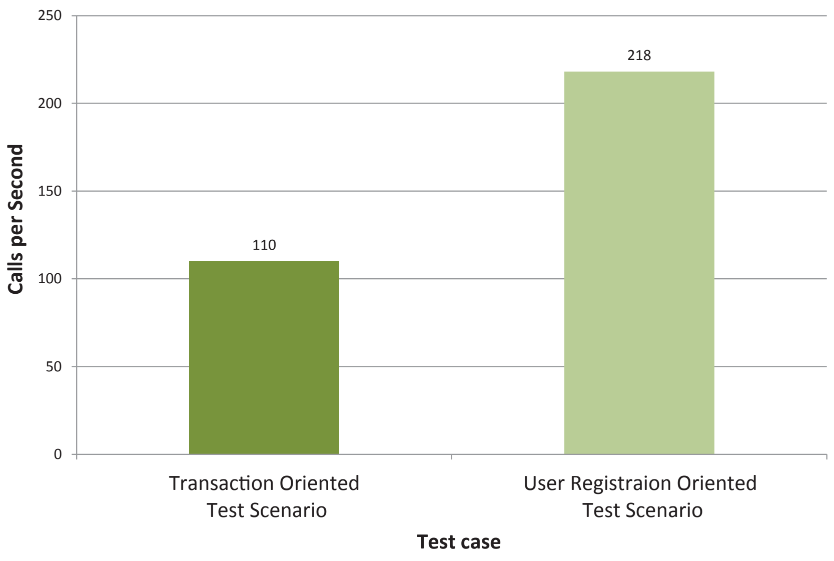

Figure 4.10: Maximum measured throughput of the developed system

The calls per second show that the developed system is slower than the comparing test scenarios. But at least a rate of 100 calls per second was reached in any test scenario.

In Figure 4.11 the message delay is depicted.

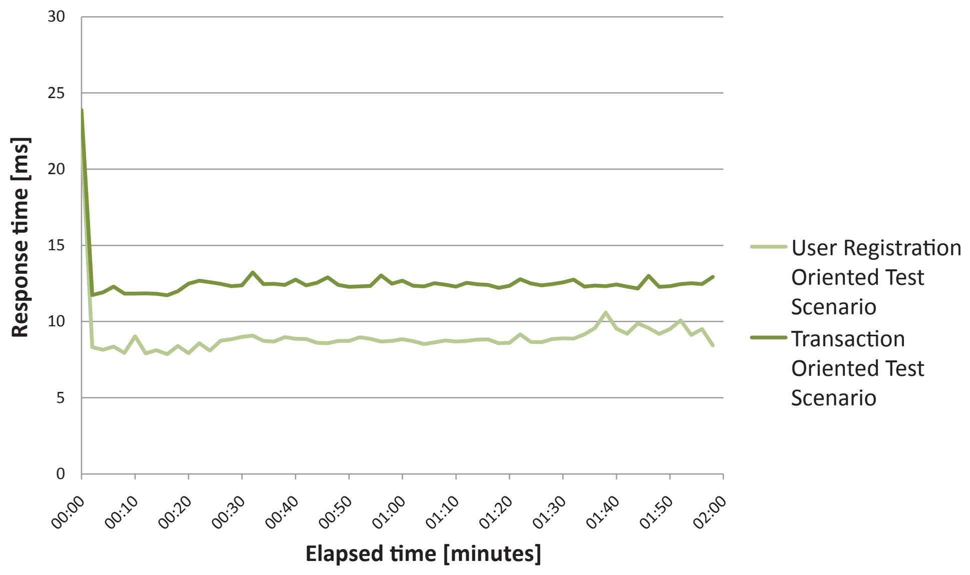

Figure 4.11: Message response time of the developed system under load

The message delay under load shows a constant delay without any spikes. This shows that the system is still responding well and in a predictable manner although it is under heavy load.

Notable is the peak at the beginning of the test at minute 0:00. This peak comes from optimization efforts described in Subsection 4.1.2. For the generation of an SDO object, a factory class needs to be loaded with the XML definition of the SDO object. This load takes a significant amount of time. Compared to other instructions executed during the processing of a SIP message, parsing XML data is an expensive operation. As an optimization the XML definition is loaded only once and then cached for later creations of an SDO object. The load of the XML definition is happening at the reception of the first SIP message, therefore the processing of this request will take longer. Future requests does not need to undergo the mentioned initializations and therefore the requests will be processed faster. Due to this, a peak can be seen in each test of the message delay at the beginning.

4.3 Discussion

The implemented prototype shows that the proposed design in Section 3 is realizable. In particular, the prototype shows that the interception and changing of SIP messages can be realized without violating the standard and thus, the system is compatible with existing SIP clients. Also it proves that the anticipated benefits of faster and easier development and re-usability of services can be achieved.

During development and functional tests, available SIP clients are used to prove the capability of the system to work with existing SIP components. Different clients running on different operating systems worked without problems. From the point of functionality, the prototype meets all desired requirements to its fullest.

To meet the requirements related to the limited hardware it is achieved that every aspect of the prototype is implemented using the programming language C/C++. So the used SIP library is implemented in C/C++ as well as the chosen SCA implementation. The file size of the SIP/SCA component is only 892 KByte, which is very small. But this is only the size of the complied Linux library. The business logic, client application for starting the component as well as other services have to be added. Additionally, sizes of the PJSIP library, SCA and SDO need to be added. The compiled Linux library of the used PJSIP APIs are in total 2.5 MByte, the deployed SCA system has roughly 11 MByte and the SDO 5.5 MByte.

In Figure 4.12 the transfer rates of all test cases are displayed of the *user registration test scenario*.

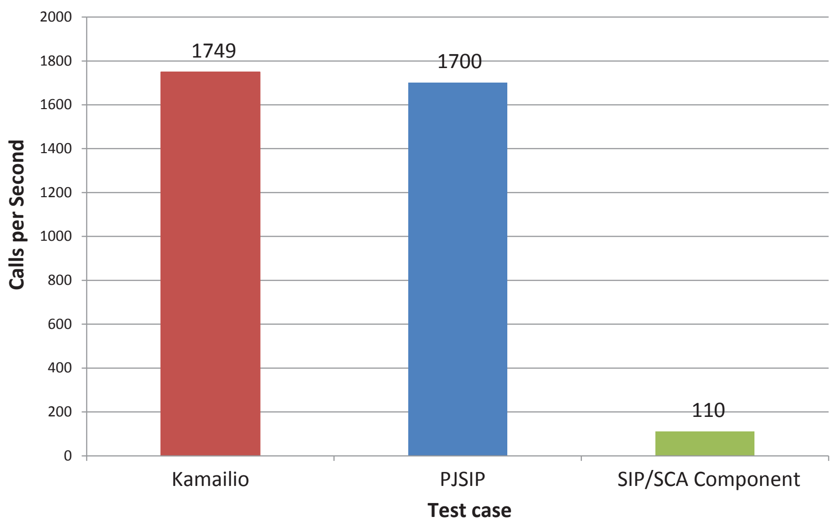

Figure 4.12: Transfer rate of all test cases at the transaction oriented test scenario

The performance of the prototype fulfills the requirements, but compared to existing solutions it can only process a fraction of requests. It shows that the prototype could only achieve 5.6% of the performance of the Kamailio SIP server.

A similar picture is shown in the *user registration test scenario*, compare Figure 4.13. The performance of the prototype is at 6.3% of the Kamailio SIP server. Impressive is the performance of the PJSIP library. In both test cases, the performance of the library is close to the Kamailio SIP server.

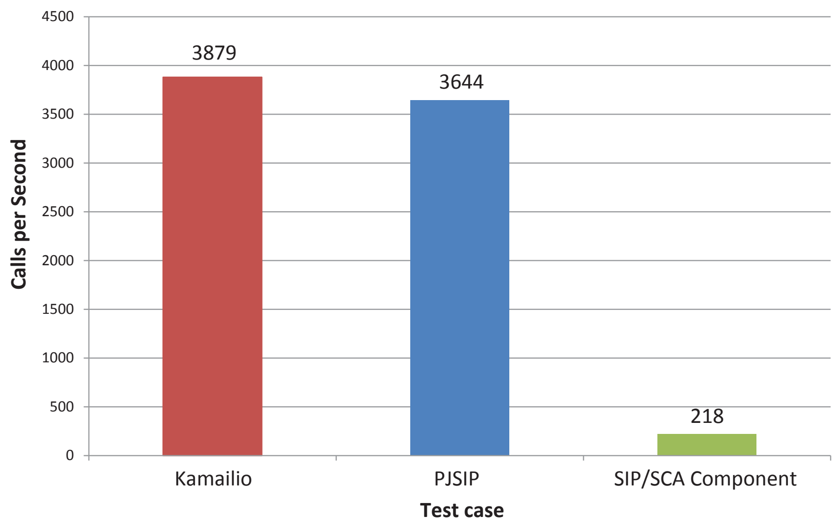

Figure 4.13: Transfer rate of all test cases at the user registration test scenario

In Figure 4.14 the measured message delay recorded under load shows similar results of the transfer rates at the *transaction oriented test scenario*.

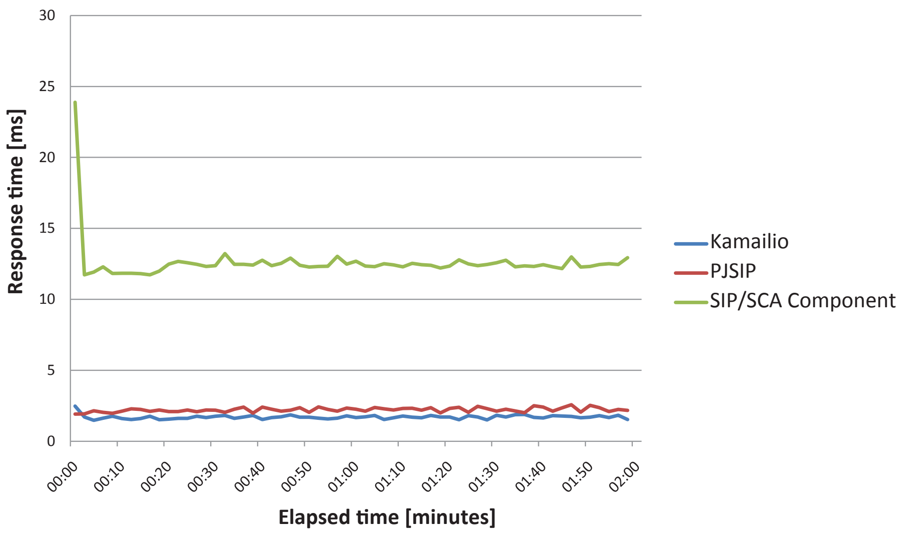

Figure 4.14: Response times of the transaction oriented test scenario under load

The Kamailio SIP server has slightly shorter response times than the test case with PJSIP. Respectable performance of the PJSIP library, specially if the fact is recalled, that the message has to pass in total two SIP server in the test case with the PJSIP server and only one SIP server in the test case involving only the Kamailio SIP server.

The test case of the prototype shows at the beginning the peak which is already discussed in Subsection 4.2.3. Compared to the other test cases the performance is in average 6,5 times slower than the test case involving only the Kamailio SIP server.

The results of the message delay with constant transfer rates for all test cases, depicted in Figure 4.15, do not show very different results compared to the results of the message delay where the systems are under load. The measured message delays are in about the same. This shows that the systems responsiveness is not reduced when getting under load in this test scenario.

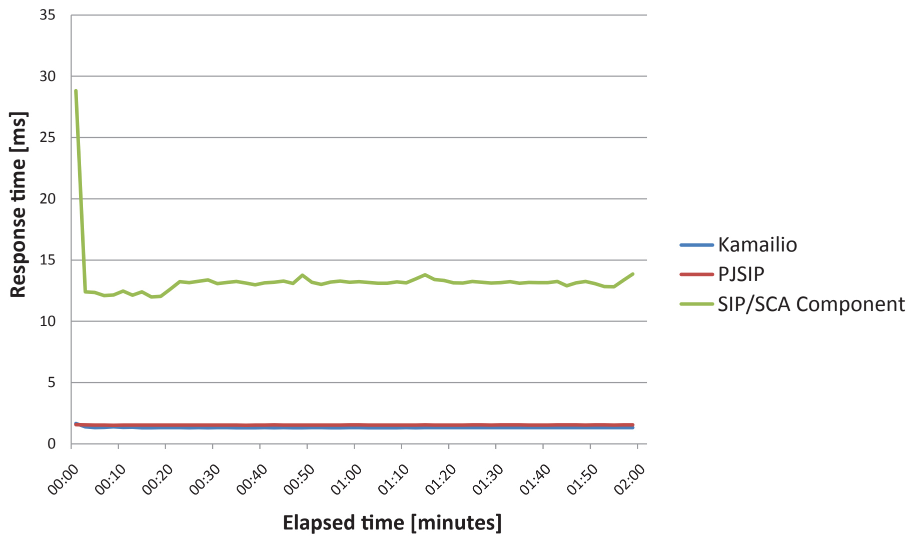

Figure 4.15: Response times of the transaction oriented test scenario with a constant transfer rate of 98 calls per second

In Figure 4.16 the measured message delay of the *user registration test scenario* is shown.

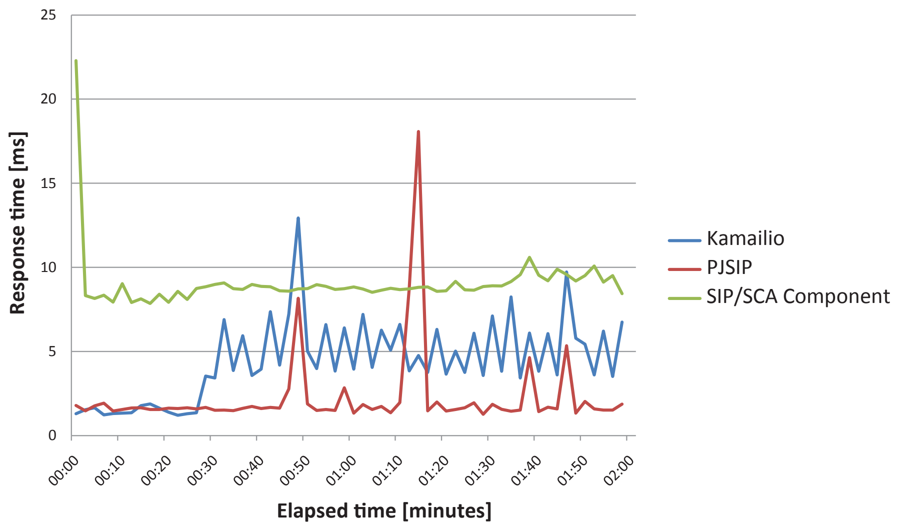

Figure 4.16: Response times of the user registration test scenario under load

The measured message delay of the *user registration test scenario* does not show such constant response rate for all test cases as in the *transaction oriented test scenario*. For the test case involving only the Kamailio SIP server the reasons of the peaks have been discussed in Subsection 4.2.3.

The reason for the peaks in the test case involving the PJSIP server can also be found in the Kamailio SIP server. Looking at the transfer rates of the test case involving the PJSIP server and the test case involving only the Kamailio SIP server, only a rather small difference between those two test cases can be measured. This can also be seen as the cause for the peaks in the test scenario involving the PJSIP server. The reason for the message delays not being so scattered as the test case involving only the Kamailio SIP server lies in the still lower transfer rates of the test case.

As the test case of the prototype is run with a fraction of the transfer rate of the test case involving the Kamailio SIP server, the Kamailio SIP server does not have any influence on the message delays.

Running this test scenario with the same transfer rate for all test cases shows a different picture, see Figure 4.17.

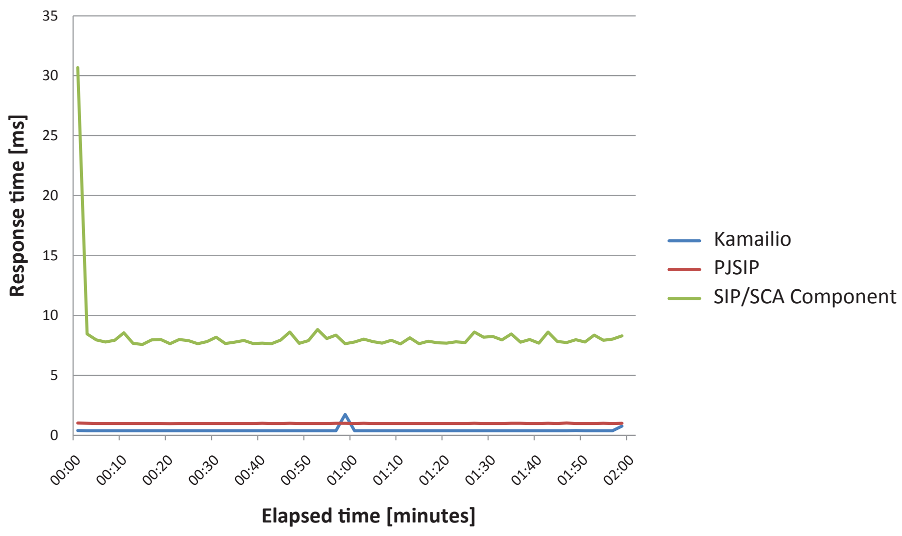

Figure 4.17: Response times of the user registration test scenario with a constant transfer rate of 196 calls per second

Compared to before, the response times of the test case involving the Kamailio server as well as the test case involving the PJSIP server are constantly low. The reason lies in the rather low transfer rate of 196 calls per second for each test case. The response times of the prototype are similar to before.

Overall, the performance of the prototype is far beneath the performance of the compared systems. The comparison to the PJSIP server shows that the SCA runtime is consuming a fair amount of processing time. At the test configuration, this delay is caused by the framework and the developed code to convert the SIP message for SCA. When viewing at the results, it need to be borne in mind, that the used business logic does not consume any processing time which in a productive use will not be the case.

Nevertheless, the measured message delay as well as the message rate meet the requirements.

Also, the SCA implementation of Tuscany is still under development. The used SCA version is only a development version and cannot be seen as a finalized version. Improvements in the performance might be possible as well as the implementation of the prototype bear potential to improve performance.

An other option to gain a higher performance is to install multiple computer with the SIP/SCA component. This is possible as the system is designed to support multiple instances. A SIP load balancer needs to be prepended to equally distribute the requests among the instances.

5 Conclusion and Outlook

A system was developed in the present thesis which integrates a SIP server into a system based on SOA, respecting the defined environment of limited hardware. The chapter gives a summary on the reasons for the development, the important steps for the design and all important results of this work. Subsequently, an outlook to future development is given.

5.1 Conclusion

In an attempt to modernize the systems of air traffic controls, the idea to build a system based on the paradigm of SOA is picked up. In the course of this modernization, the voice communication, which is mainly held between the air traffic controller and the pilot, is changed to be based on VoIP. The integration of those two technologies is a consequential step.

By the usage of VoIP, savings in the cost of infrastructure can be achieved as the same network as for the computers can be used. Also, VoIP can be easier and more cost efficient integrated into computer systems.

The rationales to build a system based on SOA lie in various reasons. One is that legacy systems are still in use and the paradigm of SOA offers a way to preserve them and use them in new applications. An other reason is the re-usability of services which a system based on SOA abets. Due to this, developed services can be reused in later projects and so save effort as well as accelerate development.

By integrating VoIP in a system of SOA, the usage of VoIP should be eased and the advantages of services should be also applied. This means, the re-usability of the developed modules should be increased as well as the ability to be orchestrated.

Safety critical systems - like the controlling of air traffic - need to be highly reliable and must undergo different verification processes. Due to this, the hardware for the present thesis is predefined. This hardware underwent different verification and certification processes which makes the hardware development laborious and thus, updates are done more seldom.

Therefore, the predefined hardware can be seen limited compared to modern computer systems which is why a lightweight solution is needed.

Not only the hardware, but also the software needs to be certified. Not different than the hardware, the process is laborious. By using a programming language which base on a virtual machine or an interpreter, those parts also need to be examined which are generally complex. To reduce the effort and in respect to the limited hardware a solution producing native code is desired.

Different approaches and systems were already engineered to integrate or partly integrate those two technologies, as described in Section 2.3. But existing solutions do not satisfy all defined requirements to the system, therefore a system is developed in the present thesis to fulfill all requirements, see Section 3.1.

As a first step, a suitable VoIP protocol and system of SOA is selected in Section 2.1 respectively Section 2.2. SIP is chosen for the VoIP protocol as it is wide spread. As an realization of SOA, SCA is chosen as it offers an implementation in C++ and is a lightweight system of SOA. Therefore, SCA does not implement a service bus, see Subsection 2.2.5. SCA uses a different approach to interconnect the services and so avoids the implementation of a expensive service bus, as described by the SOA paradigm.

Next, the problem to intercept SIP messages is addressed in Subsection 3.2.1. The related work from Section 2.3 is analyzed if it can be used for solving the described problem or parts of the problem. As a result, it turned out that no described work is suitable and therefore, an own solution is being developed. Subsequently, all SIP server described in the standard are examined if they are suitable for intercepting SIP messages. An appropriate solution for this subproblem can be found with a B2BUA.

In the third step, an appropriative configuration of the physical architecture is discussed in Subsection 3.2.2. It also covers the necessary configuration of the SIP clients as well as the DNS system. An additional SIP server is added to the system to process and maintain the registrations of the SIP user and the final routing of messages. Due to this, different scenarios are explained in which order a SIP message can pass the nodes of the system. The decision is made to first pass the SIP message to the SIP/SCA component and then to the additional SIP server. This offers the most benefits for the least cost in terms of performance expenses and complexity.

Finally, methods are described in Subsection 3.2.3 how the developed SIP server can be integrated into SCA. It is decided to run the SIP server within the SCA runtime as a service. This represents the idea of SOA best. Different ways to pass the information of a SIP message to other SCA services are discussed. Thereby, a focus is given on the easy handling of the submitted data by the user defined services. As a result, the SIP message is transformed to a SDO which is then passed to the user defined services.

To prove the concept and model, a prototype is developed, as described in Section 4.1. With the prototype, different services are implemented to show the possibilities as well as the fast and easy development of the system.

SIP load tests are conducted, as described in Section 4.2, and subsequently the results are presented. The prototype is compared with a SIP reference system and a system involving a server built similar to the prototype but not including any SCA. It shows that the performance of the prototype is only at 5.6% or rather 6.3% of the reference SIP system. More interesting are the results of the test case involving the SIP server build with the PJSIP library. As this SIP server and the prototype only differ in the usage of SCA, it points out how much performance the SCA consumes. The difference in the results are significant too and let asume that the SCA system uses a majority of the resources.

Although the performance of the prototype is far behind the reference systems it fulfills the requirements. Especially when taking into account the different optimization potentials of the prototype pointed out in Section 4.3 and the fact, that the used SCA implementation of Tuscany is a development preview and thus, the performance can increase in future versions.

The present thesis shows one way how SIP as a VoIP protocol can be integrated in a SCA as a system of SOA. This solution is due to special requirements developed in a lightweight manner. The prototype shows the feasibility and that the services using SIP can be handled as any other SCA service. It also reveals that the performance is significantly slower than existing SIP systems. As proving performance is not a primary goal of the prototype, less effort is spent on optimizations. Nonetheless, the achieved working speed is already sufficient to fulfill the requirements.

5.2 Outlook

VoIP is an emerging technology and it is slowly replacing the traditional telephone system. With the raising popularity also more development effort is invested in this area. Similar to traditional software development, the need of re-usable software packets rise. With this re-usable software packets, shorter development cycles can be realized and less error prone applications can be released.

The present thesis proposes a way to integrate a system based on SOA with SIP. The developed prototype proves the feasibility and confirms the ease of use and re-usability, but it also shows different drawbacks.

A reduction in performance was expected and the performance of the prototype is only a fraction of existing SIP systems. But the system has potential for different improvements. As the SCA part of the prototype seem to consume a fair amount of processing time, the transition of the SIP message into a SDO object and the invoking of the SCA services need to be investigated for optimization possibilities. Also the usage of multiple processor cores would significantly improve the performance of the system, as the developed prototype only uses one core.

A future improvement would be to add capabilities to review the returned SDO for correctness. Currently, the prototype relies on the services to alter the SIP message correctly. Invalid changes by the services are not detected by the prototype and lead to an invalid SIP message. Other functionalities could be added to ease the handling of SDO by the services. E.g. the calculation of the Content-Length header field could be done by the prototype so that the services could change the body of the message freely without caring to update the header field.

For specific application scenarios, investigations on the need for all header messages or SIP messages can be undergone. In some cases, only the REGISTER messages might be of interest and by ignoring other messages, the performance can be significantly improved. Also some type of SIP messages could not be useful for the services, e.g. 100 Trying or 180 Ringing. By similarly selecting only needed header fields the conversion of the SIP message to a SDO could be accelerated. It is thinkable that header fields like Via, Route or CSeq are in some cases not used by the services.

Currently, only SIP messages and header fields are supported which are defined in the SIP standard [RSC+02]. A future development might integrate SIP extensions and so broad the functionality of the developed system. Different interesting and meaningful applications could be realized. For example, the support of third party call control [RPSC04] could improve the speed of reaction in the air traffic control. In case of an impend collision of two airplanes the radar subsystem could notice this and notify the responsible air traffic controller. At the same time, the system could automatically initiate a voice call between the controller and the pilot. In this way, crucial time can be saved in suddenly developing dangerous situations.

An other future development might be to enable the system to be the sender and receiver of SIP messages. Until now, the designed system can only receive and forward messages, but it cannot initiate and thus terminate a communication. This functionality could also be used for developing clients for the air traffic controller to communicate with the pilots. Also, those clients could use a system based on SOA to use the introduced advantages.

The simple usage and the re-usability of the services enable a new way to develop SIP functionality. The services implemented in the course of the present thesis only deliver a small insight to the possibilities of the system. To be able to add information directly to calls or manipulate messages on the way to the receiver, enables the development of new system architectures. By enriching a call with all needed information, the client application can use them straight away and does not need to gather these data via a second channel, as it is mostly done with present systems.

Appendix: SDO Definition

```
<xsd:schema
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns:sdo="commonj.sdo"
    xmlns:sdoxml="commonj.sdo/xml"
    xmlns:company="SipMessageNS"
    targetNamespace="SipMessageNS">

  <xsd:element name="SipMessage" type="SipMessageType"/>
  <xsd:complexType name="SipMessageType">
    <xsd:element name="method" type="xsd:string"/>
    <xsd:element name="targetURI" type="xsd:string"/>
    <xsd:element name="body" type="SipBody"/>
    <xsd:sequence>
      <xsd:element name="sipHDR" type="SipHeader" maxOccurs="unbounded"/>
    </xsd:sequence>
    <xsd:attribute name="cs" type="sdo:ChangeSummaryType"/>
   </xsd:complexType>

   <xsd:complexType name="SipBody">
    <xsd:element name="Content-Type" type="content_type"/>
    <xsd:element name="length" type="xsd:integer"/>
    <xsd:element name="data" type="xsd:string"/>
   </xsd:complexType>

   <xsd:complexType name="SipHeader">
    <xsd:attribute name="SipHeaderType"/>
    <xsd:choice>
      <xsd:element name="From" type="fromto"/>
      <xsd:element name="To"  type="fromto"/>
      <xsd:element name="Accept" type="generic_array_hdr"/>
      <xsd:element name="Accept-Encoding" type="generic_string_hdr"/>
      <xsd:element name="Accept-Language" type="generic_string_hdr"/>
      <xsd:element name="Allow" type="generic_array_hdr"/>
      <xsd:element name="Call-ID" type="generic_string_hdr"/>
      <xsd:element name="Contact" type="contact"/>
    <!-- not used here, see the body complex type!
      <xsd:element name="Content-Length" type="generic_integer_hdr"/>
      <xsd:element name="Content-Type" type="content_type"/>
    -->
      <xsd:element name="CSeq" type="generic_integer_hdr"/>
      <xsd:element name="Max-Forwards" type="generic_integer_hdr"/>
      <xsd:element name="Min-Expires" type="generic_integer_hdr"/>
      <xsd:element name="Require" type="generic_array_hdr"/>
      <xsd:element name="Route" type="routing"/>
      <xsd:element name="Supported" type="generic_array_hdr"/>
      <xsd:element name="Unsupported" type="generic_array_hdr"/>
      <xsd:element name="Via" type="via"/>
    </xsd:choice>
   </xsd:complexType>

<!-- Start of header type definitions -->
   <xsd:complexType name="generic_array_hdr">
    <xsd:element name="count" type="xsd:integer"/>
    <xsd:sequence>
      <xsd:element name="value" type="xsd:string" maxOccurs="32"/>
    </xsd:sequence>
   </xsd:complexType>

   <xsd:complexType name="generic_string_hdr">
    <xsd:element name="value" type="xsd:string"/>
   </xsd:complexType>

   <xsd:complexType name="generic_integer_hdr">
    <xsd:element name="ivalue" type="xsd:integer"/>
   </xsd:complexType>

   <xsd:complexType name="fromto">
    <xsd:element name="uri" type="xsd:string"/>
    <xsd:element name="tag" type="xsd:string"/>
   </xsd:complexType>

   <xsd:complexType name="contact">
    <xsd:element name="star" type="xsd:integer"/>
    <xsd:element name="uri" type="xsd:string"/>
    <xsd:element name="q1000" type="xsd:integer"/>
    <xsd:element name="expires" type="xsd:integer"/>
```

```
  </xsd:complexType>

  <xsd:complexType name="content_type">
   <xsd:element name="type" type="xsd:string"/>
   <xsd:element name="subtype" type="xsd:string"/>
  </xsd:complexType>

  <xsd:complexType name="route">
   <xsd:element name="type" type="xsd:string"/>
   <xsd:element name="subtype" type="xsd:string"/>
  </xsd:complexType>

  <xsd:complexType name="routing">
   <xsd:element name="display" type="xsd:string"/>
   <xsd:element name="uri" type="xsd:string"/>
  </xsd:complexType>

  <!-- http://www.pjsip.org/pjsip/docs/html/structpjsip__via__hdr.htm#aab73da4bb1327b94d99ca93f9da71bc3 -->
  <xsd:complexType name="via">
   <xsd:element name="transport" type="xsd:string"/>
   <xsd:element name="sent_by_host" type="xsd:string"/>
   <xsd:element name="sent_by_port" type="xsd:integer"/>
   <xsd:element name="ttl_param" type="xsd:integer"/>
   <xsd:element name="rport_param" type="xsd:integer"/>
   <xsd:element name="maddr_param" type="xsd:string"/>
   <xsd:element name="recvd_param" type="xsd:string"/>
   <xsd:element name="branch_param" type="xsd:string"/>
   <xsd:element name="comment" type="xsd:string"/>
  </xsd:complexType>

</xsd:schema>
```

Listing 1: SDO definition

Appendix: SIPp Scenarios

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<!DOCTYPE scenario SYSTEM "sipp.dtd">

<scenario name="Basic Sipstone UAC">
  <!-- In client mode, the Call-ID MUST be generated by sipp. -->
  <!-- To do so, use [call_id] keyword.                       -->
  <send retrans="500" start_rtd="true" >
    <![CDATA[

      INVITE sip:[call_number]@[remote_ip]:[remote_port] SIP/2.0
      Via: SIP/2.0/[transport] [local_ip]:[local_port];branch=[branch]
      From: sipp <sip:sipp@[local_ip]:[local_port]>;tag=[call_number]
      To: sut <sip:[call_number]@[remote_ip]:[remote_port]>
      Call-ID: [call_id]
      CSeq: 1 INVITE
      Contact: sip:sipp@[local_ip]:[local_port]
      Max-Forwards: 70
      Subject: Performance Test
      Content-Type: application/sdp
      Content-Length: [len]

      v=0
      o=user1 53655765 2353687637 IN IP[local_ip_type] [local_ip]
      s=-
      c=IN IP[media_ip_type] [media_ip]
      t=0 0
      m=audio [media_port] RTP/AVP 0
      a=rtpmap:0 PCMU/8000

    ]]>
  </send>
```

```

<recv response="100"
      optional="true">
</recv>

<recv response="200" >
</recv>

<send>
  <![CDATA[

    ACK sip:[call_number]@[remote_ip]:[remote_port] SIP/2.0
    Via: SIP/2.0/[transport] [local_ip]:[local_port];branch=[branch]
    From: sipp <sip:sipp@[local_ip]:[local_port]>;tag=[call_number]
    To: sut <sip:[call_number]@[remote_ip]:[remote_port]>[peer_tag_param↩
        ]
    Call-ID: [call_id]
    CSeq: 1 ACK
    Contact: sip:sipp@[local_ip]:[local_port]
    Max-Forwards: 70
    Subject: Performance Test
    Content-Length: 0

  ]]>
</send>

<send retrans="500">
  <![CDATA[

    BYE sip:[call_number]@[remote_ip]:[remote_port] SIP/2.0
    Via: SIP/2.0/[transport] [local_ip]:[local_port];branch=[branch]
    From: sipp <sip:sipp@[local_ip]:[local_port]>;tag=[call_number]
    To: sut <sip:[call_number]@[remote_ip]:[remote_port]>[peer_tag_param↩
        ]
    Call-ID: [call_id]
    CSeq: 2 BYE
    Contact: sip:sipp@[local_ip]:[local_port]
    Max-Forwards: 70
    Subject: Performance Test
    Content-Length: 0

  ]]>
</send>

<recv response="200" crlf="true" rtd="true">
```

```
  </recv>

  <!-- definition of the response time repartition table (unit is ms) -->
  <ResponseTimeRepartition value="5, 10, 15, 20, 25"/>

</scenario>
```

Listing 2: Transaction oriented test scenarios definition

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<!DOCTYPE scenario SYSTEM "sipp.dtd">

<scenario name="Basic Sipstone UAC">
  <!-- In client mode, the Call-ID MUST be generated by sipp. -->
  <!-- To do so, use [call_id] keyword.                       -->
  <send retrans="500" start_rtd="true">
    <![CDATA[

      REGISTER sip:[remote_ip]:[remote_port] SIP/2.0
      Via: SIP/2.0/[transport] [local_ip]:[local_port];branch=[branch]
      From: sipp <sip:[field0]@[remote_ip]:[local_port]>;tag=[call_number]
      To: sut <sip:[field0]@[remote_ip]:[remote_port]>
      Call-ID: [call_id]
      CSeq: 4 REGISTER
      Contact: sip:[field0]@[local_ip]:[local_port]
      Expires: 120
      Max-Forwards: 70
      Subject: Performance Test
      Content-Type: application/sdp
      Content-Length: [len]

    ]]>
  </send>

  <!-- By adding rrs="true" (Record Route Sets), the route sets      --↩
      >
  <!-- are saved and used for following messages sent. Useful to test  --↩
      >
  <!-- against stateful SIP proxies/B2BUAs.                             --↩
      >
  <recv response="400" optional="true" rtd="true">
    <action>
      <exec int_cmd="stop_call"/>
    </action>
  </recv>

  <recv response="200" crlf="true" rtd="true">
  </recv>

  <!-- This delay can be customized by the -d command-line option -->
  <!-- or by adding a milliseconds = "value" option here. -->
  <!-- <pause milliseconds="1"/> -->

  <!-- definition of the response time repartition table (unit is ms) -->
  <ResponseTimeRepartition value="5, 10, 15, 20, 25"/>

</scenario>
```

Listing 3: Registration oriented test scenarios definition

Literature

[AF03] F. Andreasen and B. Foster. Media Gateway Control Protocol (MGCP) Version 1.0. RFC 3435, Internet Engineering Task Force, January 2003. Updated by RFC 3661.

[Apa] Apache licence version 2. [Online]. `http://www.apache.org/licenses/LICENSE-2.0.html` [retrieved at 15.6.2011].

[AWW07] A. Acharya, X. Wang, and C. 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.

[Axi] Apache axis2 tools. [Online]. `http://axis.apache.org/axis2/java/core/tools/index.html` [retrieved at 15.6.2011].

[Bad09] A. Badach. *Voice over IP - Die Technik.* Hanser, 2009.

[BBB$^+$05] M. Beisiegel, H. Blohm, D. Booz, J.-J. Dubray, A. Colyer, M. Edwards, D. Ferguson, B. Flood, M. Greenberg, D. Kearns, J. Marino, J. Mischkinsky, M. Nally, G. Pavlik, M. Rowley, K. Tam, and C. Trieloff. SCA Whitepaper Version 0.9, November 2005.

[BBB$^+$07] M. Beisiegel, H. Blohm, D. Booz, M. Edwards, O. Hurley, S. Ielceanu, A. Miller, A. Karmarkar, A. Malhotra, J. Marino, M. Nally, E. Newcomer, S. Patil, G. Pavlik, M. Raepple, M. Rowley, K. Tam, S. Vorthmann, P. Walker, and L. Waterman. SCA Assembly Model Specification Version 1.0, March 2007.

[BBC$^+$07] M. Beisiegel, D. Booz, C.-Y. Chao, M. Edwards, S. Ielceanu, A. Karmarkar, A. Malhotra, E. Newcomer, S. Patil, M. Rowley, C. Sharp, and U. Yalcinalp. SCA Policy Framework Version 1.0, March 2007.

[BPE] Business process execution language version 2.0 specification. [Online]. `http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html` [retrieved at 15.6.2011].

[BPM$^+$08] T. Bray, J. Paoli, E. Maler, F. Yergeau, and C. M. Sperberg-McQueen. Extensible Markup Language (XML) 1.0 (Fifth Edition). W3C recommendation, W3C, November 2008.

[Cha07] D. Chappell. Introducing SCA. White paper, Chappell & Associates, July 2007.

[CHvRR05] L. Clement, A. Hately, C. von Riegen, and T. Rogers. UDDI Spec Technical Committee Draft, Dated 20041019, February 2005.

[CJ08] G. Camarillo and A. Johnston. Conference Establishment Using Request-Contained Lists in the Session Initiation Protocol (SIP). RFC 5366, Internet Engineering Task Force, October 2008.

[CK08] Y. Cosmadopoulos and M. Kulkarni. Java Specification Requests: SIP Servlet v1.1. Java specification requests, Java Comunity Process, August 2008.

[CLL06] W. Chou, F. Liu, and L. Li. Web Service for Tele-Communication. In *Proceedings of the Advanced International Conference on Telecommunications and International Conference on Internet and Web Applications and Services (AICT-ICIW '06)*, pages 88–93, February 2006.

[CRS$^+$02] B. Campbell, J. Rosenberg, H. Schulzrinne, C. Huitema, and D. Gurle. Session Initiation Protocol (SIP) Extension for Instant Messaging. RFC 3428, Internet Engineering Task Force, December 2002.

[CWMR07] R. Chinnici, S. Weerawarana, J.-J. Moreau, and A. Ryman. Web Services Description Language (WSDL) Version 2.0 Part 1: Core Language. W3C recommendation, W3C, June 2007.

[DR08] T. Dierks and E. Rescorla. The Transport Layer Security (TLS) Protocol Version 1.2. RFC 5246, Internet Engineering Task Force, August 2008. Updated by RFCs 5746, 5878.

[ED109a] *Voice over the Internet Protocol (VoIP) Air Traffic Management (ATM) System Operational and Technical Requirements.* EUROCAE Working Group 67, 2009. ED-136.

[ED109b] *Interoperability Standards for VoIP ATM Components.* EUROCAE Working Group 67, 2009. ED-137 Final Part 1 Radio, Part 2 Telephone, Part 3 Recording, Part 4 Supervision.

[ED109c] *Network Requirements and Performances for Voice over Internet Protocol (VoIP) Air Traffic Management (ATM) Systems.* EUROCAE Working Group 67, 2009. Part 1: Network Specification, Part 2: Network Design Guideline.

[EK10] D. Eier and W. Kampichler. Eurocae WG-67 standards for voice-over-IP in ATM for advanced NEXTGEN conops. In *Proceedings of the Integrated Communications Navigation and Surveillance Conference (ICNS), 2010*, pages C8–1 – C8–9, May 2010.

[Erl05] T. Erl. *Service-Oriented Architecture: Concepts, Technology, and Design.* Prentice Hall PTR, Upper Saddle River, NJ, USA, 2005.

[ESS$^+$09] J.-P. Elsholz, H. Schmidt, S. Schober, F.J. Hauck, and A. Kassler. Instant-X: Towards a generic API for multimedia middleware. In *Proceedings of the IEEE International Conference on Internet Multimedia Services Architecture and Applications (IMSAA)*, pages 1–6, December 2009.

[eXoa] The extended osip library. [Online]. `http://savannah.nongnu.org/projects/exosip/` [retrieved at 15.6.2011].

[eXob] exosip parser. [Online]. `http://www.antisip.com/doc/osip2/group__howto2__parser.html` [retrieved at 15.6.2011].

[Fab] fabic3. [Online]. `http://www.fabric3.org/overview.html` [retrieved at 15.6.2011].

[FGM$^+$99] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Hypertext Transfer Protocol – HTTP/1.1. RFC 2616, Internet Engineering Task Force, June 1999.

[GHM$^+$07] M. Gudgin, M. Hadley, N. Mendelsohn, Y. Lafon, J.-J. Moreau, A. Karmarkar, and H. F. Nielsen. SOAP Version 1.2 Part 1: Messaging Framework (Second Edition). W3C recommendation, W3C, June 2007.

[GJSB05] J. Gosling, B. Joy, G. Steele, and G. Bracha. *Java(TM) Language Specification, The (3rd Edition) (Java (Addison-Wesley))*. Addison-Wesley Professional, 2005.

[Goo02] B. Goode. Voice over Internet protocol (VoIP). In *Proceedings of the IEEE*, volume 90, issue 9, pages 1495–1517, 2002.

[GPL] Gnu general public license, version 2. [Online]. `http://www.gnu.org/licenses/gpl-2.0.html` [retrieved at 15.6.2011].

[GVE00] A. Gulbrandsen, P. Vixie, and L. Esibov. A DNS RR for specifying the location of services (DNS SRV). RFC 2782, Internet Engineering Task Force, February 2000.

[H2209] H.225.0 Call signalling protocols and media stream packetization for packet-based multimedia communication systems. Itu-t recommendation, ITU-T: Telecommunication Standardization Sector of ITU, Place des Nations, 1211 Geneva 20, Switzerland, December 2009.

[H2409] H.245 Control protocol for multimedia communication. Itu-t recommendation, ITU-T: Telecommunication Standardization Sector of ITU, Place des Nations, 1211 Geneva 20, Switzerland, December 2009.

[H3209] H.323 Packet-based multimedia communications systems. Itu-t recommendation, ITU-T: Telecommunication Standardization Sector of ITU, Place des Nations, 1211 Geneva 20, Switzerland, December 2009.

[HB04] H. Haas and A. Brown. Web Services Glossary. W3C note, W3C, February 2004.

[HHP+06] J. C. Han, W. Hyun, S. O. Park, I. J. Lee, M. Y. Huh, and S. G. Kang. An application level gateway for traversal of SIP transaction through NATs. In *Proceedings of the 8th International Conference on Advanced Communication Technology (ICACT)*, volume 3, pages 1649–1652, February 2006.

[HJ98] M. Handley and V. Jacobson. SDP: Session Description Protocol. RFC 2327, Internet Engineering Task Force, April 1998. Obsoleted by RFC 4566, updated by RFC 3266.

[HSSR99] M. Handley, H. Schulzrinne, E. Schooler, and J. Rosenberg. SIP: Session Initiation Protocol. RFC 2543, Internet Engineering Task Force, March 1999. Obsoleted by RFCs 3261, 3262, 3263, 3264, 3265.

[HZ04] M. Hillenbrand and G. Zhang. A Web services based framework for voice over IP. In *Proceedings of the 30th Euromicro Conference (EUROMICRO'04)*, pages 258–264, August - September 2004.

[IBM] Websphere application server v7 feature pack for service component architecture (sca). [Online]. `http://www.ibm.com/software/webservers/appserv/was/featurepacks/sca/features/` [retrieved at 15.6.2011].

[ICT] Institute of computer technology at vienna university of technology. [Online]. `http://www.ict.tuwien.ac.at/` [retrieved at 3.7.2011].

[JDS+03] A. Johnston, S. Donovan, R. Sparks, C. Cunningham, and K. Summers. Session Initiation Protocol (SIP) Basic Call Flow Examples. RFC 3665, Internet Engineering Task Force, December 2003.

[JL06] A. Johnston and O. Levin. Session Initiation Protocol (SIP) Call Control - Conferencing for User Agents. RFC 4579, Internet Engineering Task Force, August 2006.

[JM07] C. Jennings and N. Modadugu. Session Initiation Protocol (SIP) over Datagram Transport Layer Security (DTLS). Rfc, Internet Engineering Task Force, April 2007.

[Jos08] N. Josuttis. *SOA in der Praxis*. Dpunkt.verlag Gmbh, 2008.

[Kam] Webpage of kamailio sip proxy server. [Online]. `http://www.kamailio.org` [retrieved at 15.6.2011].

[KHF06] E. Kohler, M. Handley, and S. Floyd. Datagram Congestion Control Protocol (DCCP). RFC 4340, Internet Engineering Task Force, March 2006. Updated by RFCs 5595, 5596.

[Kle08] J. Klensin. Simple Mail Transfer Protocol. RFC 5321, Internet Engineering Task Force, October 2008.

[KSF07] B. Karpagavinayagam, R. State, and O. Festor. Monitoring Architecture for Lawful Interception in VoIP Networks. In *Proceedings of the Second International Conference on Internet Monitoring and Protection (ICIMP 2007)*, pages 5–10, July 2007.

[LCLL04] F. Liu, W. Chou, L. Li, and J. Li. WSIP - Web service SIP endpoint for converged multimedia/multimodal communication over IP. In *Proceedings of the IEEE International Conference on Web Services (ICWS'04)*, pages 690–697, July 2004.

[MD00] M. Mealling and R. Daniel. The Naming Authority Pointer (NAPTR) DNS Resource Record. RFC 2915, Internet Engineering Task Force, September 2000. Obsoleted by RFCs 3401, 3402, 3403, 3404.

[MSR$^+$03] A. Milanovic, S. Srbljic, I. Raznjevic, D. Sladden, D. Skrobo, and I. Matosevic. Distributed system for lawful interception in VoIP networks. In *Proceedings of the IEEE Region 8 International Conference on Computer as a Tool (EUROCON 2003)*, volume 1, pages 203–207, September 2003.

[Nat03] Y. V. Natis. Service-Oriented Architecture Scenario. SSA Research Note AV-19-6751, Gartner, April 2003.

[OAS] List of technical committees within the oasis open composite services architecture (csa) member section. [Online]. `http://www.oasis-opencsa.org/committees` [retrieved at 15.6.2011].

[Ora] Oracle soa suite. [Online]. `http://www.oracle.com/us/technologies/029118.pdf` [retrieved at 15.6.2011].

[Osg07] OSGi Service Platform Release 4. Osgi specification, OSGi Alliance, Bishop Ranch 6, 2400 Camino Ramon, Suite 375, San Ramon, CA 94583 USA, May 2007.

[OSI] The gnu osip library. [Online]. `http://www.gnu.org/software/osip/` [retrieved at 15.6.2011].

[OSO] Website of the open service oriented architecture collaboration. [Online]. `http://www.osoa.org` [retrieved at 15.6.2011].

[PJSa] Pjsip - open source sip stack. [Online]. `http://www.pjsip.org` [retrieved at 15.6.2011].

[PJSb] Pjsip documentation. [Online]. `http://www.pjsip.org/docs.htm` [retrieved at 15.6.2011].

[Pos80] J. Postel. User Datagram Protocol. RFC 768, Internet Engineering Task Force, August 1980.

[Pos81] J. Postel. Transmission Control Protocol. RFC 793, Internet Engineering Task Force, September 1981. Updated by RFCs 1122, 3168.

[Pro02] J. Prosise. *Programming Microsoft .NET (Core reference)*. Microsoft Press, 2002.

[PvdH07] M. P. Papazoglou and W.-J. van den Heuvel. Service oriented architectures: approaches, technologies and research issues. In *the VLDB Journal*, volume 16, issue 3, pages 389–415, March 2007.

[Res00] E. Rescorla. HTTP Over TLS. RFC 2818, Internet Engineering Task Force, May 2000. Updated by RFC 5785.

[RM06] E. Rescorla and N. Modadugu. Datagram Transport Layer Security. RFC 4347, Internet Engineering Task Force, April 2006. Updated by RFC 5746.

[Roa02] A. B. Roach. Session Initiation Protocol (SIP)-Specific Event Notification. RFC 3265, Internet Engineering Task Force, June 2002. Updated by RFCs 5367, 5727.

[Ros04] J. Rosenberg. A Presence Event Package for the Session Initiation Protocol (SIP). RFC 3856, Internet Engineering Task Force, August 2004.

[Ros06] J. Rosenberg. A Framework for Conferencing with the Session Initiation Protocol (SIP). RFC 4353, Internet Engineering Task Force, February 2006.

[RPSC04] J. Rosenberg, J. Peterson, H. Schulzrinne, and G. Camarillo. Best Current Practices for Third Party Call Control (3pcc) in the Session Initiation Protocol (SIP). RFC 3725, Internet Engineering Task Force, April 2004.

[RS02a] J. Rosenberg and H. Schulzrinne. An Offer/Answer Model with Session Description Protocol (SDP). RFC 3264, Internet Engineering Task Force, June 2002.

[RS02b] J. Rosenberg and H. Schulzrinne. Reliability of Provisional Responses in Session Initiation Protocol (SIP). RFC 3262, Internet Engineering Task Force, June 2002.

[RS02c] J. Rosenberg and H. Schulzrinne. Session Initiation Protocol (SIP): Locating SIP Servers. RFC 3263, Internet Engineering Task Force, June 2002.

[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, Internet Engineering Task Force, June 2002. Updated by RFCs 3265, 3853, 4320, 4916, 5393, 5621, 5626, 5630.

[RSC05] J. Rosenberg, H. Schulzrinne, and G. Camarillo. The Stream Control Transmission Protocol (SCTP) as a Transport for the Session Initiation Protocol (SIP). RFC 4168, Internet Engineering Task Force, October 2005.

[RSL06] J. Rosenberg, H. Schulzrinne, and O. Levin. A Session Initiation Protocol (SIP) Event Package for Conference State. RFC 4575, Internet Engineering Task Force, August 2006.

[SCA] Service component architecture specifications. [Online]. `http://www.osoa.org/display/Main/Service+Component+Architecture+Specifications` [retrieved at 15.6.2011].

[SCFJ03] H. Schulzrinne, S. Casner, R. Frederick, and V. Jacobson. RTP: A Transport Protocol for Real-Time Applications. RFC 3550, Internet Engineering Task Force, July 2003. Updated by RFCs 5506, 5761.

[SCO] Scorware. [Online]. `http://www.scorware.org/projects/en` [retrieved at 15.6.2011].

[SDO] Service data objects specifications. [Online]. `http://www.osoa.org/display/Main/Service+Data+Objects+Specifications` [retrieved at 15.6.2011].

[SH99] P. Srisuresh and M. Holdrege. IP Network Address Translator (NAT) Terminology and Considerations. RFC 2663, Internet Engineering Task Force, August 1999.

[SIP] Sipp. [Online]. `http://sipp.sourceforge.net` [retrieved at 15.6.2011].

[SN96] W. R. Schulte and Y. V. Natis. "Service Oriented" Architectures, Part 1 . SSA Research Note SPA-00-7425, Gartner, April 1996.

[Spr] The spring framework. [Online]. `http://www.springsource.org` [retrieved at 15.6.2011].

[SPS04] G. Scheets, M. Parperis, and R. Singh. Voice over the internet: A tutorial discussing problems and solutions associated with alternative transport. In *IEEE Communications Surveys and Tutorials*, volume 6, issue 2, pages 22–31, Second Quarter 2004.

[SXM$^+$00] R. Stewart, Q. Xie, K. Morneault, C. Sharp, H. Schwarzbauer, T. Taylor, I. Rytina, M. Kalla, L. Zhang, and V. Paxson. Stream Control Transmission Protocol. RFC 2960, Internet Engineering Task Force, October 2000. Obsoleted by RFC 4960, updated by RFC 3309.

[Tus] Apache tuscany. [Online]. `http://tuscany.apache.org` [retrieved at 15.6.2011].

[TW07] U. Trick and F. Weber. *SIP, TCP/IP und Telekommunikationsnetze: Next Generation Networks und VoIP-konkret.* Oldenbourg Wissenschaftsverlag, 2007.

[Ubu] Ubuntu linux. [Online]. `http://www.ubuntu.com` [retrieved at 15.6.2011].

[WF04] P. Walmsley and D. C. Fallside. XML schema part 0: Primer second edition. W3C recommendation, W3C, October 2004.

[Wik] Wikipedia article of service-oriented architecture. [Online]. `http://en.wikipedia.org/wiki/Service-oriented_architecture` [retrieved at 15.6.2011].