Dynamic Data Paths in OpenSER as an Application of End User Development

Diploma Thesis

by

Bastian Friedrich

October 2006 - March 2007

Albert Ludwig University of Freiburg

Faculty of Applied Sciences

Chair of Communication Systems

Prof. Dr. Gerhard Schneider

Committee: Prof. Dr. Gerhard Schneider, Prof. Dr. Günter Müller

Core Dumped Blues

Well, my terminal's locked up, and I ain't got any Mail,

And I can't recall the last time that my program didn't fail;

I've got stacks in my structs, I've got arrays in my queues,

I've got the: Segmentation violation – Core dumped blues.

If you think that it's nice that you get what you C,

Then go: illogical statement with your whole family,

'Cause the Supreme Court ain't the only place with: Bus error views.

I've got the: Segmentation violation – Core dumped blues.

On a PDP-11, life should be a breeze,

But with VAXen in the house even magnetic tapes would freeze.

Now you might think that unlike VAXen I'd know who I abuse,

I've got the: Segmentation violation – Core dumped blues.

(Greg Boyd, 1980)

Erklärung

Hiermit erkläre ich, dass ich diese Abschlussarbeit selbständig verfasst habe, keine anderen als die angegebenen Quellen/Hilfsmittel verwendet habe und alle Stellen, die wörtlich oder sinngemäß aus veröffentlichten Schriften entnommen wurden, als solche kenntlich gemacht habe. Darüber hinaus erkläre ich, dass diese Abschlussarbeit nicht, auch nicht auszugsweise, bereits für eine andere Prüfung angefertigt wurde.

Freiburg, 26. März 2007

. . . .

Bastian Friedrich

Danksagungen

Für die Unterstützung bei meiner Diplomarbeit möchte ich Herrn Professor Dr. Gerhard Schneider recht herzlich danken. Er hat mich in seiner Arbeitsgruppe herzlich aufgenommen. Fachlichen und inhaltlichen Rat erhielt ich zu jedem Zeitpunkt von Herrn Dirk von Suchodoletz, wofür ich ihm ebenfalls meinen Dank aussprechen möchte.

Für die Dauer der Diplomarbeit wurde ich freundlich von der Firma Collax GmbH aufgenommen und erhielt alle nur erdenkliche Unterstützung. Besonders bedanken möchte ich mich beim CEO und CTO des Unternehmens, Herrn Boris Nalbach, der für meine Sorgen und Probleme immer ein offenes Ohr hatte, sowie bei Herrn Andreas Hofmeister, der mir zu jeder Zeit mit Rat und Tat zur Seite stand.

Mein ganz besonderer Dank gilt meiner lieben Kirsten, die mir auf dem Weg zur vorliegenden Arbeit und in jeder anderen Situation meines Lebens liebevoll zur Seite stand, mir Mut machte, wenn nicht alles nach Plan lief, und sich mit mir freute, wenn ich Erfolg hatte.

Meinen Eltern möchte ich für die Geduld danken, die sie mit mir in den vergangenen Jahren hatten, und für die Unterstützung, die sie mir auf vielerlei Art zukommen ließen.

Viele Ungenannte haben mich in den letzten Monaten und Jahren begleitet. Auch wenn Ihr hier nicht genannt werdet: ich werd's Euch nicht vergessen!

Zusammenfassung (Deutsch)

OpenSER ist eine leistungsfähige Open-Source-Lösung, die praktisch sämtliche im SIP-Protokoll definierte serverseitige Funktionalitäten abdeckt. In vielen großen SIP-Umgebungen sind der SIP Express Router SER oder dessen Abkömmling OpenSER im Einsatz.

Die internen Schnittstellen der Software, über die Daten aus Datenbanken bezogen werden, sind strukturell auf ein relationales Modell ausgerichtet; existierende Anbindungen beschränken sich auf MySQL, PostgreSQL sowie die ODBC-Schnittstelle. In vielen Umgebungen stehen äquivalente Daten jedoch in anderer Form, vor allem in LDAP-Verzeichnissen, zur Verfügung.

Aufgabe der Arbeit war, eine Datenbankschnittstelle zu entwickeln, die es dem existierenden OpenSER-Code ermöglicht, auf beliebig konfigurierbare Daten-Backends zurückzugreifen. Dazu ist jedoch nicht nur eine strukturelle, sondern in den meisten Fällen auch eine inhaltliche Transformation von Daten notwendig.

Aus diesem Grunde wurde OpenSER mit einer Perl-Schnittstelle ausgestattet. Eine fundamentale Infrastrukturschicht, die auch losgelöst von den Datenbank-Aufgaben vielfältig genutzt werden kann, wird über die entwickelte „Perl virtual database" (PerlVDB) zum Übertragungsweg für die Daten; Adaptoren verwandeln relationale „insert"-, „update"-, „delete"- und „query"-Anfragen in Funktionsaufrufe für Perl.

Ein Exkurs in das Thema End User Development abstrahiert den Vorgang der Einbettung von Programmiersprachen. In dieser Arbeit werden vor allem technische Aspekte des End User Development diskutiert und die gefundenen Antworten auf die konkrete Situation im OpenSER übertragen.

Die Thematik wurde in den üblichen Schritten der Softwaretechnik bearbeitet. In der Analyse-Phase wurden u.A. mittels Anwendungsfällen die anfallenden Daten untersucht und anschließend kategorisiert. Während des Designs wurden die notwendigen Schnittstellen etabliert und eine Klassenhierarchie entwickelt. In der vorliegenden Arbeit werden auch die Implementierung der entwickelten Lösung sowie die dabei auftretenden Probleme dokumentiert.

Abstract (English)

OpenSER is a powerful open source solution that provides almost all server functionalities defined in the SIP protocol. The SIP Express Router SER or its fork OpenSER are in action in many large SIP environments.

The internal interfaces of the software that transmit data from databases are structurally dependent on a relational model; existing back-ends are restricted to MySQL, PostgreSQL and the ODBC interfaces. On the other hand, in many real world environments equivalent data are stored in other forms, particularly in LDAP directories.

The task of this thesis was the development of a database interface that makes it possible for existing OpenSER code to access arbitrary, configurable data back-ends. However, to accomplish this, not only a structural transformation, but in most cases also a transformation of content is necessary.

Due to this, OpenSER was equipped with a Perl interface. A fundamental infrastructure layer that may also be used in multiple ways independent of database requests becomes a transmission route for data through the separately developed "Perl virtual database" (Perl VDB); adaptors transform the relational "insert", "update", "delete" and "query" operations into Perl function calls.

A digression to the topic of End User Development abstracts the process of embedding a programming language. In this thesis, mainly technical aspects of End User Development are discussed. The answers found are then transferred to the particular situation in OpenSER.

The theme was examined with the usual steps of software engineering. In the analysis phase, use cases and other techniques were applied to evaluate and categorize the incurring data. During the design, necessary interfaces were established and a class hierarchy was developed. In the present work, the implementation of the developed solutions as well as the arising problems are documented.

Contents

1. Introduction **1**

1.1. From telephony... 1

1.2. ... to Voice over IP 1

1.3. About this work 2

1.3.1. Objectives 2

1.3.2. Structure of this thesis 3

1.3.3. References and related work 3

I. VoIP technology **5**

2. Telephony and VoIP **7**

2.1. Telephony 7

2.2. A brave new VoIP world 8

2.3. The SIP protocol 9

2.4. Open source technology 12

2.4.1. Asterisk 12

2.4.2. OpenSER 13

2.4.3. Other products 13

3. OpenSER **15**

3.1. What is OpenSER? 15

3.2. OpenSER from a user's perspective 16

3.2.1. The OpenSER configuration 16

3.2.2. A sample routing block 16

3.2.3. Database bindings 18

3.2.4. Pseudo variables and AVPs 19

3.2.5. In a nutshell... 20

3.3. OpenSER from a code perspective 21
3.3.1. Types of modules 22
3.3.2. Export structure 22
3.3.3. Parameter fixup 23
3.3.4. In a nutshell... 24

4. Data handling 27
4.1. About data and information 27
4.1.1. Distinguishing data and information 27
4.1.2. Types of information 28
4.1.2.1. Configuration data 28
4.1.2.2. Transitional data 28
4.1.2.3. Authentication and authorization 29
4.2. Technologies 29
4.2.1. The simple life: plain text 30
4.2.2. Relational databases and SQL 30
4.2.3. Directory services: LDAP 31
4.2.4. Authentication services 32
4.2.4.1. RADIUS and DIAMETER 33
4.2.4.2. Kerberos 34
4.2.5. ENUM 34
4.2.5.1. Status quo 35

II. Analysis and specification 37

5. Requirements analysis 39
5.1. Use case analysis 39
5.1.1. Use case: Incoming phone call 41
5.1.2. Use case: Outgoing phone call 42
5.1.3. Use case: Call transfer (External: VoIP/POTS, internal: VoIP) . 43
5.1.4. Use case: Call deflection 43
5.1.5. Use case: conference calls (internal, external, mixed) 44
5.1.6. Use case: SIP registration (attaching a phone) and authentication 44
5.1.7. Use case: user and authorization/permission management 45
5.1.8. Use case: personal self-administration 46

5.1.9. Use case: Accounting 47
5.1.10. Use case: ACD, IVR, Call hunt groups 47
5.2. Categorizing data 47
5.3. Data handling in OpenSER 48
5.3.1. Data paths in OpenSER 51
5.3.2. Data management bindings in OpenSER 52
5.4. Refining the requirements 53

6. Specification 55
6.1. A flexible database backend for OpenSER 55
6.2. Digression: configuring vs. programming 56
6.3. Functions in OpenSER modules 58
6.4. Expressing a data path 60
6.4.1. Possible tradeoffs 60
6.4.2. Evaluating the options 62

III. End User Development 65

7. End User Development in dynamic systems 67
7.1. Embeddable languages 68
7.1.1. Perl 68
7.1.2. PHP 69
7.1.3. Lua 69
7.1.4. Python 69
7.1.5. Ruby 70
7.1.6. LISP and Scheme 70
7.1.7. Basic dialects 70
7.1.8. Other languages 71
7.2. Analyzing EUD implementations 71
7.2.1. Apache and mod_perl 71
7.2.1.1. Analyzing the framework 72
7.2.2. Office suites and macro languages 73
7.2.2.1. Technical aspects of Visual Basic for Applications 74
7.2.3. Gimp and Scheme 74
7.3. Properties of EUD environments 74

7.3.1. Interfaces 75
7.3.2. Design constraints 76

IV. Design, Implementation, Testing 77

8. Design 79
8.1. Considerations on an EUD environment for OpenSER 79
8.1.1. Choosing a language 80
8.2. Data paths 82
8.3. Design patterns 83
8.3.1. The bridge pattern 83
8.3.2. The adapter pattern 84
8.4. A Perl module 84
8.5. Virtual database 86
8.5.1. Class structure 87
8.5.2. Adapters 89

9. Implementation 91
9.1. Tools 91
9.1.1. Build environment 91
9.1.2. Revision control 92
9.2. Development process 92
9.3. Embedding Perl 93
9.3.1. The Perl interpreter 94
9.3.2. Data types in Perl 94
9.3.3. Perl memory management 95
9.4. Perl module 95
9.4.1. The module itself 96
9.4.1.1. The “dlopen problem” 97
9.4.1.2. The “reload problem” 98
9.4.1.3. The “global variable problem” 99
9.4.2. The Perl extension 99
9.4.2.1. The “constants problem” 101
9.4.2.2. The “fixup problem” 102
9.4.3. The Perl library 104

9.4.4. Documentation . . . 106
9.5. perlvdb module . . . 107
9.5.1. Module interface . . . 108
9.5.2. Database access functions . . . 108
9.5.3. Data transformation . . . 109
9.5.4. Perl classes and adapters . . . 110

10. Testing 111
10.1. Debugging OpenSER . . . 111
10.2. Testing environment . . . 112
10.2.1. sipsak . . . 112
10.2.2. SIPp . . . 112
10.2.3. Computer hardware . . . 113
10.2.4. Benchmark module . . . 113
10.3. Test cases . . . 113
10.4. Testing procedure and results . . . 115
10.4.1. Stress testing and performance evaluation . . . 115
10.4.2. Invalid messages . . . 117
10.4.3. Invalid Perl code . . . 117
10.4.4. Regression tests and coverage analysis . . . 118

V. Discussion 119

11. Discussion and Conclusion 121
11.1. Revisiting Use Cases . . . 121
11.2. Specific solutions . . . 124
11.2.1. Aliases . . . 124
11.2.1.1. alias_db and perlvdb . . . 125
11.2.1.2. Aliasing in Perl . . . 126
11.2.2. Authentication . . . 127
11.2.3. Accounting . . . 127
11.2.4. Authorization . . . 128
11.3. Perl vs. 130
11.3.1. Perl vs. VDB . . . 130
11.3.1.1. VDB usage . . . 131

11.3.2. Perl vs. SEAS 133
11.3.3. Perl and VDB vs. ldap modules 134
11.3.4. Perl vs. SIP-CGI 134
11.4. The VDB approach 134
11.5. Perspective 135

VI. Appendix 137

A. Accompanying CD and website 139

B. Glossary 141

C. Tools 145

Bibliography 147

1. Introduction

1.1. From telephony...

After the invention of the telephone by A. G. Bell in the 1870s, there were no great changes for a century: A microphone and a speaker, connected with a copper wire, transported speech signals from one device to another. But then a hundred years later, telephone carriers began to migrate from analog to digital signaling, ISDN was introduced, convenience functions were added, and voice quality was increased. Mobile telephony began to conquer the world. Besides that, everything remained the same.

The end of the 20th century brought another substantial change in communication systems: The Internet developed from a scientific testbed to a global main stream communication method. Although a packet switched network such as the Internet introduces a number of new problems when transporting multimedia and voice data, it opens a whole new world of features and properties for users. Around 2004, Internet telephony (Voice over Internet Protocol, VoIP) began to replace conventional telephony.

1.2. ... to Voice over IP

While end users are supplied with cheap multi function devices that provide for a smooth transition from conventional telephony to VoIP, enterprise scale setups can either be realized with – often expensive – commercial hardware or on the basis of open source software.

The recent years have brought the second large VoIP wave. The first wave of Internet telephony – driven mainly by residential users and technology enthusiasts – led to implementations that were using non-standardized proprietary protocols or the now more or less deprecated H.323 protocol. These products were not capable of fulfilling the users' needs and reliability requirements: the speech quality was considerably lower than in traditional telephony, communication was half duplex, or the phone calls failed after

some time. Today, modern, SIP based systems can in fact replace standard telephony networks and are also fulfilling enterprise requirements for reliability and service quality.

There now is a considerable number of software projects targeting most aspects of VoIP communication. Two of these projects are the SIP Express Router SER and its fork OpenSER, the latter being the primary target of this work.

1.3. About this work

Commercial and open source products provide a foundation for the establishment of VoIP services. Nonetheless, a full featured VoIP network requires a large amount of setup and maintenance effort. Many products – including OpenSER – require a fine grained tuning of their environments.

Munich based Collax GmbH has been looking into the integration of a VoIP system with their technology platform, a Linux distribution dedicated to the needs of the small and medium size business (SMB) market. While evaluating available solutions, it became apparent that the integration of OpenSER with a long-established open source based environment raises a number of problems, one of which is the migration of user identity information between the platform's LDAP service and OpenSER's relational user database. This work aims to find a solution for this problem.

Work on the thesis started on 27th of September 2006. This document was submitted on 27th of March 2007.

1.3.1. Objectives

While the integration of OpenSER with LDAP environments may be a very interesting concept from a commercial point of view, the topic of data access will be examined and discussed on the background of various technologies. OpenSER's data access layer will be extended so that accessing arbitrary data sources is possible. By describing a few real world cases, the helpfulness of the solutions found will be proved. Among these examples will be the core functionalities of authentication and authorization mechanisms of OpenSER as well as the ubiquitous aliasing mechanisms that make up an important part of a SIP proxy.

1.3.2. Structure of this thesis

After the discussion of the fundamental technologies and OpenSER as a software product, the software implementation used to solve the described integration problems is developed with standard software engineering techniques. After refining the requirements, a solution is specified. A separate part about End User Development abstracts the ideas developed during the specification.

The implementation of the concepts will be discussed in detail. A discussion of the provided implementations will complete this work, providing a basis for future elaboration on the topic.

The developed code and other documentation is available on the accompanying CD. See Appendix A on page 139 for details.

1.3.3. References and related work

The literature used for this thesis includes a wide variety of different topics, as the subjects of interest are spread over a wide area. The bibliography thus can be divided into the following sub-categories:

- Internet drafts, Requests For Comments (RFCs) and VoIP standards
- VoIP basics, technology and software solutions
- Database, authentication, authorization and accounting technologies
- End User Development and programming languages
- Perl literature and technical documentation

Due to the practical approach of this thesis, a large part of the references is non-academic literature.

The topic of End User Development is a rather new approach in software engineering. The textbook “End User Development” edited by Lieberman, Paternò and Wulf [35] provides a comprehensive analysis of the topic.

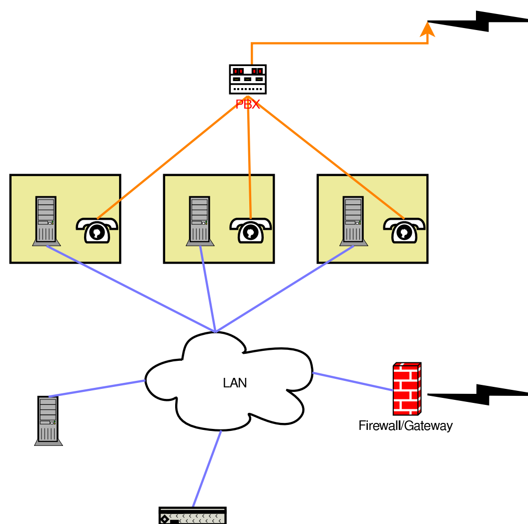
While numerous books deal with various aspects of Voice over IP, the German textbook “Internet-Telefonie. VoIP mit Asterisk und SER” [16] by Flaig, Hoffmann and Langauf provides a structural and technical introduction to the deployment of VoIP networks based on the most important open source products.

Common features of traditional telephony networks can be gathered from feature descriptions of network carriers ([54, 48]) and from textbooks about communication ([13, pp. 288ff], [27, pp. 165ff], [22, pp. 36ff]), while feature descriptions of VoIP networks can be found in [28, pp. 55ff] and [24].

Part I.

VoIP technology

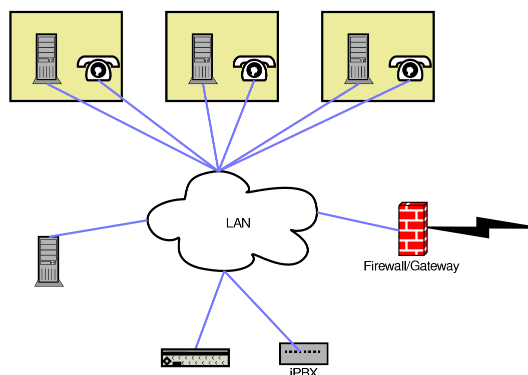
2. Telephony and VoIP



In common enterprise size communication systems, two parallel networks are deployed: an IP based computer network is complemented by a telephony system.

Gateways, firewalls and other infrastructure devices oppose telephone exchanges (PBX) that basically provide a similar functionality for different media. This duality leads to a doubled infrastructure investment, two parallel points of failure, and two separate administrative instances – especially with regards to user management. Applications live in either world, synergy effects from the integration of these networks do not exist[1].

Development in the past decade has created the opportunity for solving these problems. Voice over IP technologies have initiated a movement towards converging networks that unite the telephony and IP data flows in a single media network.



Not only does this save organizations from establishing (and buying) separate network infrastructure while reducing administrative costs; value is added by new features that are only made possible by this integration.

2.1. Telephony

Conventional telephony, often refered to as POTS (Plain Old Telephone System) or PSTN (Public Switched Telephone Network), reserves a dedicated line for every communication (circuit switched network). With the ISDN (Integrated Services Digital

[1] Modern specialized, expensive PBX hardware may provide access vectors for similarly specialized and similarly expensive proprietary software components to accomplish a certain amount of integration. These technologies are far from being standardized. Interoperability between different systems is non-existent.

Network), out-of-band signaling was introduced, so that signals for ringing, hanging up etc. are transmitted over a different logical channel.

The Internet Protocol IP provides a packet switched network. A shared medium is used for multiple parallel communication streams. Based on the IP protocol, a number of dedicated telephony protocols have evolved. During the last years, the Session Initiation Protocol SIP and its relatives have taken over market leadership and are regarded by experts to be the most important VoIP protocol for the next years. This thesis is going to deal with products that are based on the SIP protocol group. These protocols will be described in more detail below.

2.2. A brave new VoIP world

While numerous commercial as well as free products have emerged into the market during the last years, development is still in its beginning. Simple telephony – dialing a number, letting a telephone device ring, making a call and hanging up – is well possible, but neither the comfortable features of modern POTS networks that customers have already got used to nor the added features of network integration are easily accessible.

Choosing the right equipment, the deployment of a VoIP system not only can recreate the features of a traditional telephony network (these features can be found in the literature, see section 1.3.3). The integration of multimedia and Internet communication networks can also add to their respective values by creating a world of new usage possibilities and thus raises the users' productivity. A number of these new features are described in [28, pp. 19ff, pp 53ff]. While many of these features deal with the buzz word "Unified Messaging" (UM), making the aggregation of multiple user communication channels possible, the term "Computer Telephony Integration" (CTI) refers to the convergence of information available for call center and help desk employees.

During the evolution of modern Voice over IP networks, a number of networking standards have been developed. The first important standard, developed by the ITU-T (International Telephony Union, Telecommunication Standardization Unit) in the mid 1990s, was H.323. It was dedicated to create a layer on top of IP that lets other ITU-T protocols work on top of it. H.323 closely resembles the ISDN protocols.

The next notable standard was the Session Initiation Protocol (SIP). While H.323 was developed by telephony specialists, SIP was specified by the Internet specialists of the IETF (Internet Engineering Task Force) in 1999. Within a short time, SIP became the most important VoIP protocol, although H.323 still maintains a market share.

A number of other protocols have since been established. The most prominent system is the commercial Skype, originally a softphone with extraordinary ease of use. Today, a small number of Skype hardphones is available as well. As the Skype protocol is proprietary, obfuscated and not an open standard, only a small number of licensed products use this protocol. These products do not include local servers, so Skype is not a good solution for the SMB market and networks deployed therein.

The open source instant messaging protocol Jabber led to the development of extensions such as Jingle, which is used by Google's "Google Talk" software as a multimedia transport protocol. Today, Jingle is not regarded to be of importance in global VoIP networks.

2.3. The SIP protocol

This section will discuss the basis of the software solutions examined, the Session Initiation Protocol (SIP). As a deep understanding of the protocol and its relatives will not be necessary for this work, details will be omitted. A good explanation of all protocols can be found in [16].

When talking about SIP telephony, in fact a whole family of protocols is concerned. The term "SIP based telephony" is thus more appropriate. The Session Initiation Protocol is responsible for the establishment of a communication channel only.

Real world SIP environments use these protocols[2]:

- SIP [46] – Session initiation. Handles the signaling of telephony.
- SDP [23] – Session Description Protocol. Negotiates the parameters of the communication, such as media codecs.
- RTP [50] – Realtime Transport Protocol. Transports the media streams between user agents.
- RTCP – Realtime Transport Control Protocol. In fact part of the RTP (and specified in the same RFC). Signals timing information between user agents for jitter and bandwidth control. Not all user agents implement RTCP.

[2]Theoretically, the protocols mentioned could be used in other contexts or in conjunction with other protocols. This is not commonly practiced, however.

The SIP protocol is a rather slim text-based protocol. Its grammar and syntax bears a resemblance to the Hyper Text Transfer Protocol, HTTP. SIP messages may be exchanged over UDP or TCP, possibly encrypted via TLS; the UDP transport is the most common underlying layer, TCP is not implemented in all products.

SIP is based on a request/response model. There are only two fundamental types of agents in the model, which are user agents and servers (referred to as "proxies" in SIP). Despite the name, "user agents" in the SIP perspective may be server programs, such as media servers (e.g. providing "music on hold"), back-to-back user agents (b2bua, may e.g. be used for call exchange or conference rooms) or media gateways that mediate between SIP telephony and other technologies.

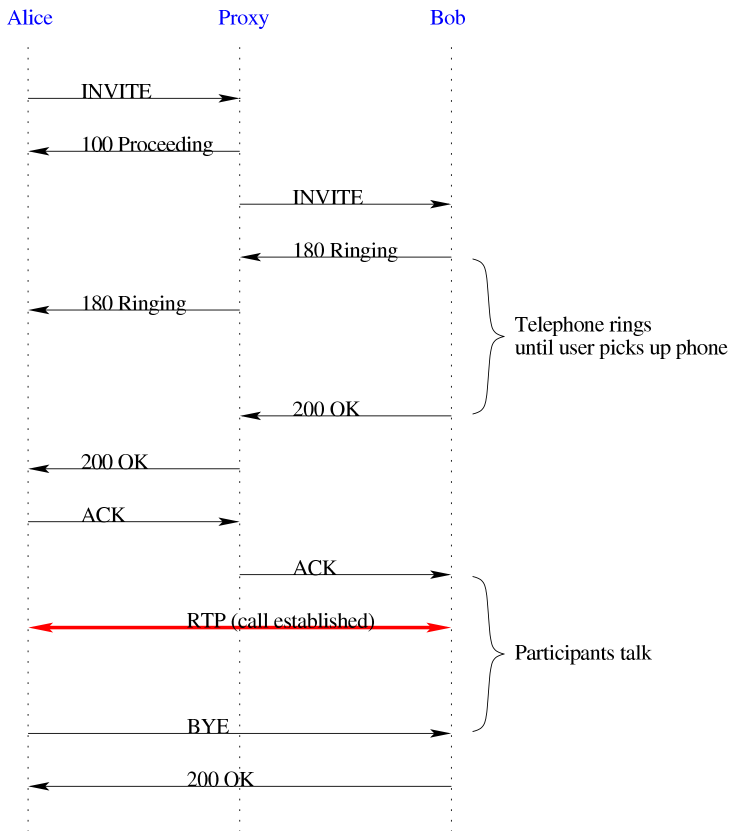
During a call, user agents are categorized as User Agent Clients (UAC) that send a request, and User Agent Servers (UAS) that react on the request.

While user agents send and receive SIP messages, servers forward and possibly modify messages according to their configuration – thus their name "proxy".

SIP messages consist of a request or response line, a header, and (possibly) a message body. A request line starts with a *method*, denoting the requested service. The most important methods are REGISTER, INVITE, ACK, BYE and CANCEL. Each request is destined to a recipient URI (uniform resource identifier), referred to as the message RURI. A RURI often specifies a user SIP address, but as well may describe a destination host.

Every request should be answered by a response, including a return code and a description.

The following image shows a minimal message flow for a phone call from Alice to Bob:



The image should be self-explanatory. Alice's INVITE message to the proxy could look like this:

```
1  INVITE sip:bob@server.de SIP/2.0
2  Via: SIP/2.0/UDP 172.16.3.247:2051;branch=z9hG4bK-8z1ns0sfgqdz;rport
3  From: "alice" <sip:alice@server.de>;tag=6pwp0e5bcq
4  To: <sip:bob@server.de>
5  Call-ID: 3c26b281afc8-owtmk2f6oogz@snom360-000413231835
6  CSeq: 1 INVITE
7  Max-Forwards: 70
8  Contact: <sip:alice@172.16.3.247:2051;line=fazh8jjv>;flow-id=1
9  P-Key-Flags: resolution="31x13", keys="4"
10 User-Agent: snom360/6.2.3
11 Accept: application/sdp
12 Allow: INVITE, ACK, CANCEL, BYE, REFER, OPTIONS, NOTIFY, SUBSCRIBE, PRACK, MESSAGE, INFO
13 Allow-Events: talk, hold, refer
14 Supported: timer, 100rel, replaces, callerid
15 Session-Expires: 3600;refresher=uas
16 Min-SE: 90
17 Content-Type: application/sdp
18 Content-Length: 475
19
20 v=0
21 o=root 314438619 314438619 IN IP4 172.16.3.247
22 s=call
23 c=IN IP4 172.16.3.247
```

```
24 t=0 0
25 m=audio 63950 RTP/AVP 0 8 9 2 3 18 4 101
26 a=crypto:1 AES_CM_128_HMAC_SHA1_32 inline:nMmx+dq9G8h0lmkuQ9LujqR1XVn37yrv11uGk43N
27 a=rtpmap:0 pcmu/8000
28 a=rtpmap:8 pcma/8000
29 a=rtpmap:9 g722/8000
30 a=rtpmap:2 g726-32/8000
31 a=rtpmap:3 gsm/8000
32 a=rtpmap:18 g729/8000
33 a=rtpmap:4 g723/8000
34 a=rtpmap:101 telephone-event/8000
35 a=fmtp:101 0-16
36 a=ptime:20
37 a=encryption:optional
38 a=sendrecv
```

Line number 1 is the request line, beginning with the method (INVITE) and the callee to be contacted. Lines 2 through 18 consist of the SIP headers; a number of these headers are obligatory (e.g. Via, From, To), others are not (e.g. User-Agent, Min-SE). Starting with line 20, the message body follows. This body is part of the SDP protocol communication. Each line defines properties about the expected communication, such as the local RTP end point (line 23) or the expected audio (line 25) codecs (lines 27 through 33).

2.4. Open source technology

Numerous open source software projects provide products for multiple aspects of Internet telephony. In this section, two enterprise grade products will be introduced which are commonly deployed in VoIP environments.

2.4.1. Asterisk

Asterisk probably is the best known VoIP software on the market. As a media gateway, it understands the most important VoIP protocols, such as SIP, H.323, Jingle, ISDN, its proprietary IAX, and more. Although Asterisk can be used in a pure SIP environment, it focuses on the mediation of different protocols. Its most commonly used feature probably is the connection of one or more POTS or ISDN lines with a local SIP environment.

As a dedicated media gateway, Asterisk intercepts not only the SIP traffic as such, but also handles the underlying media streams. This has a negative impact on its performance and scalability.

2.4.2. OpenSER

OpenSER (and its ancestor SIP Express Router, SER) is a dedicated SIP proxy. Based on its registrar and location services, this modular software system can provide all necessary functionalities for large scale SIP environments. OpenSER or SER are used by numerous SIP providers, such as the German companies 1&1, Arcor, Freenet and sipgate.

In conventional VoIP environments, every user agent registers itself with a server, using the REGISTER SIP message method. The server requests and then checks the agent's username and password and stores its contact information (IP address, port, ...) in its location database. Although location and registrar services may theoretically be implemented independently, most systems use a unified server process – as does OpenSER. When a contact request to a specific user is received, the server checks its location database and forwards the incoming message accordingly.

OpenSER provides sophisticated mechanisms to evaluate and modify incoming SIP messages, while it never touches the transported media streams[3]. A detailed description of SER and OpenSER can be found in [16], while the next chapter will describe central elements of the project that will be of interest for this thesis.

2.4.3. Other products

The open source market provides a large number of server (and client) software for SIP telephony [6], but none of the other SIP Proxies provides comparable levels of performance and functionality to SER's and OpenSER's.

The VOCAL server provides additional b2bua support and thus is able to mediate to H.323 and other protocols, but its development seems to have stagnated.

SipX is – similar to Asterisk – a semi commercial solution. An open source edition is available for the community from SIPfoundry.org, while commercial products including maintenance and support may be purchased from the company Pingtel. SipX has been rapidly growing during the last years and now provides a full featured PBX for SMBs. According to SIPfoundry's web site, SipX has been deployed in environments with over 5000 users. It includes b2bua features not unlike Asterisk's such as auto attendants for multiple scenarios and thus plays a completely different role in a SIP environment than OpenSER does.

[3]The rtpproxy module provides proxying for RTP streams, which may be necessary for NAT traversal. The multimedia stream is not modified by the module.

3. OpenSER

To understand the details of the necessary work, it is crucial to understand how OpenSER works as a piece of software. After a description of the project's purpose, two orthogonal views of the program will be plotted in this chapter.

3.1. What is OpenSER?

OpenSER is a full featured SIP server. Based on the core proxy functionalities, it features integrated location and registrar servers for stateful and stateless SIP routing. Each of these functionalities is provided by a distinct module. Additional modules add functionalities in multiple areas: Some add core SIP functionality, others provide bridges to other operative areas such as a Jabber interface for instant messaging. Administrators are given tools for accounting and security hardening.

The development of OpenSER's ancestor SER was initiated at the Fraunhofer Institut für offene Kommunikationssysteme (FOKUS) in Berlin in 2001 and later continued by the start-up enterprise iptelorg GmbH. This company was acquired by Tekelec in August 2005. The software was released under the GPL. Due to controversies regarding release policy and code progress, the project was forked in 2005 by the two core developers Bogdan-Andrei Iancu and Daniel-Constantin Mierla together with the SER contributor Elena-Ramona Modroiu. This fork was based on IPTel's SER 0.9.3. In June 2005, OpenSER 0.9.4 was released.

In June 2006, OpenSER version 1.1.0 was released. This version provided the first basis for the implementation work of this thesis. Shortly before the submission date of this work, version 1.2.0 was released which contains one part of the code developed for this thesis. The statistics found below result from studies of the development branch between the 1.1.1 and the 1.2.0 release.

OpenSER development now is led by the Romanian company Voice System under the management of the fork's initiators. Voice System earns its revenue with development

and consulting jobs related to OpenSER and VoIP setups. Throughout the work on this thesis, the author frequently had contact with the OpenSER developer team.

3.2. OpenSER from a user's perspective

Installing OpenSER as a user leads to four distinct entities that are located in the file system: The program itself including control scripts, a number of modules, the configuration directory and an SQL database schema.

3.2.1. The OpenSER configuration

OpenSER's behavior is defined in a single configuration file. Despite the name "configuration", its syntax and semantics are very similar to a "program"[1]. In a rough outline, the configuration file consists of four sections [49]:

Global parameters. This section sets global parameters, such as the hostname, IP addresses and debugging levels.

Modules. The "`loadmodule`" statement loads the numerous available OpenSER modules. The modules developed for this thesis are also loaded here.

Module parameters. Modules can define variables that may be set in the configuration file. This is commonly used e.g. for the database URI to be used by the module, or for Boolean flags to adjust the module's behavior.

Routing blocks The routing blocks are the core of the OpenSER configuration. The statements in the blocks are similar to a normal program: conditions are checked with if-statements, functions are called, and their return values are evaluated.

The functions called from these blocks are implemented (and exported for user access) either by the OpenSER core, or by the loaded modules.

3.2.2. A sample routing block

To convey a sense of how OpenSER's configuration works, the following (non-functional) routing block excerpt will briefly be discussed:

[1] See section 6.2 on page 56 for a distinction between the terms "configuration" and "program"

```
route{
        acc_db_request("foo", "acc");

        if (!uri==myself) {
                append_hf("P-hint: outbound\r\n");
                route(1);
        };

        if (uri==myself) {
                if (method=="REGISTER") {                                          10
                        if (!www_authorize("sampledomain.de", "subscriber")) {
                                www_challenge("sampledomain.de", "0");
                                exit;
                        };
                        save("location");
                        exit;
                };

                if (!lookup("location")) {
                        sl_send_reply("404", "Not Found");                         20
                        exit;
                };
                append_hf("P-hint: usrloc applied\r\n");
                setflag(1);
        };
        route(1);
}
```

Line 1 An OpenSER configuration file may hold multiple routing blocks. This one is the top level request routing block, as it is not described by an additional feature. Routing blocks may be request, failure, reply or branch routing blocks.

Line 2 Every message transiting the system is passed to the accounting subsystem. The "`acc`" module contains – among others – a function `acc_db_request` that stores information about SIP messages in a database. The first parameter is an arbitrary string that is included in the log; the second parameter indicates the SQL table to store the information in. The definition of the database itself is done in the module parameters block.

Line 4 Messages that are not destined to this server instance are flagged, and then directly passed to the routing block number one (not shown here).

Line 10 If the current message is a registration message, its credentials are checked. If they are absent or invalid, correct credentials are requested and the processing of the message is interrupted.

If the credentials are valid, the registration is stored in the location database.

Line 19 For all other local messages, the location of the user is looked up in the user location database. If the user does not exist, an error message is sent back. Otherwise, routing block number one is entered which eventually delivers the message to the registered end point.

As seen in this example, there is a wide range of functions available. The accounting function, header manipulation, user location database including the registration mechanism – all of these are part of different modules.

Each of these functions can take between zero and two parameters. Inside the OpenSER script, these parameters have to be strings (marked by the surrounding quotation marks); in case of the `sl_send_reply` function, the numerical value 404 is written as the string "404". The reason that other "calls" in the script use a numerical value instead of a string (such as "`setflag`" or "`route`" in the sample above) is that these functions are core functions rather than module functions.

A list of available modules and their functions is available on the OpenSER web site. The flexibility and power provided by the configuration environment and the module functions enables the user to create a wide range of different functionalities. On the downside, it is not easy to create correctly working, RFC-compliant scripts. Because of that, the SER team has started to work on providing template scripts for the most common use cases. It is likely that most of these scripts will be easily adaptable for use with OpenSER.

3.2.3. Database bindings

OpenSER features the following relational database engine implementations:

mysql is probably the most widely used engine. Although MySQL lacks features crucial in many enterprise sized relational database systems, it is extremely fast and has proven its stability in many contexts.

postgres provides an interface to the open source database management system PostgreSQL (often referred to by its old name Postgres). PostgreSQL provides more functionality than MySQL, but on the other hand is not as easy to maintain and to use.

unixodbc implements a binding to the ODBC interface of unixODBC, a project aiming to provide a technology-independent database layer for non-windows systems[5]. unixODBC supports numerous enterprise scale as well as light weight database backends.

flatstore is a slim pseudo-database back-end that only supports write operations. Data are written to a human-readable and computer-parsable (character separated values, CSV) text file. Developed solely for use with the accounting module `acc`, it might also be used with other write-only modules such as `siptrace`.

dbtext provides another back-end for text files. Unlike flatstore, it does not only support write operations, but also queries, updates and deletes. It is intended for test environments and small installations where full-featured SQL databases are not available.

Anticipating the categorization of data in the next chapter, non-relational data access is available for – among others – RADIUS servers and the ENUM DNS zone.

3.2.4. Pseudo variables and AVPs

In OpenSER's configuration, so-called pseudo variables can help setting up more complex environments. These pseudo variables may be used in the configuration script and contain information about the message that is currently processed: the variable `$ru`, for example, holds the recipient URI of the message, while `$Ri` represents the IP address the message was received on. Additionally to the core pseudo variables, modules may export their own.

A full list of all available pseudo variables is available in the OpenSER documentation.

In the configuration file, e.g. equivalence of these pseudo variables with static data or other variables may be checked to decide on further behavior. One could imagine checking the source IP address `$si` to blacklist a certain user:

```
if ($si == "127.0.0.1") {
        sl_send_reply("401", "No local calls.");
        exit;
}
```

Additionally to the configuration statements, numerous modules provide functions that "understand" pseudo variables as well. One example of these functions is the `xlog()` function provided by the `xlog` module. This function can be used to log messages that include information about the currently processed SIP message.

Closely related to the pseudo variables (in fact being a special type of pseudo variables) are the attribute/value pairs, AVPs. These variables may be stored in or fetched from a database, or from a RADIUS server through the `avpops` and `avp_radius` modules. An interesting example of using AVPs can be found in OpenSER's RADIUS documentation [39]. In this sample, the RADIUS entries include AVPs containing a start and end time during which users may initiate calls.

Pseudo variables and AVPs are crucial instruments for complex OpenSER configurations. In simple environments, their usage is rarely necessary.

3.2.5. In a nutshell...

OpenSER is an extremely flexible and powerful SIP proxy. By its flexible and expressive configuration language, the system can be adapted to a wide variety of environments. Users have installed OpenSER instances in environments ranging from single user to hundreds of thousands of users. The sample configurations that are available in the package and on various Internet sites help to overcome common problems. The configuration wizard offered by sipwise.com can provide configuration templates for many scenarios.

As a drawback, the configuration of an OpenSER server can reach a complexity that is no longer easy to handle. An inexpert user can easily misconfigure the system, rendering it unusable, or – even worse – opening security holes.

Still, the features offered by OpenSER make it a powerful tool for a large range of SIP environments.

In February 2007, network equipper Cisco systems chose OpenSER as their SIP proxy choice for the Cisco Service Node for Linksys One.

3.3. OpenSER from a code perspective

OpenSER is developed in C. A main focus of the implementation was performance; not only was this the main reason for the choice of the language C, but it is also the primary reason for some design aspects in the program. Missing inheritance and class concepts in the programming language resulted in rather low level and not well abstracted structures in the OpenSER source – for the sake of performance. The “non-object-oriented feeling” of the underlying code is passed on to the user by means of the configuration file.

OpenSER consists of a number of distinct units that interact through defined (although unfortunately not well documented) interfaces. These units, reflected by individual subdirectories in the source tree, are:

1. OpenSER Core
2. SIP message parser
3. Memory Management
4. Database interface
5. Core extension for TLS support
6. Management interface
7. OpenSER modules

While listed separately, points two through six are tightly integrated into the core and cannot be separated from other parts of the code. Most of OpenSER’s functionality is located in the modules, however. The core and its components consist of approximately 67’000 lines of C code, while the modules total up to 157’000 lines, currently distributed into almost 70 separate modules[2].

[2]These numbers were evaluated in January 2007 and change rapidly

3.3.1. Types of modules

The possible functionality of OpenSER modules can be categorized into three groups: Providing functions exported for user access, implementation of the database API, and definition of an API for access from other modules. Additionally, modules focused on all kinds of functionality can offer user access through the management interface.

The largest share of modules is responsible for the creation and extension of features, ranging from different routing strategies over the access of external data to the rewriting of SIP messages. This is accomplished by passing control to the core configuration file, where the user can "call" functions exported by the modules. Only half a dozen modules explicitly provide an API for usage by other modules.

Although structurally identical, the five different database modules provide a completely different functionality. Their functions are not marked for usage in any type of routing definition and thus cannot be accessed from within the configuration file.

3.3.2. Export structure

All modules contain a nested C structure `module_exports`, defined in `sr_module.h`:

```
struct module_exports{
        char* name;                   /* null terminated module name */
        unsigned int dlflags;         /* flags for dlopen */

        cmd_export_t* cmds;           /* null terminated array of the exported
                                         commands */
        param_export_t* params;       /* null terminated array of the exported
                                         module parameters */

        stat_export_t* stats;         /* null terminated array of the exported          10
                                         module statistics */

        mi_export_t* mi_cmds;         /* null terminated array of the exported
                                         MI functions */

        item_export_t* items;         /* null terminated array of the exported
                                         module items (pseudo-variables) */

        init_function init_f;         /* Initialization function */
        response_function response_f; /* function used for responses,                  20
                                         returns yes or no; can be null */
```

```
    destroy_function destroy_f;     /* function called when the module should
                                       be "destroyed", e.g: on openser exit */
    child_init_function init_child_f;/* function called by all processes
                                       after the fork */
};
```

The most important components of this structure are the "`cmds`" and "`params`" variables. `cmds` refers to an array of structures; each of these structures contains the definition of a function that is accessible from other components (other modules, through the database API, or directly for user access). `params` holds the names and types of the module parameters that can be set from within the configuration file.

During the initialization of a module, the export structure is loaded by the core and can later be referenced from the script or by other modules. This query is accomplished by a set of core functions that return pointers to the module functions, parameters and variables.

3.3.3. Parameter fixup

As described above, function references in the configuration file always use strings as parameters. As some module functions in fact operate on different types of data – such as simple integers, deeply nested structures, or arrays of pointers, the function call is preceded by a parameter conversion. The export structure `cmd_export_t` may contain pointers to so-called fixup functions for every listed function. Static strings in the configuration file are converted to an arbitrary data type (C type `void *`) upon initialization; at runtime, the data transformation does not have to be repeated.

Samples of possible fixups are:

- Conversion of a string to an integer (e.g. reply codes)
- Conversion of a string containing variable names to an array of pointers to the string segments plus pointers to functions that return the values for the variable names (e.g. the pseudo-variable printing of the "`xlog`" module)
- Conversion of raw C strings to OpenSER's "`str`" type (a character pointer plus a length)
- The presence server module ("`pa`") registers parameters in its internal database upon parameter fixup. The fixed data type is a linked list.

There are two main reasons for the distinction between the parameter preprocessing and the function call itself. Obviously, the preprocessing largely increases the execution speed and thus the message throughput of the system. Secondly, modules that call other modules' functions can prepare "sensible" parameter structures themselves instead of constructing a string representation.

As a drawback, two problems arise due to the fixup separation: Firstly, the C type system is circumvented – all parameters are cast to `void *` variables. While casting arbitrary pointers to void pointers is still more or less sensible, casting integers to `void *` deeply hurts the type concept. Explicit type casts are usually regarded as a weakness in modern programming environments. Additionally, the system in its current state can do arbitrary operations during the fixup (this results in memory space allocation in many cases). There is no abstraction of this operation, so that other calling modules possibly need to replicate code if they want to pass parameters of the correct type. Furthermore, there is no encapsulated method to undo the fixup. If – for whatever reason – a module calls a fixup function for creating arguments, the result's structure needs to be known in that module so that it can revert the operations (e.g. freeing the right memory structures).

3.3.4. In a nutshell...

The priorities of software engineering have changed during the last decades. While efficiency was a central design aspect in the beginning of computing, its meaning has significantly decreased over time. OpenSER (and its ancestor SER), however, was designed to handle carrier grade VoIP traffic, so efficiency was again a big issue. The design goals derived from this resulted in a number of facts that make OpenSER's code a little difficult to handle in some aspects.

From the software engineering point of view, C is a deprecated programming language. More modern languages provide better abstraction methods and object orientation. The latter could have helped with classes for SIP messages and URIs, or with classes modeling module export structures. On the other hand, the direct access to low level system functionality greatly increases the system's processing speed. A comparison of different memory handling methods – particularly SER's `pkg_malloc`/`pkg_free` versus the standard `malloc`/`free` calls [29] – shows the advantages of low level programming in the context of a SIP server which are difficult to maintain with more abstract programming.

Comparing other open source projects, OpenSER's source code documentation is limited. Numerous central components are not well commented on. A core API reference does not exist.

Despite these negative points, OpenSER features an open design. Prototype modules could be written fairly simple. Adding features to the core was not a problem either, as the data structures in use are abstracted well enough to allow simple modifications.

4. Data handling

The initial concept for this thesis originated from the idea of integrating software systems that use different methods to store and manage data. In this chapter, this term will be discussed and a number of technologies that are available for data management will be examined.

The first part will describe different types of data, while the second part explains some backend technologies that are commonly used for different aspects of data handling.

4.1. About data and information

Everybody has an intuitive understanding of the term “data”. However, it is not so easy to find a formal definition. Computer scientists use the term for arbitrary series of bits, bytes or words. Data are the object on which programs and algorithms work; only programs give a meaning to data.

Computers can store, receive, send and modify data. Only by doing so, they can solve the problems they were created for.

4.1.1. Distinguishing data and information

Closely related to the term “data” is the term “information”. In many contexts, these words are used synonymously, and so will they be used in large parts of this work. Still, there is an important difference between the two of them [9]. Information is a piece of data that has a “meaning”, while data do not have to. While a series of 16 bits will be identical data on two machines, the information they contain may be different for a little and a big endian machine. A series of eight digits is a piece of data, but read as a person’s birthday, it becomes information.

The systems that will be connected in this thesis do not only need access to the other entity’s data, but they need to receive the same information as the other part does. In chapter 6, it will become clear that the problem is not a connection to a data storage, but that it will be much more difficult to create information from the data it may contain.

4.1.2. Types of information

A VoIP system needs to handle different types of information: Besides the obvious telephony data (SIP and POTS/ISDN traffic), there are a number of meta data types: Static and dynamic configuration data, transitional data (accounting, current registrations and calls, ...) and authentication/authorization data.

Differences in content and structure of these meta data categories are responsible for the difficulties of system integration in current VoIP software, as will later be described.

4.1.2.1. Configuration data

As described in [52], configuration of software systems is often twofold: a static configuration defines an environment for the software system to work in, whereas a dynamic configuration creates a tier with dynamically changing environmental circumstances. In the aforementioned paper, which discusses QoS aspects of a middleware system, static configuration contains information about the existing resources. The assignment of these resources is done dynamically.

Another example of the duality of static and dynamic configuration can be found in an MTA: static information – IP addresses and network environment or smarthost setups – stand in contrast to the dynamic aliases tables or domain responsibilities.

The dividing line between these two types of configuration data is often blurred; static configuration may sometimes be reloaded during a system's runtime, while aliases tables are frequently configured via regular configuration files.

A planned VoIP system ought to be able to handle a number of dynamic data, such as user databases, routing information or voice box setup. On the other hand, the configuration of how to react to certain dynamic configuration statements may be done statically. A user ought to be able to turn his voice box on or off dynamically, the voice box configuration itself may be static.

Configuration data are flowing *into* the system.

4.1.2.2. Transitional data

Software products commonly need a way to persistently store internal data. Although the underlying technologies may possibly be the same as for configuration data[1], the

[1] OpenSER currently uses SQL databases for both configuration and transitional data, as will later become visible.

semantics within the system differ largely. In terms of the MTA example, such a system will need a space to handle the mails that pass the system.

Similarly, the evolving VoIP system needs to store information about connected user agents or active calls.

Although transitional data usually are only handled in the software system itself, it is sometimes desirable to retrieve these data from the system. An MTA should have an interface to query the queue of currently waiting mails.

Transitional data flows *within* the system.

4.1.2.3. Authentication and authorization

It was shown that user databases are dynamic configuration data. At a first glance, authentication data seems to be equivalent to the user database. This is true on the syntactical level only, however. While a software system needs full access to the user database to retrieve all available information, access to authentication data should be restricted as far as possible for security reasons. As long as a software system knows that a user is authenticated, it is unnecessary for it to know *how* the user authenticated himself.

In practice, a VoIP system should not be able to retrieve a user's password while checking his authentication credentials. A strict distinction between the authentication data and the authentication itself is desirable. The user credentials can reside on a highly secured machine.

So-called "Triple-A systems" such as RADIUS provide an interface for this task.

Authentication data are *external* data that should be queried, but should never need to enter the system.

4.2. Technologies

Numerous alternative technologies exist to provide functionality for data handling. On the first glance, their objectives may look similar, but in fact the technologies are quite distinct. While some may be deployed in a wide range of purposes, others are extremely specialized.

4.2.1. The simple life: plain text

The most simple way of managing data is the usage of plain text files. No external constraints restrict their usage; their format is commonly only defined by the application they are used in[2]. For text files, there often is a large difference between data and information, as the knowledge that can be gathered from them depends on the context that is defined by their applications.

While text files reflect human methods of working, computers need to parse text files to receive information. This process can be quite expensive, depending on the file structure. Therefore, text files are only a good choice for small amounts of data at the human-computer interface.

4.2.2. Relational databases and SQL

As the amounts of data and the constraints on correct and efficient processing of them grew in the early days of the computer age, the need for dedicated systems for these tasks arose. In the 1960s, the concept of database management systems (DBMS) as an additional software layer was born and refined and formalized in the 1970s, eventually forming the relational database model. The SQL (Structured Query Language) defines a standardized way of storing, modifying and requesting data in relational DBMS.

A database management system handles data stored in a database; their sum is referred to as a database system.

In contrast to earlier models, the relational database model reflects the formalized mathematical models of relational calculus and relational algebra [30]. Data are stored in sets of relations. More practically, the model presents a set of tables with strictly defined names and types of columns. The set of these definitions is called a database schema.

Commercial as well as free software products today implement RDBMS. Due to the fact that they have evolved over a long time, they are among the most mature, most efficient and best understood software systems existent in computing. The relational model is fairly easy to understand, and the environments created by its implementations result in valuable, well-established programming interfaces.

Still, the relational database model implies a number of shortcomings that modern RDBMS still suffer from. The systems store data that do not have implied semantics.

[2] Specialized text file formats – such as Character Separated Values (CSV) files, and even XML – do exist.

Thus, the data mature to information only at the time the processing application applies its level of semantics. Different applications working on the same database system may or may not be "compatible" with each other in the sense of understanding each other's implied information.

Despite all criticism, nowadays the relational model drives all considerable database management systems on the market. Applications in almost every aspect of computing use relational databases as underlying data storages.

4.2.3. Directory services: LDAP

The Lightweight Directory Access Protocol (LDAP, see recent work in [42]) provides a more specialized technology of data storage. While relational databases are flexible enough to store almost arbitrary types of information, directory services are dedicated to store information about users and resources in computer networks. The LDAP protocol provides today's standard access protocol for directory services.

A directory hierarchically stores objects that belong to a set of classes. Standardized object classes can be complemented by private definitions. An object can belong to multiple classes. Each class defines attributes the object can or must have. The semantics of an object's attribute is implied by its definition in the schema. Thus, an LDAP directory can directly contain information, rather than raw data.

A common schema used to describe users in a network is the so-called `inetOrgPerson` class which includes attribute definitions for e.g. telephone numbers, e-mail addresses, or room numbers. More technically related, the `posixAccount` stores information about user accounts in Unix environments, adding attributes for e.g. user and group IDs, for home directories or for user's shells.

In many organizations, objects describing users belong to both the `inetOrgPerson` as well as the `posixAccount` class.

The ITU-T recommendation H.350, also depicted in RFC 3944 [26], standardizes LDAP schemata that may be used in VoIP environments. These classes reflect user agents (telephones) however, e.g. allowing a SIP registrar to store registry information in the LDAP. Classes that may be useful for user identity mapping are not provided.

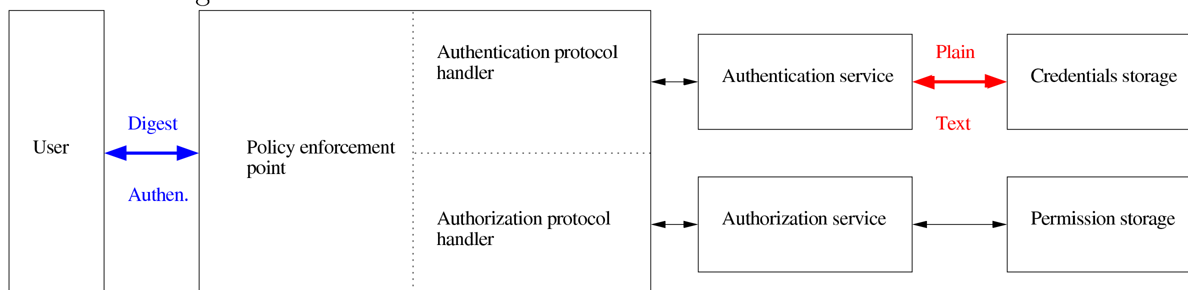
LDAP directories solve a set of rather clearly cut requirements. The fact that they are not quite as fast as relational databases is offset by the advantage of managing pieces of information instead of raw data.

4.2.4. Authentication services

As described, a directory such as an LDAP service stores information about users in a computer network. The `posixAccount` schema includes an attribute to store users' passwords. Passwords[3] are used to verify a user's identity. This is necessary to enforce a permission policy for an offered service.

Although it is sufficient for authentication mechanisms of modern operating systems – including Unix and Windows – to store encrypted (or, in fact, hashed) passwords, this is not sufficient for numerous other authentication mechanisms. SIP, for example, demands digest authentication, a challenge-response authentication mechanism already commonly used for HTTP.

Challenge-response mechanisms require the server to store the plain text password, or a password equivalent token [37]. In case of the digest mechanism, the (shared) key that is used by both sides to encrypt the nonces is called "HA1" and is an MD5 hash of the username, the authentication realm, and the password. While the original plain password does not have to be stored on the server, access to the service can be accomplished with the HA1 value alone. The following graphic shall visualize the objects concerned during authentication.



Coming back to the topic of OpenSER, the policy enforcement point is the OpenSER core; the authentication protocol handler may be one of the available authentication modules, `auth_db`, `auth_radius` or `auth_diameter`. While `auth_db` also includes the authentication service, `auth_radius` and `auth_diameter` rely on a distinct entity. The credentials storage may e.g. be `auth_db`'s relational database, or an LDAP directory back-end for RADIUS and DIAMETER.

The steps in the bottom half of the graphic, describing the authorization process, are similar, except that this path does not transport any secrets, but only information about the policy to be enforced.

[3] While other tokens – such as biometric properties or certificates – may be used for authentication, this discussion will be restricted to passwords

When the authentication service is integrated with the policy enforcement point, this means that the executed program containing these instances needs access to the plain text password, which in many cases is not desirable. Thus, a dedicated authentication service can be used to split these entities; the authentication service may well be integrated with the credentials storage.

Reverting to the real world example of OpenSER, it may be common to run the VoIP server on one machine, and a RADIUS server as well as its back-end, e.g. an LDAP directory, on another.

4.2.4.1. RADIUS and DIAMETER

The growth of the Internet was closely related to the availability of public dial-in services offered by Internet service providers. Their demand for a solution to the problem described above led to the development of services that provide functionality for **a**uthentication, **a**uthorization and **a**ccounting, forming a group of AAA ("Triple-A") services. The de-facto standard AAA protocol today is the Remote Authentication Dial-In User Service, or RADIUS [45]. A RADIUS server takes on the job of checking the username/password pair with data stored in its configuration (or data back-end). Additionally, it provides functionality for authorization and accounting, as these three topics are closely related in the context of dial-in services.

Remote Access Service (RAS) nodes pass incoming authentication data to the RADIUS server, which can then reply with a simple (boolean) "successful" or "unsuccessful" answer. Similarly, the question whether a (authenticated) user belongs to a group with sufficient rights for a certain action can be answered with yes or no. RADIUS provides additional functionality for accounting, and storing/requesting information about properties of a user, such as the maximum available bandwidths.

While originally developed for dial in services, the RADIUS protocol has proven to be flexible enough to provide AAA services for other tasks as well. In larger networks, wireless network access using the WPA encryption protocol commonly is implemented with a RADIUS server for user authentication. Multiple VoIP server implementations (including OpenSER) can use RADIUS based authentication. The system can also be used for VPN authentication.

The RADIUS protocol itself is a UDP based Internet protocol. Various commercial and free implementations exist. In the open source market, the most important ones are freeRADIUS and openRADIUS. Available back-ends in both implementations include plain text configuration, SQL databases and LDAP directories.

DIAMETER is RADIUS' proposed successor, but has not yet gained a lot of real-world attention. The core concepts of RADIUS are adopted unchanged.

4.2.4.2. Kerberos

With the technologies described above, a user needs to authenticate – and thus enter a password – with every service he wants to use. The Kerberos protocol uses so called tickets to remove this restriction. When logging in (or registering with the Kerberos realm), a user's credentials are checked, and (in case of successful authentication) he is given a "Ticket granting ticket" (TGT). When requesting a service, an additional ticket is requested (and granted, as long as the user is authorized for the service). This ticket is used to authenticate with the particular service.

While RADIUS is completely implemented on the server side, Kerberos needs software support on the client side to handle the Kerberos tickets. Due to this, every communication protocol needs to be specified for usage with Kerberos.

The SIP protocol does not include a Kerberos specification (although its integration would be possible). Thus, no known SIP proxy or user agent implementation features Kerberos ticket handling.

4.2.5. ENUM

ENUM stands for tElephone NUmber Mapping. The latest version of the ENUM standard is specified in RFC 3761 [15]. Its primary intent is to provide a meaning of converting telephone numbers into URLs – which in turn typically are SIP addresses, but may as well be be e-mail addresses or conventional "HTTP" URLs. The records in the ENUM domain zone are NAPTR records [38]; this adds features for evaluating the returned records with regular expressions.

ENUM is based on the conventional and long established DNS service [40, 41]. A distinct top level domain, e164.arpa, was created that hosts the ENUM records[4]. The telephone number +49-761-203-1234 translates to ENUM domain name 4.3.2.1.3.0.2.1.6.7.-9.4.e164.arpa: the canonical phone number – beginning with the international prefix – is reversed, the digits are separated with the domain-delimiter ".", and the ENUM domain e164.arpa is appended.

The above DNS address might resolve in the following way:

[4] Other domains may be used for ENUM entries, too; the domain .e164.arpa is globally standardized. Other registration organizations try to establish parallel ENUM domains.

```
1 bastian@hostname:~> host -t NAPTR 4.3.2.1.3.0.2.1.6.7.9.4.e164.arpa
2 4.3.2.1.3.0.2.1.6.7.9.4.e164.arpa has NAPTR record 1 10 "u" "E2U+tel" "!^.*$!tel:+497612031234!" .
3 4.3.2.1.3.0.2.1.6.7.9.4.e164.arpa has NAPTR record 2 10 "u" "E2U+tel" "!^.*$!tel:+491721234567!" .
4 4.3.2.1.3.0.2.1.6.7.9.4.e164.arpa has NAPTR record 3 10 "u" "E2U+sip" "!^.*$!sip:1234567@sipgate.de!" .
5 4.3.2.1.3.0.2.1.6.7.9.4.e164.arpa has NAPTR record 4 10 "u" "E2U+mailto" "!^.*$!mailto:me@uni-freiburg.de!" .
6 4.3.2.1.3.0.2.1.6.7.9.4.e164.arpa has NAPTR record 5 10 "u" "E2U+http" "!^.*$!http://www.uni-freiburg.de/~me/!" .
```

An ENUM-aware agent (server or client) is able to request different contact methods that are associated with that telephone number. In this case, a caller might contact the callee via a POTS or mobile telephone number, a SIP address, by e-mail or at via web site.

In contrast to the other technologies described above, ENUM describes a special schema of data in the Domain Name Service. While different SQL-based software systems usually will be incompatible with using each other's data (as their schemata differ), every ENUM-enabled device or software will be able to use the data provided in the e164.arpa domain.

4.2.5.1. Status quo

After being standardized for some years now, ENUM has been put into operation by German DENIC in January 2006 [1]. DENIC is responsible for the subdomain 9.4.e164.arpa (which translates into the international telephone number prefix +49). Subdomains – such as 3.0.2.1.6.7.9.4.e164.arpa for the university of Freiburg – can now be registered via regular registries.

On a global perspective, ENUM trials have been initiated in most parts of the world. In many European countries, ENUM registrars have started their regular operation.

ENUM is ready to use for end users and widely available through regular domain registration processes. The technology in software systems (servers and soft phones) as well as in hardware devices has been in operation for some time. On the other hand, the business models of many well-known SIP providers have the effect that users – ENUM-enabled or not – cannot directly contact each other, as the providers earn money by routing calls to the POTS and vice versa. Routing calls through SIP that were initiated with a conventional phone number would not create any revenue. Due to these artificial hurdles, the importance of ENUM for end users is still restricted.

Part II.

Analysis and specification

5. Requirements analysis

The advantages of a VoIP environment over a conventional telephony network seem obvious: reduction of telephony and administrative costs as well as the increase of productivity provided by new features. On closer consideration, the desired tasks and features of a VoIP system have to be determined a lot more detailed, however.

The object of this diploma thesis is the integration of VoIP products into a homogeneous system. Integration will mean that there have to be common data paths between the interfaces of the subsystems. The discussion of data structures will occupy an important place in this chapter.

5.1. Use case analysis

It is necessary to outline the use cases in a VoIP system. Special consideration will be given to the types of data and information handled by the system during the processing of the tasks.

Derived from the feature lists of telephone carriers ([54, 48]) and taken from literature ([22, pp. 36ff], [13, pp. 288ff], [27, pp. 165ff], [28]), typical use cases can be found; for most cases, common sense is sufficient, as we have a good expectation of what a telephony network should do. The use cases of a VoIP system that will be examined are:

- Incoming VoIP/POTS call
- Outgoing VoIP/POTS/“auto” call
- Call transfer
- Call deflection
- Conference calls (internal, external, mixed)
- SIP registration (attaching a phone) and authentication

- User and authorization/permission management
- Personal self-administration
- Accounting
- ACD, IVR, Call hunt groups [8]

Most of these use cases are fairly obvious; their description can therefore be short. Attention is mainly given to the data handled within the respective use cases.

Thus, the types of data found during this analysis will later be categorized and attributed to entities in the OpenSER software.

5.1.1. Use case: Incoming phone call

Participating Agents: Caller, Callee

Description: A caller reaches the VoIP system via POTS or VoIP. The following cases are possible:

- Callee answers phone. The conversation must be set up.
- Callee has set "Do not disturb" mode. Caller is to receive busy signal.
- Callee is already engaged. Depending on setup, caller is to receive a busy signal, or the callee is to be signaled "call waiting".
- Callee does not answer phone. The call is to be dispatched to a voice mail system after a number of rings.
- Callee does not exist. Caller is to be signaled a "user unknown" message.

The phone device (softphone or hardware) of the called user signals the incoming call. If the caller's ID is transmitted, it ought to be displayed.

Data in use:

- Identity information: who is the receiving user?
- Registration/location information: Where is the receiving user?
- Status information: is user engaged?
- Voice box setup
- Private/public address books: display of caller identity

5.1.2. Use case: Outgoing phone call

Participating Agents: Caller, Callee

Description: Caller uses one of the following methods to initiate a call:

1. Dials a phone number on his phone
2. Dials a SIP address on his phone
3. Uses a web interface or dedicated software to dial a number/address ("Click to dial")

Caller needs to have access to a system wide address book. The system shall autonomously choose a route to the callee depending on a number of factors. Possible outgoing paths include POTS/mobile phone and VoIP.

Data in use:

- Public/private address books
- Linking information between address/phone number and a user
- Location information of users ("how do I reach user X?")
- Authorization (May caller initiate call to this address/number?)
- Accounting

5.1.3. Use case: Call transfer (External: VoIP/POTS, internal: VoIP)

Participating Agents: Original conversation partners P1 and P2, new conversation partner P3

Description: P1 and P2 have a conversation. P1 initiates a call transfer to P3. An incoming call is signaled to P3.
Although different cases can be distinguished, this use case behaves very similar to “outgoing call”.

Data in use: See “outgoing call”.

5.1.4. Use case: Call deflection

Participating Agents: User

Description: User configures VoIP system to route incoming calls to a third number/address.
This may be done through a web interface or via the phone.

Data in use:

- Aliases database
- Authorization data

5.1.5. Use case: conference calls (internal, external, mixed)

Participating Agents: Conference administrator, multiple users

Description: Administrator creates a "conference room" through a dedicated interface, e.g. a web site. A phone number/SIP address is assigned to this room and a secret (PIN) may optionally be set. Additionally, access to the conference room may be restricted by ACLs (incoming address/number).
Users may join the conference by dialing its number/address and entering the PIN.

Data in use:

- Conference room database
- Authorization data
- Aliases database
- Address book

5.1.6. Use case: SIP registration (attaching a phone) and authentication

Participating Agents: User

Description: User attaches a phone (hardware, softphone) to the VoIP network (LAN, VLAN, ...) and configures the device to use the VoIP server.
System checks user credentials and stores his contact information.

Data in use:

- User database
- Authentication data
- Location database

5.1.7. Use case: user and authorization/permission management

Participating Agents: Administrator

Description: The administrator can

- create user accounts
- set/change user passwords
- set permissions (e.g.: POTS calls, long distance calls, international calls, conference rooms), possibly in conjunction with time intervals (e.g.: no calls on Sundays, no calls outside of office hours)
- set up voice mail for users

Data in use:

- User database
- Authentication data
- Authorization data
- Voice mail setup

5.1.8. Use case: personal self-administration

Participating Agents: User

Description: A user may set a number of personal information stored about his identity, e.g.:

- Real name and similar information (room number, ...)
- Aliases (possibly, if permission is given)
- Voice mail setup, including announcements
- Call deflection (see use case 5.1.4)
- Status information (Presence, Do not disturb/DND)
- User password

Data in use:

- Address book
- Aliases database
- Voice mail configuration
- User and authentication databases
- Status information

5.1.9. Use case: Accounting

Participating Agents:	Calling user, accounting manager
Description:	User initiates and receives calls to/from other users. The VoIP system creates records for the calls which may later be evaluated by the accounting manager (call source and destination, used route, time and duration of call).
Data in use:	

- Accounting database
- User database

5.1.10. Use case: ACD, IVR[1], Call hunt groups

Participating Agents:	Callers, administrator, dispatchers
Description:	The call center administrator can create user groups that receive incoming calls in call hunt groups and ACD setups. It is desirable that this setup automatically works in parallel with other organizational groupings. IVR depends on a heavily customized setup. Connections between user input and actions based thereon are to be configurable from within an integrated front-end.
Data in use:	

- Groups of users
- Alias database
- User configurable meta databases

5.2. Categorizing data

After examining the data in use while handling common use cases in the last section and investigating the categories of data in section 4.1.2, a matrix of memberships is needed:

[1] The possibilities provided by Interactive Voice Response are arbitrarily large. The methods of user input, the methods of processing the input and the meaning for the call center are varying. Although the work of this thesis should provide features that help for a good IVR technology integration, IVR itself will not be discussed any further.

Data category	Data types
Configuration	• Identity information • Voice box • Address book(s) • User database • Aliases database • Conference room database • User groups • Meta databases
Transitional	• Registration/location information • Status information (Engaged, DnD, ...) • Accounting
Auth.	• Permission/authorization database • Authentication database

5.3. Data handling in OpenSER

As detailed in chapter 4, there are commonly deployed technologies for handling the data examined in this chapter. These technologies now have to be made available for usage with OpenSER.

Not all of the data categories defined above have a counterpart in OpenSER; some of the topics they belong to are not a property of a SIP environment, but are related to higher level functionalities in telephony networks. The basic technical aspects of

OpenSER have been elaborated on in the last chapter. The data categories can now be related to OpenSER's technology that handles them.

Data	Functionality/OpenSER module
Identity information	The term "identity information" is not defined well. It refers to the subscriber database used by the authentication modules as well as to the location and alias databases. See below for discussions of these databases. The Caller ID displayed on receiving devices is normally set by the calling device; although its modification is possible, there is no standard action for this task.
Voice box	OpenSER is a SIP-only server and as such does not handle voice mail. Commonly deployed configurations use call forking mechanisms to forward calls to answering machine applications. One possible way to implement this forking is to use a process similar to aliasing.
Address book(s)	Although address books and their querying are a feature of VoIP clients (hard or soft phones) rather than a server function, there are some possible points of interaction: modification of Caller ID, speed dial functions or user aliasing, to name only a few. Speed dialing is supported via the `speeddial` module, aliasing is implemented in `alias_db`. The modification of caller IDs is a topic too complex to discuss here; so-called RPID headers can be appended. Modification of the `From` header is supported.
User database	The list of subscribers is necessary for user authentication. OpenSER currently supports relational databases, RADIUS and DIAMETER.
Aliases database	Aliases may be evaluated through OpenSER's `alias_db` module that depends on a relational database.

Data	Functionality/OpenSER module
Conference room database	Conference rooms are features that are implemented in back-to-back user agents (b2bua). OpenSER can only provide SIP-based functionality for the topic, e.g. access control. A sample configuration in OpenSER's documentation describes the use of AVPs for conference room access control. This sample uses the `avpops` module with a relational database connection.
User groups	OpenSER supports user groups through the `group` and `group_radius` modules for using relational databases or a RADIUS back-end. Call branching provides the means for call hunting groups.
Meta databases	OpenSER does not have integrated support for arbitrary user-configurable data. However, the underlying relational schema may be extended, as long as these modifications do not interfere with OpenSER's modules.
Registration/location information	The location database in OpenSER, implemented in the `usrloc` module, provides sophisticated caching techniques. It is based on the relational database model. Information about user location is also given through ENUM entries. The ENUM domain zone may be queried by the functions of the `enum` module.
Status information	OpenSER's latest release incorporates a presence server as well as a user agent (`presence` and `pua` modules). Both rely on relational databases.
Accounting	When SIP-only telephony is used, accounting is not a simple subject; after establishing a SIP connection, the extent of RTP traffic is not ultimately known. SIP-only accounting thus is not reliable. OpenSER has a sophisticated `acc` module that tries to circumvent a number of problems. It can use relational databases, a syslog service, or RADIUS servers for logging. A database engine "`flatstore`" is provided to log to raw text files.

Data	Functionality/OpenSER module
Permission/ authorization database	OpenSER's `permissions` module can be used to determine if a call has appropriate permission to be established [56]. The decision is based on text files similar to Unix `hosts.allow`/`hosts.deny` files. A relational database can be used for internal caching. Permission granting based on dynamic limits (e.g. time, call rate, ...) is not supported.
Authentication database	Authentication is supported by OpenSER's `auth`, `auth_db`, `auth_radius` and `auth_diameter` modules. The first provides basic authentication functionality; the others provide implementations for relational databases, RADIUS and DIAMETER services.

The software system uses relational databases as sources and destinations of most data read/write operations. Where appropriate, the “Triple-A systems” RADIUS and DIAMETER are supported.

5.3.1. Data paths in OpenSER

OpenSER's module structure results in a number of internal paths which chunks of data need to transit. These data paths connect the interfaces of the system modules. The external interfaces described in the last section use relational data sources and sinks.

More abstract, these data paths in OpenSER can be identified:

- The **core** exchanges data with its **modules**. The primary data structures are C structs of type `struct sip_msg`.
- As mentioned before, **modules** exchange data with the **database modules** (although OpenSER does not use the term, similar entities are commonly called “drivers”) through the DB API. Multiple underlying C structs (`struct db_*`) are used; dedicated encapsulating container structures do not exist.
- The **database modules** exchange data with their underlying technology, specific **DBMS**. As different database management systems implement (at least) slightly differing APIs, they do not feature common data structures.

- **Modules** may exchange data with other **modules**. This may be done by extracting regularly exported functions, or through APIs defined by the serving modules. In the first case, the `struct sip_msg` is also used; in the latter case, there is no common data structure, as the functions may be accessed natively with their respective signature.
- OpenSER provides **external access** through the Message Interface (MI) for **core** and **module** functionality. Two modules provide alternative methods of external interaction with the MI; one uses the Unix FIFO semantics to allow for simple user interaction, while the other implements an XMLRPC mechanism. These access modules exchange data with OpenSER and its modules through a number of C structures, particularly `struct mi_node`, `struct mi_root` and `struct mi_attr`.

5.3.2. Data management bindings in OpenSER

As described above, OpenSER uses relational database schemata for multiple data processing tasks. While this holds true for most standard modules, support for other data back-ends is integrated for a limited set of functions. The "Triple-A systems" RADIUS and DIAMETER can be utilized for authentication, authorization and accounting through bindings to the freeRADIUS and openRADIUS libraries (the DIAMETER implemented does not rely on a given library). Plain text files can be used as relational database substitutes through the `dbtext` module.

OpenSER's modules are deeply interweaved with the data they work on and the structure they are stored in. Due to this, substituting the relational models with arbitrary storages is not easily possible. Despite this fact, a number of different implementations for accessing LDAP directories have been created in the past. All these implementations are currently restricted to provide data for predefined sets of purposes, however, and generally do not allow for access through existing OpenSER modules.

The first major implementation of LDAP in SER (not OpenSER, in this case) was the ldap(s) module developed at the ETH Zürich [12]. This relatively simple module can request single attributes from an LDAP directory to substitute the recipient URI of a SIP message. While these modules are relatively simple to use, they are not able to provide any other information from the LDAP service than a modified RURI.

In January 2007, another implementation of an LDAP module for SER (again, not OpenSER) was published [11]. In this case, an underlying LDAP infrastructure module can be accessed from separate modules. A single client module exists which provides

authentication against an LDAP service. While LDAP authentication in the context of a SIP server is not recommendable, the concept of an LDAP module as a data source and other modules as data sinks could be extended for other parts of the server. It would be necessary to duplicate large parts of the code, however, as every current data sink module would need an LDAP counterpart.

Although OpenSER uses a relational schema to represent its data, this does not mean that OpenSER is good at handling SQL databases. The used schema is reflected by unencapsulated details in the C sources. If a user wants OpenSER to interact with “grown”, long established data sources, sophisticated data mapping technology has to be used to map private and OpenSER’s native schemata into each other.

5.4. Refining the requirements

In this chapter, OpenSER’s handling of dynamic configuration, transitional, and authentication data has been examined, including the data access occurring in OpenSER and its modules. Data paths from these modules to their respective back-ends were identified. It became obvious that the current data access framework rigidly restricts the data paths to the currently implemented relational model, without the potential to exchange data with yet unknown, possibly well-established data sources. In particular, LDAP directories which commonly contain information about present users are not accessible.

The tasks of this work can now be refined:

Currently existing OpenSER modules shall be given access to a flexible, configurable data access layer. As the current code base is in use in various large, well-established environments, the currently existing database API cannot be substituted, but needs to be complemented.

The use case analysis tried to anticipate the most important functionalities of a VoIP system, but many other OpenSER modules add to the features of the software subsystems found above. Thus, it is necessary not only to implement complementary modules that provide methods for foreseeable cases and technologies (such as an LDAP binding for aliasing), but open the data paths for as yet unpredictable data flows.

While the current data paths are restricted to a limited number of possible end points for each of the paths, the extension that will be developed for this thesis will create a network of arbitrary data sinks and sources, where every possible combination of database and data consuming module can be realized.

The access methods for the underlying technologies and the design of the new system still have to be defined.

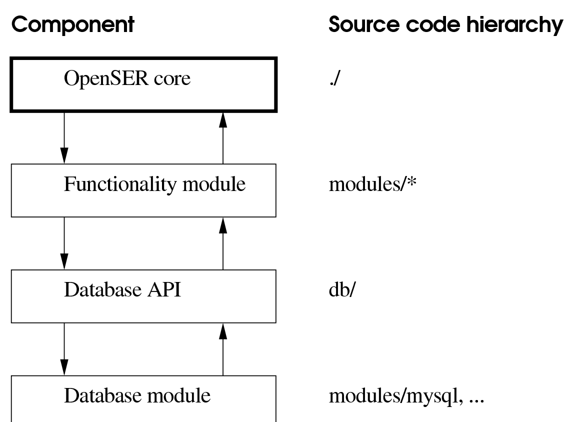
6. Specification

Chapter 5 provided a better understanding of the data handling entities in OpenSER and the problems that arise when adding new data sources and sinks to the system. This chapter specifies the stops that data transit and proposes a solution for extending the data paths.

6.1. A flexible database backend for OpenSER

OpenSER is a modular software system. Approximately two thirds of its code are located in the modules. The core system provides the base functionality for the SIP protocol, TCP/UDP interfaces, config file parsers and similar parts; modules add functions e.g. for authentication, aliases, accounting and many others.

Database access involves three entities: First, there are modules that require database functionality, such as the alias database. The second component is a generic database API that provides basic relational database handling functions, such as *query*, *insert* or *delete* functions. The third participant is a set of particular database modules for relational database management system (RDBMS' for SQL databases or text files). These components are organized in a layered structure: the database implementations provide a common interface for low level database access. The database API creates a higher abstraction layer that facilitates a unified access to the low level drivers without the need to operate with raw SQL queries. Based on this API, modules can request database information. On top of the described three layers, the core system decides on routing based on the answers provided by module requests.



A way to express the desired behavior when exchanging the relational database with an arbitrary data back-end such as an LDAP directory has now to be developed.

The formulation of a data path has to be done in a static configuration file (see section 4.1.2.1). The next section will thus examine what the term “configuration” describes in a formalized way.

6.2. Digression: configuring vs. programming

There are two methods of modifying the behavior of a software system: To configure it, and to program it. This section will discuss some differences between these two expressions in semi-formal terms.

In a strict sense, every program is a configuration, and every configuration is a program. Each piece of software may be regarded as a configuration of the underlying computer system, while every configuration is a program that is interpreted (“run”) by a software's configuration parser. This definition opposes common sense, however, and a deeper understanding of this common sense will help in the decisions to be made.

A number of patterns is commonly used in configuration files; some of these are described here:

- Flat attribute/value pairs: each line of the configuration file holds the name of one attribute, plus one or more values for this attribute. A sample of this type of file is Postfix' main.cf[1] configuration file. Configuration files of bash and PHP scripts often follow this pattern, as does OpenSER's global configuration.

- “INI style”: contains sections marked by “[section]” statements, plus one-line AVPs. This type of configuration file became famous with Microsoft's Windows 3.x and is still used e.g. by the Samba file server.

- Block definitions: Sections are marked by a section name; the section itself is enclosed in curly braces. The advantage of this type of configuration over flat and INI style configs is their ability to hold nested information. The name server system “bind” uses config files of this type.

- XML configuration files: Values are enclosed in XML tags denoting the described attribute.

[1]Postfix is a well-known industrial-strength open source mail server solution. Its static configuration is based on two primary configuration files.

- Hierarchical trees: Each node in a tree may hold multiple attribute/value pairs plus multiple other nodes. The file format may be binary. The best known sample of this type of configuration is the Microsoft Windows registry.

In a formalized sense, all these methods create first order relations, as the right hand side of each assignment is constant. Although that constant may describe a processing instruction, e.g. by assigning regular expressions, these expressions cannot refer any type of function.

As opposed to a "configuration", a "program" is able to express higher order functions that create a right hand side of an equation by calling arbitrary functions.

Where an expression such as

```
1   foo=8;
```

will commonly occur as a configuration statement, calling a function to produce the output "bar" will rather be regarded as being a "program":

```
1   bar=2*2;
2   foo=2*bar;
```

The next listing makes this differentiation even clearer:

```
1   bar=2*i;
2   foo=2*bar;
```

calculates the value of "foo" by referring to a variable "i" that is only defined in the context of the program executing the statement. This clearly looks like an excerpt from a program source code.

All this boils down to a gradual distinction between a programming language featuring rich, powerful expressions on the one hand and configuration languages with simple attribute/value pairs on the other.

The following table shows a number of common expressions in "pure" configuration files and programs:

Configuration	Program
Scalar values: integers, numbers, strings	Variables, operations on variables
sections, nodes	functions, subroutines
Information about the subject	Information about how to handle the subject

6.3. Functions in OpenSER modules

In this section, the process of calling a function in an OpenSER module will be described. One of the most interesting module functions is the alias lookup function that currently exists in the `alias_db` module. This function shall serve as an example throughout this section. Upon an incoming call, it is able to look up possible destination addresses in case that the incoming call is destined to an alias address.

To formalize the low level processes during a call to the `alias_db_lookup` (and other module functions that rely on a database), a couple of data types need to be defined first:

Let Υ be the set of valid URIs. Parts of these URIs are the user (Θ) and domain (Δ) part. There exists a bijective function g so that

$$g(t,d) = u; g'(u) = \begin{pmatrix} t \\ d \end{pmatrix}; u \in \Upsilon, t \in \Theta, d \in \Delta$$

[2]. This also means that $\Upsilon \equiv \Theta \times \Delta$. Let Φ be an arbitrary set (the remaining attributes of the `sip_msg`). The set of SIP message objects in OpenSER, represented by the C structure `sip_msg`, is then implemented as $\mu = \Upsilon \times \Phi$.

Let there be a set ε denoting events (such as transiting SIP calls).

The data paths utilized during such a database request will be exemplarily displayed for an alias lookup in the `alias_db` module:

OpenSER executes

$$f_{in} : \varepsilon \rightarrow \mu$$

creating a `sip_msg` structure on the occurrence of an event, and calls the alias lookup function in module:

$$f_{lookupalias} : \mu \rightarrow \mu, f(m) = m'$$

The lookup routine extracts the user and domain parts of the URI:

$$f_{pre} : \mu \rightarrow \Theta \times \Delta$$

[2] The function g is the concatenation of the string 'sip:', the user part $\in \Theta$, the '@' symbol and the domain part $\in \Delta$. Its reverse function splits these tokens.

which is then passed to the SQL request

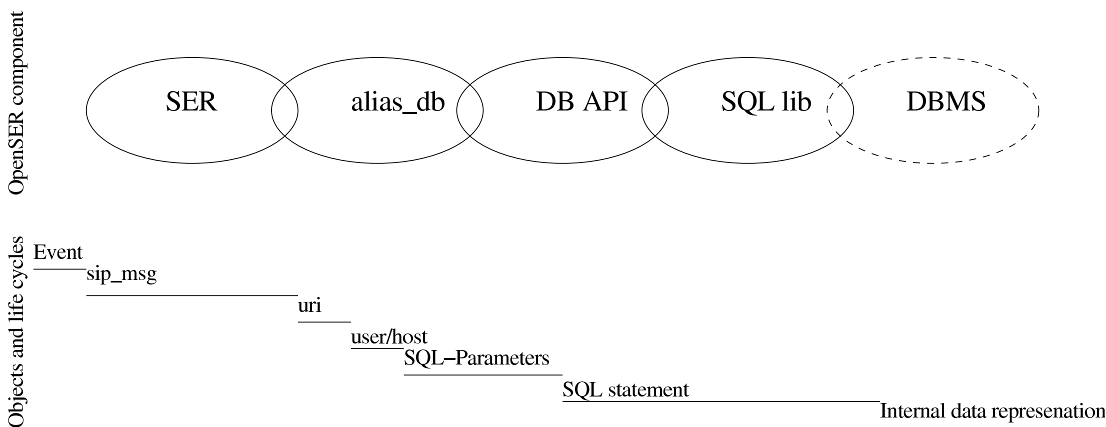
$$f_{SQL} : \Theta \times \Delta \rightarrow \Theta \times \Delta; f_{SQL}(t,d) = \begin{pmatrix} t' \\ d' \end{pmatrix}$$

It is important to note that this function $f_{SQL}$is two-figured, taking two scalar values t, d and returning the tuple t', d'; the extraction of t, d is implemented directly in the C source. The pair t', d' resulting from f_{SQL} are then postprocessed, passed back to OpenSER and the resulting SIP message is finally sent to the destination host:

$$f_{post} : \Theta \times \Delta \times \Phi \rightarrow \mu$$

$$f_{out} : \mu \rightarrow \varepsilon$$

In short, the operations on the SIP message object that are executed during the run of the alias lookup function create a number of data types with different life cycles. This can be visualized:



One of the existing OpenSER modules ("lcr") uses the raw query capability ("DB_CAP_-RAW_QUERY") of databases, creating the SQL statements in the module already.

It was shown that the invocation of a module function that uses a database back-end resolves to a number of stacked functions:

$f_{call}(m) = f_{post}(f_{SQL}(f_{pre}(m)))$

To provide maximum flexibility for the user to define this call, it is necessary to provide at least configuration statements to specify f_{pre} and f_{post} as well as a choice of different $f_{request}$ functions that relay the operations to a particular database back-end.

6.4. Expressing a data path

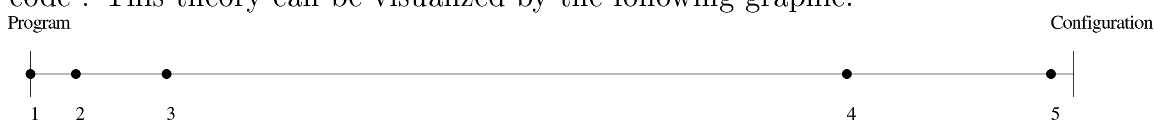
The terms "configuration" and "programming" as well as the processes involved during the run of the alias lookup in the `alias_db` module have been discussed. What is necessary to exchange the database layer? Unfortunately, the `alias_db` is closely related to the underlying relational data model; even worse, it is already dependent on the given database schema that consists of tuples of this form: (user, domain, alias_user, alias_domain). The following table shows a sample alias database table for OpenSER:

user	domain	alias_user	alias_domain
gjas	uni-freiburg.de	gerhard.schneider	rz.uni-freiburg.de
gjas	uni-freiburg.de	big.boss	rz.uni-freiburg.de
dsuchod	uni-freiburg.de	dirk.von.suchodoletz	rz.uni-freiburg.de
friedrib	uni-freiburg.de	bastian.friedrich	uni-freiburg.de

The extraction of the alias_user/alias_domain tuple from the incoming `sip_msg` structure is done in the `alias_db` module[3]. Directly trying to attach an LDAP backend that only has a single value for addresses (e.g. the email attribute, email aliases, or the user's telephone number) will involve a major re-implementation of the `alias_db` module[4]. Additionally, there needs to be a way to express the pre- and processing steps before and after database access. These functions f_{pre} and f_{post} would have to be defined in the configuration file as well as the choice of $f_{request}$ (which would substitute f_{SQL}).

6.4.1. Possible tradeoffs

It was demonstrated in section 6.3 that the type and amount of information transiting the OpenSER is different in each layer. Depending on the software design, the definition of each of the functions described there may be implemented at various points. The "intelligence" contained in the implementation of f_{pre}, f_{post} and $f_{request}$ can reside at any point between the extremes "only C code, no configuration" and "only configuration, no C code". This theory can be visualized by the following graphic:



[3] String processing is done by the message parser; the `alias_db` module initiates the parsing process and requests these components from the structure.

[4] The same holds true for other modules and their respective data.

- (1) denotes the point where all information about the data path is located in the OpenSER sources. An LDAP access would be implemented directly in the C source. Configuration is non-existent.
  Programs of this type are small, monolithic pieces of software. This type of architecture is not appropriate for publicly deployed server software.

- (2) might denote the current state for accessing the alias database. The structure of the database schema (two columns input, two columns output) is reflected in the C source. Configuration may only change the database URI and the column names. This information is implicitly included as parameters in the f_{SQL} calls.

- (3) would be a possible setup for adding a new database technology. Additional data sources need to be reflected by adding new modules for each of them. An alias database based on LDAP requests could work this way: an `alias_ldap` module could complement the current `alias_db` module. The OpenSER configuration would call two distinct functions from two modules for LDAP and SQL requests; they would be separately configured for certain URIs and filter strings. The "one input/one output" schema in LDAP had to be reflected in the C source, similar to the current two-figured SQL version. The f_{pre} and f_{post} functions are implemented in C; f_{SQL} is complemented with an independently implemented f_{LDAP}.
  Some existing modules in OpenSER such as the `usrloc` module resemble this architecture by letting the user configure the usage of the underlying database. The SER ldap module by Rogelio Baucells (see section 11.3.3) also follows this template.

- (4) would include a very rich configuration language that enables all existing OpenSER modules to utilize common data paths. Pre- and postprocessing of input and output have to be defined in the configuration file – which eventually turns them into "programs" in the sense of section 6.2. To mimic the behavior of the current `alias_db` module, a configuration language needs to be defined that is able to split and concatenate values in arbitrary ways, passing sub-strings to arbitrary back-ends. $f_{pre/post}$ are defined in the configuration file; a unified data definition language creates a common knowledge about the structure of data between the OpenSER modules with the configuration language. Processed data are then passed to a database API that decides on the back-end to be called based on the configuration. f_{SQL} is substituted by a unified f_{query} which in turn passes

values to underlying new functions.
This type of architecture is similar to a three-tier-architecture such as CORBA.

- (5) Only a thin path from the OpenSER core to arbitrary configurations exists at (5). The expression power is maintained through a "real" programming language. $f_{pre/postprocess}$ and $f_{SQL/LDAP/query}$ are no longer defined in OpenSER itself but are fully defined and implemented within the configuration. An appropriate infrastructure helps administrators in this process.
  The well known web server software "Apache" features mod_perl to accomplish arbitrary administrative control.

6.4.2. Evaluating the options

Each of the options above has unique advantages and disadvantages, some of which will be discussed here.

	Pros	Cons
(1)	The administrator does not need to configure the database. Setting up the system is easy.	Flexibility is lost. Adaption to new database engines is as difficult and expensive as the integration of new modules. The setup of new data paths is only possible by modifying OpenSER's source code.

	Pros	Cons
(2/3)	The current OpenSER structure can remain untouched. Existing implementations do not have to be modified. Most of the current system layout can be transported to newly implemented modules for the desired functionality.	Every pair of data source and sink needs a distinct module, or each module needs to know about every possible pair of data source and sink. E.g., the `alias_db` module needs to be changed when a different database schema such as a one-figured LDAP schema needs to be utilized. Implementing this would result in a large number of new modules that compensate current weaknesses, but the gained flexibility is restricted to the modules where the features are implemented.
(4)	A clean abstraction layer between data sources and data sinks exists. The exchange of a data source in the context of a certain module (data sink) is simply done by changing a set of configuration options.	A new programming language has to be defined, and an interpreter for this language has to be implemented. Currently implemented modules have to undergo far reaching modifications. Currently, the module is able to use its knowledge of the structure of the data it handles. This ability is lost for the sake of flexibility. The design of an abstraction layer and porting the current interfaces to this would create the necessity to re-implement/modify the current database API and thus all modules based on it – a laborious task. In the context of actively used open source software, it is questionable whether the OpenSER maintainers would accept such a far reaching change.

	Pros	Cons
(5)	Maximum flexibility is given to the administrator. He is able to configure arbitrary system behavior without modifying the OpenSER sources. As long as the incorporated programming language is able to access any data source, it can be used in OpenSER. Data paths completely different from the ones currently considered become possible.	The administrator no longer needs to be able to "configure", he rather has to "program". Generation of dead locks and other software bugs in the configuration are a possible consequence. These disadvantages can be largely concealed by creating an extensive "programming library" that covers the most common tasks.

Without any doubt, option (1) is not a good choice. It reduces flexibility below the current level while increasing the need for low level programming.

Option (2/3) will be the easiest choice, but the disadvantage of having to implement OpenSER module code for each and every pair of data source/sink outweighs it's usefulness. The inherent lack of flexibility would just be transferred to a new back-end. Realizing this option is a programming task and thus will not be done here.

This leaves the choice between a major redesign of the current database access mechanisms including the reimplementation of large OpenSER parts on the one hand, and the integration of a programming language on the other.

While choice (4) has the advantage of a clean design that exactly fits the needs found in the requirements analysis, the scope of this project would vastly be increased by integrating a full featured programming language in OpenSER in the sense of (5) – far beyond the range of database access. Due to this, the implementation will be based on the concept (5).

Integration of programming languages in software systems for user interaction is described as "End User Development" (EUD). In the next part, the idea of EUD will be discussed and a number of other EUD implementations will be evaluated. This examination will support the process of designing a sensible environment.

Part III.

End User Development

7. End User Development in dynamic systems

As discussed above, the integration of a programming language in OpenSER will create a large value and lots of new opportunities. The idea of integrating languages can now be extended to software systems in a wider range.

The idea of scripting software systems to dynamically modify their behavior is not new. A large number of standard and domain specific software is equipped with programming interfaces [44, 47].

Recent research has coined the term "End User Development" (EUD) [35] to denote software extensions and modifications by end users. The benefits of EUD have been examined [53] in detail. Although scripting and macro languages are a key to EUD, this paradigm may be more broadly understood when it comes to scientific computing. In that context, EUD also includes e.g. bioinformatics programming using specialized programming libraries. Research on EUD has been focused on social and political aspects rather than on technology. This chapter will thus examine particular implementations of EUD environments under various aspects.

To achieve a deeper understanding of how embedded scripting languages work in software, some embeddable languages will be described first. A number of software systems with scripting integration will then be analyzed and their common features will be discussed. This will lead to a better comprehension of how the embedded scripts can modify a system and how transactions between the software itself and the scripts are handled. The analysis will help in choosing a sensible EUD language and the appropriate design of the environment.

7.1. Embeddable languages

Before discussing applications that integrate scripting abilities, a number of programming languages that are commonly deployed for such tasks will briefly be described. The list does not claim completeness, but shows different aspects of integrated languages.

The focuses of the languages differ widely. Some of them are domain specific languages (DSL, [57, 55]) that are specifically designed to support the application they run in, while others are general purpose languages with an integration framework.

7.1.1. Perl

One of the best known scripting languages in the open source community is Perl. The word Perl (with a capital P) denotes the language as such, while perl (with a lower case p) is used for the interpreter. The language and its single implementation were initially developed by Larry Wall. The first version was released in 1987, while the core language as it is known today was released as Perl 5 in 1994.

Starting in 2006, work on Perl 6 has begun. Language syntax as well as the runtime environment are being completely reworked. Currently, the availability of a stable version is not yet be foreseeable. Even after the availability of Perl 6, version 5 will be supported on a long-term basis, as both systems – Perl 5 and 6 – will not be source code compatible.

Initially, Perl was focused on simple text processing, but has since evolved to a general purpose language. Focus was put on ease of use. This led to a dynamic (implicit) type system that can help in the development of small scripts, but creates a number of problems in larger scenarios.

Perl is equipped with a powerful embedding framework for C/C++ programs. The interpreter is available as a shared library and can be linked into arbitrary programs. Multiple C function calls allow C programs to access Perl functions, objects and variables inside the interpreter. On the other hand, so-called extensions make it possible for Perl scripts to modify values or call functions that are implemented in C. Programming languages other than C and C++ are not supported.

One of the main strengths of Perl is not the language itself, but its huge source code archive, CPAN (Comprehensive Perl Archive Network). This repository offers more than 4000 modules and numerous bundles of modules for all aspects of computing.

7.1.2. PHP

PHP, the PHP Hypertext Processor, was designed as a domain specific language to be integrated in web server mechanisms. Its primary target is the simple evaluation of program sections in HTML pages delivered by a web server. Special attention was given to the access of relational databases. In contrast to client side scripting (JavaScript, VBScript), PHP scripts are executed by the server.

PHP's ancestors in terms of language design are Perl, Java and C.

During its history, PHP has been extended to a general purpose language. Although it is well possible to write programs that are independent from the context of a web server, this option is rarely used.

PHP has been integrated into multiple web servers. Yet, the embedding framework for arbitrary C/C++ programs is marked as experimental code.

7.1.3. Lua

Most languages discussed in this section were designed either distinctively for a single application, or as general purpose languages that "coincidentally" support embedding. Lua was developed especially as a scripting language for embedding. Historically, it has often been used in graphically focused software and computer games.

Lua's scope is a rather simple control over the scripted environments. Its primary advantage over other languages listed in this chapter is the fact that it can instantly access C data structures, making their evaluation and modification extremely simple. All other languages listed require a more or less sophisticated data transformation.

7.1.4. Python

Python is a multi-paradigm language that features a combination of imperative and functional aspects [20]. Unlike many other scripting languages, Python features an explicit typing system [33].

While the names of the other languages discussed in this section often refer to the language itself as well as to its implementation, there are multiple implementations of this language, each with its own pros and cons. Python has a big, active community that has created a valuable collection of modules comparable to Perl's CPAN, although the number of packages in that archive is significantly lower.

7.1.5. Ruby

Similar to Python, Ruby was designed with simplicity in mind ("Principle of least surprise", POLS) [17]. Although it is slightly more related to imperative programming, functional syntax is also possible in many cases.

For quite a time, Ruby did not receive a lot of attention, as it was only available with Japanese documentation. This has changed during the last years. The language has recently gained a lot of international attention through its new web framework "Ruby on Rails".

7.1.6. LISP and Scheme

LISP, the LISt Processor, was one of the first functional programming languages. Designed in the 1950s, it was initially implemented as a language to represent the lambda calculus in a computer. As such, is does not reflect the sequential execution of statements as present in imperative programming languages and inherent to the von Neumann architecture. On the other hand, multiple concepts can be expressed in functional statements in a very short, elegant and comprehensible way.

Scheme is a dialect of LISP, but extends the language with a number of imperative concepts.

7.1.7. Basic dialects

BASIC, the "Beginner's All-purpose Symbolic Instruction Code", is a programming language with a long tradition of being used by inexperienced users. Not only was it deployed as the standard command environment on early home computers, it was also the foundation product of Microsoft. In later years, BASIC was repeatedly implemented as an integrated language in all kinds of desktop and workstation software, especially by Microsoft.

In Microsoft Office, WordBasic and later Visual Basic for Applications (VBA) displaced earlier macro languages. VBA was also licensed by other vendors and has been integrated e.g. in AutoCAD or WordPerfect. Communication between VBA and the host application is realized through the OLE interface.

StarOffice and OpenOffice use their own Basic dialect as an EUD environment.

7.1.8. Other languages

Software systems of all kinds have been equipped with embedded programming or scripting. Some of them were called “macro languages”, as their primary focus was the sequential execution of user input. Most of these languages have been dropped in favor of more widely accepted languages, such as the ones described above.

So-called “shells” in UNIX environments can be regarded as embedded languages of the operating system.

7.2. Analyzing EUD implementations

Countless modern software systems with many different scopes are equipped with scripting abilities, ranging from desktop software such as office suites over networking clients (e.g. chat software) to industrial strength server software. One of the best-known and most commonly deployed scripting frameworks is Apache’s mod_perl.

Although the services provided by these pieces of software differ largely, their scripting interfaces provide a level of similarity by creating a means for the user[1] to modify or define the system’s behavior in reaction to certain events.

The examined solutions are examples from different areas of computing and represent alternative approaches to the problem.

7.2.1. Apache and mod_perl

The Apache HTTP Server is the leading web server software [36]. Although a number of script languages can be integrated into the open source project [19], the most sophisticated framework is the Perl module mod_perl, as it allows for much more than “only” delivering dynamic content.

The mod_perl website states the following features on their “What is mod_perl?” site [7]:

- Accelerate your existing dynamic content
- Easily create custom modules that become part of Apache
- Gain access to all request stages

[1] The term “user” in the context of server software refers to the system’s administrator, as opposed to the end user that uses the service.

- Configure Apache with Perl
- Install Third-party modules
- Application Frameworks
- Apache 2.X support
- Active Support Community

Although the primary usage of mod_perl is the accelerated delivery of dynamic content, similar to the execution of CGI scripts, the feature list vastly extends this option.

Apache is a modular system. Modules can register with different types of handlers [21], only one of them being the content handler. The Perl module hooks into the standard module chain and thus provides methods to largely modify Apache's behavior. By using the server's API, the "glue layer" allows Perl scripts to access the underlying core technology of Apache. A list of categories of functions in the Apache server API can be found in [21, p. 51]. A Perl extension creates an access path for perl scripts to the API functions.

The mod_perl project has been initiated in 1996 and has since evolved to a considerable part of the Apache project.

As mentioned, the primary target of mod_perl is the generation of dynamic web content. Upon receipt of a page request, Apache calls a Perl function[2] and returns its output as a web page. It is completely up to the script's responsibility how the function generates its output: Commonly, databases are accessed, but the returned data may as well depend on any other information available for the system.

Besides that, the scope of mod_perl is widely extended by its class library. Methods of its classes let the module access a large number of Apache internals, e.g. its configuration tree.

7.2.1.1. Analyzing the framework

It would be out of proportion to fully analyze the class structure of mod_perl's library. Nonetheless it will be helpful to examine the way Apache interacts with the mod_perl.

mod_perl's API is structured in three main namespaces[2], `Apache::`[3], `APR::` and `ModPerl::`. While the `Apache::` namespace contains classes that provide access to

[2] Depending on the chosen setup, the function may be the "anonymous" function defined by a script's body

[3] Replaced by `Apache2::` in Apache 2.x/mod_perl 2.0

low level Apache internals – such as processes or connections, the `APR::` namespace provides the glue layer for higher level functions from the Apache Portable Runtime API. `ModPerl::` represents the set of classes that provide access to the functions that are unique to mod_perl, e.g. the exploration of the Perl interpreter's namespace (`ModPerl::MethodLookup`).

7.2.2. Office suites and macro languages

In the 1980's, text processing programs such as the famous WordStar began to integrate macro languages in their products. Through these means, the user could automate frequently occurring steps of their work. This technique was soon adopted in a wider range of end user programs such as the ubiquitous spreadsheet software.

While in the beginning the definition of macros was merely the notion of a sequence of key presses, modern office suites are equipped with full-featured programming languages. Microsoft introduced WordBasic in Microsoft Word that was later substituted by Visual Basic for Applications (VBA). In StarOffice, the language BASIC was adopted, eventually forming today's OpenOffice.org Basic.

Most modern office suites consist of a text processing program, a spreadsheet, graphic software and database management system. Their common feature is the usage on end user desktops, but the functionality, scope and user focus largely differ.

In text processing software, the primary objective of an integrated language is more or less simple macro processing, e.g. switching between certain text formatting options or sequentially inserting external data. More sophisticated programs might carry out dynamic search-and-replace processes or remove unwanted data.

The execution of functions in the context of spreadsheet software is a little more interesting, where mathematical calculations can be implemented in a programming language. This enables the user to carry out powerful calculations without being restricted to the – usually limited – subset of functions offered by the spreadsheet system itself.

In database management systems (DBMS), SQL has been established as the standard processing language. Unfortunately, this language lacks a number of concepts. While regular relational DBMS have facilitated Stored Procedures and integrated languages such as Oracle's PL/SQL, Microsoft's desktop DBMS provides an API for Visual Basic for Applications.

A sample of a scriptable graphic program, although not part of an office suite, will be discussed in the next section.

7.2.2.1. Technical aspects of Visual Basic for Applications

VBA offers the full range of full featured programming languages. In the context of an EUD environment, however, it is primarily targeted at controlling the host software system. The main API that is used in this context is the OLE API offered by the software; additionally, the interfaces of all .NET classes are available. VBA thus offers a wide range of usage from "simple macros" to "almost full featured integrated applications".

The downside of this omnipotent EUD environment is that document macros can also be used to spread viruses and other malware by tricking users into opening malicious files.

7.2.3. Gimp and Scheme

One of the first publicly observed open source programs was the graphic program Gimp. It provided a remarkable power as free software. One of its strengths was the integrated scripting plugin "Script-Fu", based on a "small footprint Scheme interpreter" SIOD [4]. Although today there are interfaces for other languages as well, Scheme is still the most commonly used language in Gimp.

A wide range of scopes exists for scripts:

- Creation of graphics. A web designer can create graphical buttons for web sites, but with different text
- Automated processing of graphics. A photographer could eliminate or add certain properties to his pictures
- Web services can control Gimp to react to user input, creating or modifying graphics

Script-Fu is able to control most aspects of Gimp through a dedicated interface. The Procedural DataBase (PDB) stores information about the functions that are available from the core system or from plugins. By calling these functions, scripts have the same control over the system that the user has.

7.3. Properties of EUD environments

A set of environments that embed programming languages in software systems for different contexts have been demonstrated above. Their obvious common feature is the

user's opportunity to automate responses to recurring events in the system. The focuses of the discussed systems differ widely, however.

In interactive desktop software, the EUD environments focus on automation of steps which can – theoretically – equally be executed by the user. The interface to which the embedded language is bound is on a similar level as the menus, widgets and forms in the user front-end. In server software such as the Apache/mod_perl combination discussed above, the EUD environment is integrated to define the system behavior in reaction to server requests.

7.3.1. Interfaces

All systems discussed above provide programming interfaces (APIs) for the language environments they integrate. Formally, an interface is a set of declarations of functions or object methods and the data types and variables on which they may operate. While traditional APIs provide type signatures for the same programming language for which they are developed, the EUD interfaces provide a programming interface for a different language. Thus, a mapping of type signatures from the host software to the EUD environment has to be developed.

The EUD programming interfaces provide a level of abstraction to access the internal functionality and data in a uniform way. A mapping of the internal software design to the API available for EUD has to be found. This leads to the following questions:

- Who is the average user of an EUD environment?
- What are the constraints regarding the complexity of the API?
- What functionality and data should be accessible? Does the answer to this question depend on the host software system?

The answer to the first question results in answers to the other issues.

The expected end user in EUD environments will often not be a qualified software engineer, but rather a regular software user. End users will often have only a poor knowledge of the system, about the programming language, and about programming concepts and software quality.

This suggests an answer to other questions: The API provided should be as small as possible, but still include enough functionality not to restrict the user in the development

of his plans. An inappropriately complex API will make the system overly hard to understand and to learn. The exposed functionality should be restricted to the anticipated amount necessary for the users.

A coverage of every internal function

- confuses users with an excessive amount of functionality
- is a source of errors
- unnecessarily reveals internal complexity

7.3.2. Design constraints

The EUD environments discussed in this chapter focus on different aspects of computing. The comprehension of programming will largely vary among their users. A scientist using a spreadsheet software may know a lot about the basic mathematical constructs for his calculations, but not much about the way these constructs are conveyed to a piece of software. While a web server administrator may well have fundamental knowledge about the way computers work (and – in this case – networks and client/server architectures), a photographer using a graphics software such as Gimp may not. On the other hand, Gimp may as well be used by a software developer to modify images on the fly. These differences between users of the same software make it difficult to find custom characteristics and necessary features of EUD environments.

It is thus not possible to fully define globally valid qualities for possible EUD environments. Instead, a number of questions that should be considered during the design of such a system can be collected:

- What is the expected user profile?
- How much user knowledge about underlying technology and programming is expected?
- Which applications and use cases can be anticipated?
- Will it be sufficient to provide a domain specific system, or will a user need a full featured programming system?
- How much system internals need to be exposed?

Answers to these questions for an EUD environment in OpenSER will be given in the next chapter.

Part IV.

Design, Implementation, Testing

8. Design

In the first half of this work, a fundamental concept was developed: The integration of an EUD environment in the OpenSER SIP server shall give the user the flexibility to access arbitrary sources of information to modify the routing of SIP messages. In this chapter, a design of the necessary implementation details will be developed.

8.1. Considerations on an EUD environment for OpenSER

The results of the studies in chapter 6 suggested the implementation of an EUD environment in OpenSER. As comparable systems were examined and their common constraints were derived, answers to the questions developed in the last chapter will now be given to help to design a sensible OpenSER environment.

In the technology analysis of chapter 3 OpenSER has been demonstrated to be a software that is not simple to install and to maintain. Thus, it can be expected that its users will be qualified system administrators that do have a considerable amount of experience with SIP based systems. Adapting OpenSER to local circumstances requires users to become acquainted with the available modules and their functions.

Although end users cannot be expected to be qualified software developers, basic knowledge of script programming languages can be assumed. The best known scripting environments on Unix based systems will probably be the shell interpreters (bash, (t)csh...) and the programming language Perl discussed above. While the development of a domain specific language for SIP processing may well be possible, an already known, stable and widely available language will be easier for administrators to use. The risk of missing necessary features is lower when using a general purpose language.

The use cases examined in the requirements analysis will define a subset of the anticipated use cases of the EUD environment. On the other hand, integrating a full featured programming language will create lots of new opportunities that will create new use cases.

8.1.1. Choosing a language

In the last chapter, a set of commonly used embedded languages was discussed. The following table demonstrates advantages and disadvantages of the options. Both user and technological aspects are listed.

Language	Pros	Cons
Perl	• Widely spread • Widely known by system administrators • Large and technologically advanced class library • Appropriate for short, simple functions as well as for complex programs • Powerful string processing with regular expressions • Base technology in Collax products	• Can be quite confusing, depending on usage
PHP	• Comparably simple and easy to learn • Good integrated database access	• Embedding framework and language concepts only apply to context of web servers

Language	Pros	Cons
Lua	• Very good integration framework	• Not widely known • Small class library, no third party module repository. LDAP and SQL bindings are available on the net, however
Python	• Modern, multi-paradigm script language • Ambitious projects evolve around python in the Unix scene	• While similarly focused, not (yet) as widely known as Perl
Ruby	• Modern, promising language	• Not (yet) widely known • LDAP access only available through third party module
Lisp	• As a functional language, it seems to be well suitable for the concept of defining the functions f_{pre} and f_{post} • Multiple implementations to choose from	• Multiple implementations to evaluate • Database access and regular expressions only available as third-party modules • Not easy to master by users who are accustomed to imperative programming

Language	Pros	Cons
Basic	• Widely known • According to its name, suitable for beginners • ScriptBasic provides integration framework	• Data types in standard Basic not sufficient for sensible data handling • Limited libraries available
Others	• Domain specific language would fit needs best • Lots of options available	• Developing a new language is complex • Evaluation of more choices increasingly costly

While the choice of a sensible embedded language influences some design aspects and obviously the implementation, more than one language would fit the project's needs.

Three primary reasons fix the decision for Perl:

- Collax as this project's initiator uses Perl
- Perl is well known among system administrators, the intended audience
- Perl's implementation and embedding framework is stable and well supported

8.2. Data paths

The integration of Perl to open data paths in OpenSER has now been fixed as the resolution method of this thesis' goal. Coming back to the definitions of section 5.3.1, it remains to be defined to which data paths the Perl functions should attach. The functionalities defined here will be designed later.

Core/module interface should be used for embedding of simple Perl function calls which can define OpenSER's routing. Answers of simple database requests can directly be injected into the routing decisions.

Module/database API While the available database API should not be substituted, a module can be attached to the database API to provide access to arbitrary data. This module will be a database module comparable to the currently existing relational database bindings.

database module/technology Perl features packages for many given database technologies, including LDAP and Berkeley DB. The database module/technology interface is not an internal one, so users of the database module are free to use any technology binding available.

Module/module While it will be possible for other modules to attach to the core/-module interface ("incoming" from the perspective of the developed code), outgoing requests should be made available through interfaces to module functions and central APIs. Initially, a separate API will not be provided.

MI interface An MI implementation is not necessary at the moment.

The implemented bindings thus will target towards the core/module interface and the module/database API.

8.3. Design patterns

Design patterns are traditional methods of software engineering to find and solve commonly recurring issues in software development. The topic is rather complex, so the following discussion will only briefly describe two patterns that can be found in this context.

8.3.1. The bridge pattern

The bridge pattern is meant to "decouple an abstraction from its implementation so that the two can vary independently" ([18]). The bridge uses abstraction mechanisms

(depending on the programming languages) to provide an interface to different "classes" that provide a defined set of functionality.

The database API follows the bridge pattern. While the implementation of certain database modules and the data they return can vary, a single bridge from regular modules to the database back-ends is provided by the DB API.

While the normal module implementation only abstracts the module binding mechanisms, the core/module API also follows the bridge pattern.

As the implementation of bindings to these interfaces is planned, the bridge pattern is used "unknowingly": the APIs are already given and will not be modified.

8.3.2. The adapter pattern

When two objects with different APIs are given, they cannot exchange data. An adapter can "adapt" the interface of a given class into the one expected by a client class. A non-software example of this would be a USB to parallel port adapter for printers.

The adapter pattern is also referred to as the wrapper pattern. Like many other design patterns, it is more or less directly connected to object or class oriented design, so its applicability in the context of the purely imperative OpenSER is restricted. It will be reflected by the adaption of non-relational data interfaces to the relational module requests, however.

8.4. A Perl module

As the OpenSER's module API is given, the design of the Perl module has to follow the given structure.

OpenSER modules contain functions that can be accessed by the core. During SIP message processing, a reference (pointer) to the SIP message structure, and (possibly) two user-defined string values are passed to the functions. The concept for the Perl module includes a pair of functions that call Perl functions, passing them the same parameters. While the first function, `perl_exec()`, will be equivalent to regular module functions, the other one, `perl_exec_simple()`, will not be passed a reference to the SIP message. The latter may be used for trivial requests that return an information independent of the SIP message content, e.g. based on the current time ("no calls outside office hours"), or based on the information that they were triggered (increase a message counter).

On possession of control by the Perl function, it may evaluate or modify the message. To do this, it needs getter and setter functions for the SIP message. Getter functions will return parts of the SIP message (that was already parsed by OpenSER), such as the request method, or the recipient URI. Setter functions will let it modify the RURI and internal properties and flags used by OpenSER. The availability of textual modification is not necessary, as other modules (especially the `textops` module) provide functions for this task.

In most "real world" cases, only minor parts of the SIP message – such as as the RURI – will be evaluated in the Perl script. Transforming the structure to a Perl equivalent and re-transforming its output would have a big performance impact. Instead, the C pointer (`struct *sip_msg`) can be wrapped into a Perl structure of a dedicated, newly created Perl class. This class will be called `OpenSER::Message`. The getter functions in that class will return Perl representations of the strings contained in the structure. Thus, the functions that have to be implemented are pre-determined by the attributes of the `sip_msg` structure.

A SIP RURI consists of different parts, such as user and host names, ports, or transport methods. The OpenSER parser creates a structure `struct *sip_uri` inside the `sip_msg` structure that contains the parsed components of this URI. Although other URIs in a message are not parsed, an additional class for URIs will be created. This class will be called `OpenSER::URI`. It will have additional getter functions for each component of the URI.

Regular OpenSER scripts often use pseudo variables and AVPs. While pseudo variables are properties of a SIP message, AVPs are not. Thus, a function `pseudoVar(string)` can be implemented in the `OpenSER::Message` class, parsing the argument for pseudo variables and substituting these by their values. AVP getter and setter functions are defined in the third package, `OpenSER::AVP`.

Other core access functionality will be put in the base package, `OpenSER`. While the AVPs in fact are core functionality, their distinct nature justifies a distinct package.

The Perl API thus shall consist of the following functions:

- Core access: logging
- Message access:
  - "firstline": Getters for message type (request or reply), status code and reason in replies, SIP method and RURI in requests, SIP version string. Setter for RURI.

  - RURI: Getters for all available attributes
  - Headers: Getters for full header, single header (identified by parameter), header names
  - Message: Getters for full message, message body
  - Flags: Getters and setters for OpenSER's message flags
  - Additional functionality: Call of other module functions, branching, pseudo variable evaluation

- AVPs: get, set, destroy

8.5. Virtual database

While the Perl module provides a straight forward EUD environment for SIP message handling, the original requirements specified the opening of data paths from currently existing OpenSER modules to arbitrary data providers. The Perl integration will thus be used to implement a database module that can relay the relational database requests to Perl functions. These user provided functions in turn can operate on arbitrary technology back-ends.

This module will be referred to as the Perl Virtual Database, or VDB.

As the VDB is working on top of the Perl module, it will add to its namespace `OpenSER::`.

The subset of relational operations available through OpenSER's database API includes insert, update, delete and query. Additionally, raw SQL queries may be sent to the DB layer, which is not appropriate in the context of this module. A fetch operation can fetch parts of the current query result; this will not be realizable in a sensible manner. Currently, existing modules do not use the raw queries. The `usrloc` module can use the fetching capability, but will work without it. Both raw query as well as fetch support will thus not be provided by the VDB module.

In the relational algebra and calculus, queries can include the operations of projection, selection, union, set union and set difference ([30]), which are reflected by the respective SQL statements/concepts "select", "where", joins, "union" and "except". The OpenSER query operation only defines projection and selection on single tables.

OpenSER database modules can define a subset of the six possible DB API functions. Four functions for insert, update, query and delete will be provided. After the argument

mapping to Perl types, the Perl functions of a (configured) Perl package is called. These packages have to inherit from an abstract base class, `OpenSER::VDB`.

8.5.1. Class structure

The database API defines the type signatures of the supported operations. To understand these signatures, the low level data structures need to be examined:

- A `db_key_t` is a string. The term refers to a column name in an SQL table rather than to a key in the relational model.
- A `db_op_t` (operation) is one of the strings <, >, =, <=, >=, !=.
- A `db_val_t` (value) is a structure with a type identifier, and a variable of that type. The types available are INT (integer), DOUBLE (floating point), STRING (C string, i.e. `char *`), STR (OpenSER's extend strings, i.e. a C string plus it's length as an int), DATETIME, BLOB (Binary Large OBject) and BITMAP (an integer containing a set of bits).
- A `db_row_t` is a structure containing an array of values, and the number of elements in that array
- A `db_res_t` (result set) is:
  - A structure `col` containing two arrays for the names and types of columns, and a column counter
  - An array of `db_row_t`
  - The number of rows (three different variables are used by database modules that provide the DB_CAP_FETCH capability)

These types reflect an intermediate level of abstraction between the C source code and the relational model. There are no container types for each of the operations. The analysis also shows that a result contains two parallel type definitions for each contained value: one in the column definition, and another one with each value. While this is not well designed, it will usually not create problems, as the database modules will set identical information in both variables.

The DB API functions operate on these types:

- insert takes an array of keys and an array of values, plus a counter.

- update takes three arrays (keys, operations, values) to select the rows to be modified, and two arrays (keys, values) for the data to set, plus two counters for the lengths of these arrays.
- delete takes three arrays (keys, operations, values) to select the rows to delete, plus a counter.
- query takes three arrays (keys, operations, values) plus their length to specify the select, an array of column names to return plus its length, the column to order by, and a pointer to a `db_res_t` structure in which the result is stored

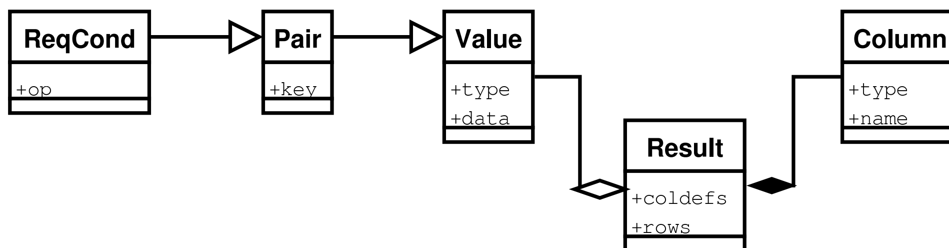
Each of the functions returns a boolean success value. While queries theoretically "return" the data set, they do not use the C `return` statement. Instead, the result set is passed as a call-by-reference argument.

The structure shown here provides only a limited level of abstraction. In Perl, container classes that aggregate variables of these requests will be created:

- An `OpenSER::VDB::Value` has the same elements as OpenSER's `db_val_t`
- An `OpenSER::VDB::Pair` inherits from `::Value` and adds a (column) name
- An `OpenSER::VDB::ReqCond` inherits from `::Pair` and adds an operator attribute
- An `OpenSER::VDB::Column` contains a column name and a column type
- An `OpenSER::VDB::Result` contains an array of `::Columns`, and an array of arrays (each containing a row) of `::Values`

Each of the database functions inside the OpenSER module will convert the C variables to a Perl representation. When a query function is implemented in Perl, it can directly return an object of class `OpenSER::VDB::Result`, which is not (simply) possible in C.

These classes are visualized in the following class diagram:



The Perl functions need to take these argument types:

- Insert takes an array of `::Pairs`
- Update takes an array of `::ReqConds` and an array of `::Pairs`
- Delete takes an array of `::ReqConds`
- Query takes an array of `::ReqConds`, an array of strings for the requested columns, and a string for the sorting column

Array length parameters are not needed, as Perl implicitly stores array sizes.

8.5.2. Adapters

When a client module requests data from a database, it specifies the table to query with a call of the `db_use_table` function. The table to be used has to have the schema the receiving module expects. Due to this, it is necessary to implement Perl functions that reflect the correct schema that is used by the client module.

Thus, the concept of adapter classes that can be used in conjunction with client modules is introduced. For every module that needs to cooperate with the VDB module, an adapter class is implemented. This adapter transforms the parameters of the types above into simple imperative-style parameters for Perl functions. These adapters must inherit from the base class `OpenSER::VDB`, resulting in classes in the namespace `OpenSER::VDB::Adapter::`.

Virtual databases in turn contain one or more virtual tables; these provide the concrete (imperative) functions that will finally return the requested results. While the adapters should be part of the code package, the VTabs will have to be implemented by a user. For his convenience, a number of options are made available:

- User defines a single function. In this case, pass the operation name (insert, update, query, delete) as the function's first parameter. Numerous client modules use only a single database operation function (e.g. insert for `acc`, select from `alias_db` and `speeddial`), so a single function may often be sufficient.
- User creates a Perl package (class). In this case, the class is initialized with a call to its `init()` function. Subsequent calls to its operations are function calls to the functions `insert()`, `update()`, `query()`, `delete()`
- User creates an object. This will provide better mechanisms to store internal data. The class is instantiated with a call to a `new()` function (Perl does not really have

constructors); the `insert()`, `update()`, `query()`, `delete()` calls are then object method calls instead of function calls[1].

[1] The difference in the Perl semantics is small but existent: Functions receive the class name as their first parameter, while methods receive a reference to their instance – similarly to the Java/C++ `self` variable

9. Implementation

Based on the detailed system design, the implementation process was straight forward. As there are no sophisticated algorithms or data structures involved, the problems encountered primarily were of technical nature, although not always trivial to solve.

In this chapter, a number of technical details will be discussed which had to be taken care of during the implementation phase. The tools involved will be briefly listed, the major technical hurdles will be explained and their solutions outlined.

The outcome of the implementation phase were two OpenSER modules and accompanying Perl code.

9.1. Tools

The tools involved during an implementation process are the technical basis on which the system is built. This gives them a strategic significance and makes their choice an important one. Although this choice was largely predetermined by OpenSER's build environment, a short comment on the tool chain will be given.

9.1.1. Build environment

OpenSER is implemented in C. Although it is not explicitly specified, its language style aims to be standard compliant C. Still, only the GNU, Sun and Intel C compilers are supported in the build environment. The Perl and VDB modules were developed with gcc 4.1.0; frequent compilation tests with version 4.1.2 were done.

In contrast to a majority of open source software, the OpenSER build environment does not depend on a configuration environment such as GNU autotools (autoconf, automake) or Scons (an up-and-coming build tool), but rather on manually maintained Makefiles for the traditional (GNU) make. The predefined OpenSER Makefiles provide a flexible, simple to use basis for the creation of new modules.

The embedding of a Perl interpreter adds a small number of special tools to the queue. Perl extensions such as the one implemented are written in a meta language called XS.

Although the actual functions are implemented in C, their interface description is a little different. The xsubpp preprocessor transforms this meta language into pure C.

9.1.2. Revision control

Another central component of software development is a versioning management system. Again, OpenSER's choices preset the used tools in a certain way. Before the transition to Subversion in February 2007, CVS was used for the OpenSER development. CVS has a number of well known shortcomings:

- Directories cannot be removed from the repository
- Files and directories cannot be renamed without losing their history
- There is no notion of a "coherent system revision". Every file carries its own revision number

To avoid these problems, the work on this thesis (including documentary as well as implementation work) was supported by Subversion, a modern versioning management system that eliminates most of CVS' weaknesses. In quite a short period of time, Subversion has become CVS' successor as the standard versioning management system in the open source community, although it has to be seen as an "evolutionary successor" rather than a revolution. Numerous valuable new concepts of other modern versioning systems such as distributed repositories or multi-branch merging are not implemented in Subversion.

Due to the fact that the code that was produced for this thesis was integrated into OpenSER's core repository at a very early stage, this led to the problem that there were two concurrent software repositories with almost, but not totally identical program code. It would have been possible to represent OpenSER's code base as a separate branch in the private repository. The system's merging features could then have been used to transport modifications in either branch to the other one. As branching and merging create other obstacles, this idea was not implemented.

9.2. Development process

After the evaluation of the results from the analysis that led to the concepts of an End User Development environment and the integration of Perl, numerous prototypes for

multiple parts of the projects were developed. The embedding of Perl was examined (including the "dlopen problem" described below), Perl extension stubs were implemented, LDAP tests were done and database access was evaluated.

Based on the results, the development phase was straight forward and only few obstacles were encountered. Thus, the resulting program code was soon submitted to the OpenSER core repository, which was by that time a "HEAD" branch independent of the stable 1.1.x version branch. Development continued in this branch, so a small number of patches to the code was integrated to allow the continued use in newer code versions.

These modifications were done by the OpenSER core team:

- The API of the "`stateless`" module was changed to provide a function easily accessible from other modules. The Perl module was updated to use this function instead of the deprecated style.
- The FIFO functionality was replaced by the new Message Interface. A new interface to the FIFO functionality of the Perl module was added.
- A compilation bug fix on obsolete Perl versions was applied a second time for a second occurrence of the same bug
- 64bit compilation warnings were fixed
- The core "action" data structure type was modified during the development phase. The OpenSER team modified the Perl module's use of this type.

Obeying the (implicit) OpenSER coding guidelines, the implemented modules reside in respective subdirectories in the `modules/` directory hierarchy; the Perl module is called `perl`, while the virtual database is called `perlvdb`. As both share a single Perl interpreter instance, the Perl libraries of both modules reside in a shared `lib/` subdirectory of the Perl module. Both source code trees have distinct documentation directories, which in turn contain a samples directory.

9.3. Embedding Perl

Providing an embedding framework was a primary design target during the development of Perl. The Perl documentation features a set of articles on the C language interface [3] that gives a detailed introduction to the embedding mechanisms.

As can be seen in the design concept of the Perl module, there are two ways of communication between the modules and the executed Perl functions: In a first step, the module code has to execute a function or method that is implemented in Perl. The returned data have to be evaluated afterwards. These steps are roughly outlined in the "perlembed" document. Evaluation and creation/instantiation of Perl variables within C is more thoroughly described in "perlguts" and "perlapi", while the detailed semantics of the different ways to execute perl functions is clarified in the "perlcall" article.

9.3.1. The Perl interpreter

On Unix style platforms, the Perl interpreter is available as a shared object library (`.so` file). Some convenience wrappers make it easy to link against that library and include the correct header paths in the compiler run.

Inside a C program, a Perl interpreter is an instance variable of the data type `PerlInterpreter`, usually called "`my_perl`" (if a different variable name is used, an additional C preprocessor macro has to be executed each time the interpreter context is switched). This interpreter variable is initialized by a small set of functions. Eventually, the interpreter is ready to either evaluate script snippets located in the C source code (with the `eval_*` functions), or it can parse and compile a conventional Perl file (via `perl_parse`). If scripts contain an initialization (predefined variables or global script content), the script can be run via `perl_run`.

After parsing a file, the `call_*` functions are available to invoke Perl subroutines defined in the script. This is called *callback* in the Perl context.

9.3.2. Data types in Perl

Perl uses a different way than C to pass arguments to and return values from functions; additionally, the available data types differ. Even semantically identical types – such as strings or integers – are represented differently. Before calling a callback, possible parameters have to be converted to Perl types; after the function returns, its return value(s) need to be converted back. There are three primary data types in Perl, which are Arrays (AV), Hashes (HV) and Scalars (SV). More specialized data types are certain sub-types of scalars, e.g. integers (IV), strings (PV) or references (RV) [10]. As Perl knows – unlike C – the concept of introspection, it is possible to request the sub-type of a scalar value.

Perl's notion of an object in the sense of object oriented programming is merely a reference to a value, usually a hash, "blessed" into a class, so an object itself is a scalar (SV) with sub-type RV. Blessing means that the variable reference is "attached" to a class that provides methods for the variable.

9.3.3. Perl memory management

While traditional compiled languages mostly rely on explicit, developer-controlled memory management, most interpreted languages – including Perl – feature a dynamic garbage collection mechanism. In case of Perl, this is accomplished by reference counting [34].

Every object stored in memory is supplemented by a reference counter. Each pointer or reference to this object increases this counter by one; when these references are removed, the reference counter eventually drops to zero and the object itself can be destroyed.

When programming in pure Perl, the garbage collection scarcely becomes visible[1]. When both a C program as well as a Perl script running therein need access to the Perl data structures, this can become a problem, however: A data structure that is created in a Perl script and passed back to the calling C program must explicitly be freed by manually decreasing its reference counter. Another – in many cases sufficient – option is to (implicitly or explicitly, depending on context) mark the object as "mortal" or temporary and wrap the concerned code section in "magic" macros (`ENTER`/`SAVETMPS`, `FREETMPS`/`LEAVE`) which will take care of reference counters. When different Perl objects with different life times transit larger portions of code, this automatic handling mostly fails, however, as some memory portions need to be freed while others may not.

Put briefly, it is necessary to take care of the differences of dynamic and static memory management at the interface of C and Perl.

9.4. Perl module

The Perl module consists of three different entities:

- The module itself. Handles module initialization, interpreter instantiation and execution of Perl functions
- The Perl extension. Provides Perl functions to access OpenSER internals

[1] Reference counting becomes a problem when there are cyclic references. This can also be an issue in Perl.

- The Perl library. Features a set of convenience functions that will commonly be used in the context of a SIP server

These parts will now be discussed in more detail. The difficulties and bugs encountered during their implementation will be described.

9.4.1. The module itself

The OpenSER Perl module is a rather thin layer. Its parts can be subdivided into the following categories:

- Declaration of the interface. What functions are exported by the module? What arguments are available?
- Module initialization, construction and destruction. Allocating memory, Perl interpreter initialization, import of "`stateless`" module API.
- Perl function call handling. Provides a set of four alternative functions to be used from within the OpenSER configuration.

While the first two parts are implemented in the module's `perl.c`, the latter is implemented in `perlfunc.c` (and their respective header files). `perl.c` follows the same pattern as all other modules. This is the interface definition of the Perl module:

```
/*
 * Exported functions
 */
static cmd_export_t cmds[] = {
	{ "perl_exec_simple", perl_exec_simple1, 1, NULL, REQUEST_ROUTE
						| FAILURE_ROUTE
						| ONREPLY_ROUTE
						| BRANCH_ROUTE },
	{ "perl_exec_simple", perl_exec_simple2, 2, NULL, REQUEST_ROUTE
						| FAILURE_ROUTE                          10
						| ONREPLY_ROUTE
						| BRANCH_ROUTE },
	{ "perl_exec", perl_exec1, 1, NULL, REQUEST_ROUTE | FAILURE_ROUTE
						| ONREPLY_ROUTE | BRANCH_ROUTE },
	{ "perl_exec", perl_exec2, 2, NULL, REQUEST_ROUTE | FAILURE_ROUTE
						| ONREPLY_ROUTE | BRANCH_ROUTE },
	{ 0, 0, 0, 0, 0 }
```

```
};

/*                                                                  20
 * Exported parameters
 */
static param_export_t params[] = {
        {"filename", STR_PARAM, &filename},
        {"modpath", STR_PARAM, &modpath},
        { 0, 0, 0 }
};

[...]
                                                                    30
/*
 * Module interface
 */
struct module_exports exports = {
        "perl",
        RTLD_NOW | RTLD_GLOBAL,
        cmds,       /* Exported functions */
        params,     /* Exported parameters */
        0,          /* exported statistics */
        mi_cmds,    /* exported MI functions */                     40
        0,          /* exported pseudo-variables */
        mod_init,   /* module initialization function */
        0,          /* response function */
        destroy,    /* destroy function */
        child_init  /* child initialization function */
};
```

The second line of the module interface shows the flags to be used while initializing the module with the standard library "`dlopen`" call and reflects one of the major problems that occurred during the implementation.

9.4.1.1. The "dlopen problem"

On POSIX style systems such as Unix, the function `dlopen` is used to dynamically load shared libraries during runtime. OpenSER uses this call to load its modules at runtime. In addition to the file name of the library to be loaded, `dlopen` expects a bitmap of possible flags. On most common Unix variants, the `dlopen` implementation provides at least flags for specification on the load time (lazy vs. instantly). However, a

large problem was encountered with the flags defining the symbol resolution mechanisms (`RTLD_GLOBAL` vs. `RTLD_LOCAL`).

Perl itself uses the `dlopen` call to load Perl extensions necessary for its modules. These extensions are a very fundamental technology in the Perl standard library as well as in third party modules and in the implemented OpenSER module. The Perl `dlopen` call uses the `RTLD_GLOBAL` flag, while OpenSER originally loaded the modules with the `RTLD_LOCAL` flag. This led to rather obscure segmentation faults when Perl tried to load libraries itself, probably due to unreachable symbols. Because of that, a patch for OpenSER was proposed (and accepted, after a number of modifications) that lets modules set their "desired" dlopen flags: The module is loaded with the default flags, and its export structure is evaluated. When the flags contained therein differ from the default set, the module is unloaded again and reopened with the correct flags.

9.4.1.2. The "reload problem"

OpenSER does not have the feature of modifying its behavior during runtime. To reflect changes in the configuration file, it is necessary to restart the server process. In many cases, this is creates problems, as it leads to unwanted downtimes of the system (although restarting is quick, and infrequent restarts will be tolerable). It would therefore be helpful if the Perl script running inside the interpreter could be reloaded without restarting the whole server process.

OpenSER, similar to other server processes such as the Apache web server, forks after starting. Several more or less identical processes run simultaneously, processing requests in a round robin scheduling system. Because of that, also the instantiated Perl interpreter is cloned. Launching a reload of the Perl script through one of the available interfaces leads to a reloading of the script in the instance that "accidentally" handles the request – the other instances will remain untouched.

The proposed way to implement an administrative function in a module is the management interface, MI, that has undergone extensive modifications during the time this thesis was written. The old FIFO implementation was replaced by a much more general interface which may be accessed currently by either a new FIFO interface, or by an XMLRPC interface. A reload function was registered with this interface, leading to the problems described above.

A proposal to extend the MI with a functionality to execute certain functions in *all* instances of a module was submitted. A complying feature will eventually be integrated.

9.4.1.3. The "global variable problem"

Although not directly a property of the Perl module itself, the following problem has the same background as the reload problem and is thus described here.

As there are multiple instances of the Perl module running concurrently, global variables set in one instance are not "seen" by the remaining instances. On the one hand, this is an advantage, as it is unnecessary to integrate locking mechanisms for the access of these variables. Side effects are restricted to the same instance. On the other hand, some types of functionality could rely on a shared memory. One might imagine a simple "call counter": each SIP message is counted in a global variable in Perl. The simple implementation

```
my $globalcounter = 0;
my FILE;
sub count {
  $globalcounter++;
  print FILE "I counted $globalcounter messages. \n";
}
```

will not work, as the instances do not share their global variables.

This shortcoming can easily be circumvented by a number of different methods. An obvious solution would be a database back-end to store global information; in the case above, an SQL database could store information on processed messages (or simply count them). An even simpler method would be the usage of a Perl module such as `IPC::Shareable` that allows to use POSIX-style shared memory to share conventional Perl variables. Instead of the `$globalcounter` above, one could use a statement like

```
tie $globalcounter, IPC::Shareable,
   { key => "foo1", create => 1, destroy => 1 }
```

to bind the variable to a shared memory segment.

A sample script that demonstrates both cases is included with the Perl module.

9.4.2. The Perl extension

Returning to the topic of End User Development, the central part of the Perl module is the Perl interface to SIP messages. OpenSER provides a large number of functions in the core and in its modules to evaluate and modify different parts of messages. These functions should be available in Perl, too.

During the call of Perl functions, the pointer to SIP messages is transformed into a Perl reference value that can be accessed just as a conventional Perl variable:

```
struct sip_msg *_msg;
[...]
m = sv_newmortal();
sv_setref_pv(m, "OpenSER::Message", (void *)_msg);
SvREADONLY_on(SvRV(m));
[...]
XPUSHs(m);  /* Push the message reference on the stack */
[...]
PUTBACK;    /* make local stack pointer global */
                                                              10
call_pv(fnc, G_EVAL|G_SCALAR); /* call the function */
```

In Perl, called functions then can fetch a reference to the SIP message from the stack. This reference, belonging to the Perl class `OpenSER::Message`, allows functions to access methods of this class. Similarly, a SIP URI structure is reflected by the `OpenSER::URI` class.

The implementation of the `OpenSER::Message` as well as the `OpenSER::URI` and `OpenSER::AVP` classes is located in the Perl extension. As described above, an extension is written in the meta language XS that is later translated into a C file. The XS file name is `openserxs.xs`.

The accessor functions providing interesting functionality for Perl scripts can be subdivided into core and module functions. A unified access method to the module functions was implemented (leading to a major problem, see below), but the core functions had to be reflected individually.

For every "interesting" core function, an equivalent in the Perl extension had to be developed. As these wrappers only need to validate the arguments and return the core functions' results, most of them are extremely simple. One example of these functions is the `getBody` function:

```
SV *
getBody(self)
    SV *self
  PREINIT:
  PREINIT:
    struct sip_msg *msg = sv2msg(self);
```

```
INIT:
CODE:
    if (!msg) {
        LOG(L_ERR, "perl: Invalid message reference\n");          10
        ST(0) = &PL_sv_undef;
    } else {
        parse_headers(msg, ~0, 0);
        ST(0) = sv_2mortal(newSVpv(get_body(msg), 0));
    }
```

This excerpt demonstrates:

- A possible option for returning values is putting them on the stack. In this case, this is done by using the stack access macro `ST`.[2]
- "Non-values", in C programs usually identified by NULL pointers, zero integer values or dedicated constants, are described by the Perl scalar value `undef`. On the C side, `undef` may be referenced by the `PL_sv_undef` variable.
- Perl's SIP message reference is transformed back to the pure C pointer in a small inline function `sv2msg`. This function checks the validity of the reference.

Module functions are handled somewhat differently. There is a single function `moduleFunction` that tries to find a function according to the passed function name and parameters. A module function that would be written as `foo(“param1”, “param2”)` in the configuration file can be called as `$msg->moduleFunction(“foo”, “param1”, “param2”);` in Perl. The Perl autoloader mechanism implemented in `Message.pm` makes it possible to directly call the function: `$msg->foo(“param1”, “param2”);`

Calling module functions is done by requesting and evaluating a function export structure for that function from the OpenSER core (similarly as it is done in the configuration file parsing). Calls to these functions are preceded by parameter preprocessing.

9.4.2.1. The "constants problem"

In multiple contexts of the OpenSER code, constants are used for all kinds of information. SIP messages can, for example, be either a request or a reply message, so the two constants `SIP_REQUEST` and `SIP_REPLY` are created; a request message can belong to

[2] Another option is using the implicitly defined `RETVAL` variable. This technique is used e.g. in the `getType()` function

exactly one of the message classes ("methods" such as `INVITE`, `BYE`, `REGISTER`, `MESSAGE`), so appropriate constants are defined. In OpenSER, constants are implemented by the preprocessor `#define` macro, or by `enums`.

Unfortunately, it is crucial also to have many of these symbolic names on the Perl side. As the C language does not integrate any introspection features, there is no feature like "give me all names and values in the enum". Due to this reason, the only simple possibility to reflect the OpenSER constants was a manual transformation into Perl constants.

Perl also has a variety of options on how constants can be set. In this context, the "`constant`" Perl module was used. Some hacks help the user to easily get access to the constants defined, e.g. by automatically exporting them. All constants were defined in a newly created package, `OpenSER::Constants`.

The Perl embedding environment features a tool called "h2xs". This tool is Perl's answer to the problem described here, which is anything but unique in this project. The output of h2xs is a bloated Perl extension (.xs file), featuring transformations for `enums` as well as for `#defines`. h2xs creates a large overhead for the access of constants. The output is meant to be a template for further development rather than to be a self-contained system that can be reconstructed on a regular basis to reflect modifications of the base system's header files. Due to these weaknesses, h2xs was not used in OpenSER's Perl module.

[31] describes another way to transform C constants to Perl equivalents by (still manually) creating constant functions for every C constant. Again, this procedure uses an XS and slows down constant access by featuring constant functions instead of constant variables. Considering the downsides, this technique was also turned down.

9.4.2.2. The "fixup problem"

During the discussion of the OpenSER code, the preparation of function arguments by fixup functions (see section 3.3.3) was outlined. As described, the fixup functions have full control over the data types they return, as they are implemented in the same context as the functions themselves. During the initial implementation, the far-reaching consequences of this fact had been ignored.

A module function such as `xlog()`[3] allocates considerable amount of memory during its fixup. The Perl module does not have any possibility of freeing this memory again.

[3]xlog is a logging function that understands pseudo variables. Other functions are considerably worse in respect to their memory usage...

Simply calling the C function "`free`"[4] is not an option, as the underlying variable *may* be something that cannot be freed at all (particularly an integer, or a pointer to a nested structure).

First tests with the module functions did not show any abnormal behavior – that was, not until the server was confronted with several thousands of consecutive calls, depending on the setup[5]. In every call, a chunk of memory was allocated to contain the fixup'd data set. This memory was never freed again.

There are several conceivable options of solving this problem; what they have in common is the fact that the module that does the initial fixup will have to revert it again. The Perl module does not have any knowledge about the data types of variables returned by the fixup function and thus cannot free their memory, unless every available module function is reflected by a dedicated cleanup code snippet in the Perl module (which would not be a good, clean architecture).

The following line is taken from the module's current moduleFunction implementation:

```
*retval = exp_func_struct->fixup(&(act->elem[2].u.data), 1);
```

The best option would be to pass an additional parameter to the fixup function that lets it return a pointer to a different function that frees the allocated space and obliterates the process of the fixup. In many cases, this will be very easy: for flat structures, it will be sufficient to return "`pkg_free`"; for integers, "`NULL`" would be sufficient. As the implementation of this feature would be very invasive on literally every other module, this task was postponed.

The module export structures are declared and implemented in `sr_module.c/.h` , including a typedef for fixup functions:

```
typedef int (*fixup_function)(void** param, int param_no);
```

By adding a third parameter to the fixup function and an additional destruction function type, the described problem could be solved:

```
typedef int (*unfixup_function)(void* param);
typedef int (*fixup_function)(void** param,
        unfixup_function *unfix, int param_no);
```

While calling the fixup function, the destruction function would be returned and could later be used to remove the created structures. The example described above could then be extended:

[4] In any case, OpenSER's internal memory management function, `pkg_free`, would have to be used

[5] The exact number depends on the size of OpenSER's internal memory and the module function that is called from inside Perl. Problems occurred between 5'000 and 22'000 calls.

```
int (*unfix)(void *param);
*retval = exp_func_struct->fixup(&(act->elem[2].u.data), &unfix, 1);
[...]
unfix(act->elem[2].u.data);
```

The next OpenSER release 1.3 is expected to contain a mechanism such as the one outlined here. Complete module function calling will then be available in the Perl module.

9.4.3. The Perl library

The Perl environment implemented for the OpenSER module is rather small. The following Perl packages are contained in the tree:

- OpenSER
- OpenSER::Message
- OpenSER::URI
- OpenSER::AVP
- OpenSER::Constants
- OpenSER::LDAPUtils::LDAPConf
- OpenSER::LDAPUtils::LDAPConnection
- OpenSER::Utils::PhoneNumbers
- OpenSER::Utils::Debug

While the first four packages provide functionality for the interface of OpenSER and the Perl module, the `OpenSER::LDAPUtils::*` and `OpenSER::Utils::*` packages provide basic functionality that may be interesting in this context. `OpenSER::LDAPUtils::*` and `OpenSER::Utils::PhoneNumbers` were not developed by this thesis' author, but were kindly provided by Collax GmbH. After minor modifications, they created a large value for this project.

OpenSER The root of the OpenSER Perl module namespace has quite a simple job: it "bootstraps" the Perl extension (making the interpreter load the dynamic library), and defines error handling functionality that logs Perl error messages and warnings through OpenSER's logging functionality (instead of printing them to the standard error device, which would not be sensible in this context). The only user accessible function is the log method, which is implemented in the Perl extension.

OpenSER::Message and OpenSER::URI These two packages provide access to OpenSER's central internal structures, i.e. SIP messages and URIs contained therein. `Message.pm` adds an autoloading functionality to the package that lets the user access `$m->foo()` as an abbreviation for `$m->moduleFunction("foo")`. Additionally, an (empty) `DESTROY` function is defined to prevent Perl from trying to autoload this function. The effective functionality of this module is provided in the Perl extension. A list of implemented methods is available in the module documentation.

`OpenSER::URI` is solely defined in the Perl extension; a `.pm` file does not exist. Its methods simply return a component of the examined URI.

OpenSER::AVP While the AVP implementation in OpenSER is not a distinct part of the code, it is reflected as a separate Perl package. Its three functions, `add()`, `get()` and `destroy()`, directly interface the OpenSER core functions `add_avp()`, `get_avp_val()` and `destroy_avp()`. Numerical as well as symbolical AVPs are supported.

The package's functions take and return scalars (strings or integers). The package thus reflects a namespace rather than an object class. It would have been possible to create a dedicated `OpenSER::AVP` class with value and name attributes; as AVPs reflect rather atomic entities (namely scalar values), this option was rejected.

OpenSER::LDAPUtils::* Perl's LDAP module `Net::LDAP` provides sophisticated methods for LDAP access. Its routines are however not simple to use, so a simple wrapper was added that provides more specialized implementations for simple requests.

`OpenSER::LDAPUtils::LDAPConf` is able to evaluate system configurations of the open source LDAP implementation openLDAP; `LDAPConnection`'s single remarkable method is its `search` function that can use `LDAPConf`'s configuration to execute a search on an LDAP server.

OpenSER::Utils::PhoneNumbers This module defines functions for conversion of telephone numbers to their canonical form and vice versa. Within the University of

Freiburg, the canonical form of the number “1234” would be “+49-761-203-1234” – the function `canonicalForm` does this translation[6]. In the same context, the number to dial when contacting the canonical number “+49-761-203-4321” from the Collax GmbH developing department would be “0-203-4321”. This transformation is provided by the `dialNumber` method.

The dial context, consisting of a number of prefixes for international, national, local area and local calls, is set by the class constructor.

OpenSER::Debug During the development of the modules, the life cycles of Perl objects had to be evaluated. This led to the creation of a Debug class that contains a `DESTROY` handler logging the end of an object life time (as soon as it inherits from this class). Despite minor differences, the `DESTROY` handler can be compared to a destructor in languages like C++ or Java.

Eventually, this module could contain additional functionality for the context of debugging.

9.4.4. Documentation

The last duty before the code release was writing a proper documentation. OpenSER modules are documented with DocBook documents that may be transformed to text, PDF, html and other formats. The regular OpenSER documentation features a common style that is also used the Perl and VDB module documentation.

Perl modules usually feature a built-in documentation, the so-called “POD” (Plain Old Documentation). This markup language provides some simple commands that can be used to embed documentation into the code. The POD segments of a Perl module or program can be extracted and transformed into numerous different formats.

In case of the Perl module, a large part of the DocBook documentation is auto-generated from the POD segments of `openserxs.xs` with the pod2docbook [51] tool. Every function in `openserxs.xs` is preceded by a description that is later transformed to user documentation.

The resulting module documentation is available on the OpenSER website and as a README file. Additionally, users can build their own PDF, HTML or text documentation.

[6] Dashes added for easier readability

9.5. perlvdb module

OpenSER's Perl Virtual Database is implemented in an analog way to other available database modules: the module exports functions with special function names, describing their functionality. The implemented functions are:

```
db_con_t* perlvdb_db_init(const char* _url);
void perlvdb_db_close(db_con_t* h);

int perlvdb_use_table(db_con_t* h, const char* t);

int perlvdb_db_insert(db_con_t* h, db_key_t* k, db_val_t* v, int n);
int perlvdb_db_replace(db_con_t* h, db_key_t* k, db_val_t* v, int n);
int perlvdb_db_delete(db_con_t* h, db_key_t* k, db_op_t* o, db_val_t* v, int n);
int perlvdb_db_update(db_con_t* h, db_key_t* k, db_op_t* o, db_val_t* v,
            db_key_t* uk, db_val_t* uv, int n, int un);                          10

int perlvdb_db_query(db_con_t* h, db_key_t* k, db_op_t* op, db_val_t* v,
                db_key_t* c, int n, int nc,
                db_key_t o, db_res_t** r);

int perlvdb_db_free_result(db_con_t* _h, db_res_t* _r);
```

This interface reflects the functionality defined during the design phase.

Database modules may additionally export a "raw query" function that allows modules to directly access databases with SQL statements, and a "fetch result" function that fetches a number of rows from the current result set. While loading a database module, the DB API checks for existing functions and sets flags for existing and non-existing functionalities in a capability matrix. Thus, database modules can provide an arbitrary subset of database functions. As the VDB module targets towards accessing non-SQL-databases, both the raw query and fetch result functions would not be appropriate in the context and thus were omitted.

In contrast to the Perl module, where only a reference to the data structures is passed between the C and the Perl layer, the data handled by the VDB module will usually be fully consumed by the Perl functions, or will be fully created respectively. Thus, the full data structures are transformed during the execution of a database request.

The implementation of the VDB module is divided into the following parts:

- Module interface

- C side implementations of database access functions
- Data transformation
- Perl class structure
- Adapters for database client modules

Similar to the last section, these parts will be discussed in more detail.

9.5.1. Module interface

Although database provider modules differ in some ways from regular modules, the implementation followed now well-known patterns. While regular OpenSER modules flag exported functions to be used in respective routing blocks, database modules do not set any export flags. Dedicated function export names mark the respective functions as DB API implementations.

The `perlvdb` module uses the perl interpreter instance that is provided by the Perl module. During module initialization, the VDB module checks whether the Perl module is loaded. Coincidentally, a boolean `module_loaded` function was added in OpenSER during the development phase of this thesis.

9.5.2. Database access functions

At the time of implementing the VDB module, the technique to call Perl functions from within C was a well-known task. All functions in `perlvdbfunc.c` are restricted to more or less three simple steps: Convert input data to Perl objects; call the correct Perl method; convert output data in Perl objects to C data structures. The conversion of data will be discussed in the next section. Calling Perl object methods differs only slightly from calling conventional functions.

An important point during the method calling is to take account of Perl's memory management. Objects returned by functions or methods will have to survive for a while on the C side, so relying on the automatic handling of reference counters was error prone and led to different kinds of difficulties. A result set on Perl side has to be evaluated by a number of additional Perl function calls; because of this, a number of different life times for different objects occurred.

The VDB module uses its own wrapper to the Perl API function `call_method`. This function uses the wrapper macros described in section 9.3.3, but explicitly increases the

reference counter of the returned object. The calling function will have to decrease this counter by explicitly calling `SvREFCNT_dec()` to obliterate the data structure.

9.5.3. Data transformation

OpenSER's database API provides only limited abstraction of the data structures handled by the database modules. There are only four partly abstracted data types: keys, values, operations and result sets. Keys and operations are nothing but typedefs for strings; values are structs containing a type and the data itself. Result sets consist of a set of column definitions, plus an array of rows; the latter, in turn, are arrays of values.

The VDB module partly resembles this structure, but in other parts tries to add a bit of abstraction by adding the class structure described in the design chapter (section 8.5.1).

The data transformation functions in the VDB module are responsible for the transformation of the C structures into sensible Perl objects.

The data transformation functions are tightly connected to the available Perl classes. In the header files, their class names are defined as constants.

These conversion functions are available:

```
AV *pairs2perlarray(db_key_t* keys, db_val_t* vals, int n);
AV *conds2perlarray(db_key_t* keys, db_op_t* ops, db_val_t* vals, int n);
AV *keys2perlarray(db_key_t* keys, int n);

SV *val2perlval(db_val_t* val);
SV *pair2perlpair(db_key_t key, db_val_t* val);
SV *cond2perlcond(db_key_t key, db_op_t op, db_val_t* val);

int perlresult2dbres(SV *perlres, db_res_t **r);
```

Although the implementation of these functions was rather simple – iterating arrays, calling the Perl methods to create equivalent objects, and inserting these in Perl arrays –, the interference with Perl's memory management became an issue during the debugging.

The following types of variables occur:

- Arrays of keys (requested keys in query)
- Arrays of key/value pairs (new data in insert/update/replace operations)
- Arrays of request conditions (key/operation/value data in query and update)

- Result sets (returned by queries)

When a Perl array is filled with the `av_push()` function, the elements' reference counters are not increased. Thus, it is sufficient to flush the array with `av_undef()`; the elements contained in the array will then automatically be removed.

The result set is a nested structure that is not as trivial to handle. Error handling makes it difficult to remove sub-structures at the correct position. Due to this, the development of the `perlresult2dbres()` function was a little more elaborate and was largely a trial and error process[7].

9.5.4. Perl classes and adapters

The VDB class structure falls into two parts: classes for database objects such as values or keys, and adapters for database accessing modules. Additionally, a number of sample implementations for certain use cases were constructed.

It became apparent only at the time of implementation that it would be rather impossible to create a sensible all-embracing virtual database. At this point, the design phase was re-entered and the adapter concept was developed (see sections 8.3.2 and 8.5.2).

While the object classes (`OpenSER::VDB::[Column,Pair,ReqCond,Result,Value]`) are quite simple containers for a few variables and their accessor methods (getters and setters), the adapters provide a little more sophisticated functions to convert operations on a relational schema to imperative function calling.

As the VDB module is integrated with the Perl module and uses its interpreter instance, the classes were added to the Perl module library path, instead of adding another one. The sample VTab implementations for the adapters, however, are located in the VDB `doc/samples` directory. Although it is expected that users can use some of these scripts without major modifications, minor adaptions to local circumstances will be necessary.

[7] The `OpenSER::Utils::Debug` class described above was largely created to examine the life times of the VDB objects. The evaluation of this debugging led to the correct positioning of reference incrementation and decrementation operations.

10. Testing

Large parts of the implementation phase were accompanied by functionality tests; an independent testing and debugging phase was performed after the completion of the modules, however. The testing techniques, tools and results will be described in this chapter.

10.1. Debugging OpenSER

Server processes and other dynamic software are often difficult to debug. While conventional, algorithm-driven or regular user software may be started with identical input data fairly easily, this is a challenge for Internet server software; printing debug output to the screen is also not simply possible in this context, since server processes usually do not have access to standard output devices.

Additionally, OpenSER uses the POSIX fork mechanism to spawn children that can separately answer incoming requests. Watching one specific child is not sufficient.

It is an even bigger problem that server processes are much more difficult to run in a conventional debugger. In a few cases, the examination of Unix core dumps with the GNU Debugger "gdb" was necessary, as in many cases conventional debugging with breakpoints breaks the sensitive timing constraints inherent to a SIP server.

Fortunately, two of these problems can be circumvented through OpenSER's debugging flags: forking can be turned off, so that only a main process has to be observed. Printing to the standard output/error devices also results in text to appear on the screen.

The most powerful method in this context, however, is the usage of OpenSER's logging facility. Depending on the log level that is passed as a parameter to the log function and the system log level configured in the OpenSER's configuration file, a different level of verbosity can be reached. With a sensible distribution of debugging `log()` calls, the system's reaction to externally triggered events can be examined.

10.2. Testing environment

Large parts of the programmed functionality was tested concurrently with the implementation process. Events were triggered manually via an attached hardware SIP telephone, a Snom 360. The receiving user agent usually was an instance of the Linux softphone "twinkle" [14]. An Asterisk server in the same network provided a bridge to the ISDN network.

Further tests were done using the SIP test programs sipsak (SIP Swiss Army Knife, [43]) and SIPp [25].

10.2.1. sipsak

sipsak was developed at the Fraunhofer FOKUS. This tool is able to send multiple types of SIP messages without the need to implement a full-featured SIP stack. Different message methods (`REGISTER`, `INVITE`, `OPTIONS`...) are supported; the central addresses and URIs (Recipient, To, From, Via...) in the message can be passed via the command line.

One of sipsak's features is to register devices with a user location server. This is a useful feature when testing SIP servers in conjunction with SIPp, which does not feature registration out of the box.

sipsak was developed with SER in mind. Other systems have not been tested by the program's maintainers; however, sipsak is also used by the OpenSER developer team for evaluation.

10.2.2. SIPp

SIPp was primarily developed to run performance tests against SIP servers. Basic scenarios are included to initiate rather simple `INVITE`/`BYE` calls, but almost arbitrarily complex call flows can be configured by using user-defined scenarios. These are created by XML files with a specific schema which defines messages and responses for certain events.

SIPp has been used to evaluate the performance of the perl and perlvdb modules. The program's value has also been demonstrated when a user found a memory leak in the code during SIPp testing.

A detailed introduction to both sipsak as well as SIPp is available in [16].

10.2.3. Computer hardware

The computer hardware for the testing environment consisted of a single computer that contained the binaries of OpenSER, the softphone twinkle, and the test programs sipsak and SIPp. Although SIPp consumes a considerable amount of computation power and thus could have reduced OpenSER's performance while stress testing a system, the advantage of not having to rely on a public Ethernet was of greater importance.

The computer's basic data were:

- AMD Athlon64 3700+ (2200 MHz)
- 1 GB RAM
- 80 GB Hitachi SATA hard disk (7200min^{-1})
- OpenSUSE 10.1 (Linux kernel 2.6.16.27)

10.2.4. Benchmark module

For the stress testing and benchmarking, a third OpenSER module was written, called "`benchmark`". It exports two functions, `start_timer` and `log_timer`, for user access from the configuration file. Semantically identical functions `bm_start` and `bm_log` are exported through an API.

By calling these functions before and after a block of execution context, the execution duration of this block is logged. A `granularity` variable allows the user to print log messages for every n'th call only. Average values are provided in the log. The underlying `gettimeofday` system call returns timing information with microsecond accuracy.

A sample log line looks like this:

```
benchmark: reporting msgs/total/min/max/avg - since last report: \
100/4956/38/113/49.560000 | global: 2000/105597/38/521/52.798500
```

A very similar concept was realized by an OpenSER core developer, unfortunately three weeks too late for this work.

10.3. Test cases

The reason for software testing is to ensure an appropriate level of quality. Although the term quality has an implicit meaning, it is not easy to define explicitly in the context of software systems.

Software engineering generally defines these primary quality characteristics for the programmer perspective:

- Correctness
- Conformance
- Scalability/Efficiency
- Fault tolerance
- Maintainability

Other groups of people involved (e.g. users, clients, ...) have other perspectives and quality needs.

In the context of an OpenSER module, testing is mainly useful for evaluating correctness, conformance, efficiency, and fault tolerance.

The following topics can now be examined:

- Does the system provide the expected functionality? (→ correctness, conformance)
- What impact do the modules have on OpenSER's performance? (→ efficiency)
- How does the system cope with a large number of messages? (→ scalability)
- How does the system react on invalid messages? (→ fault tolerance, robustness)
- How does the system react on invalid user-developed Perl scripts? (ditto)

Correctness tests were already stressed during the development phase, but repeated later more thoroughly. As soon as the code was regarded fairly stable, it was released to the public and delivered with OpenSER's pre-1.2.0 branch. Numerous users thus were included in a process similar to a beta test.

User feedback led to the detection of a central problem – the "fixup problem" described in section 9.4.2.2.

10.4. Testing procedure and results

As described above, the procedures of testing the conformance and correctness were not formalized. Test functions for the module core functionalities were implemented and tested, mainly through manual dialing of a telephone set.

A list of tasks was frequently updated, adding new topics, and marking others as closed. By these means, the development was focused on the intended functionality. Comparing the system's state with the initial goals of the design, the system conformed to the expected behavior. The fix for the "fixup problem" described above has not yet been implemented, but proposed.

The system conformance with the initial requirements will be discussed in the next chapter.

10.4.1. Stress testing and performance evaluation

A central part of the testing phase was stress testing of the system. The primary expected result was a view over the modules' performance impacts.

Stress testing was done with the SIPp program described above. The "fixup problem" as well as other memory leaks were found – and partially fixed – in this phase. Besides that, no further issues were found during the stress tests. Both the Perl module as well as the VDB module survived 100'000 calls without any problem.

When confronting OpenSER with more than 250 calls per second, processing errors were encountered: Firstly, the UDP processing of the underlying Linux system led to retransmissions of messages; if OpenSER is configured not to use its forking mechanism, these errors went away. Secondly, OpenSER's internal memory management became an issue with more than 350 calls per second. These numbers were independent of calling Perl functions from the routing configuration; Perl calls did not have any impact on the processing.

OpenSER as well as the underlying Linux system have a number of tuning possibilities. As no impact by the perl/VDB modules was measurable, these tuning options were not deeply investigated.

Performance evaluation was done by inserting `gettimeofday` system calls in the module and through the separately implemented benchmark module. The timing was averaged over 10'000 messages with 30 calls per second. These numbers result from tests in the Perl module on the machine on which the modules were developed:

Functionality	Time in μs
Empty Perl function	24.123800
Simple message evaluation (callback)	29.435900
Additional simple regular expression	62.102549
Callback to alias_db	354.917400
LDAP request	6699.631400

The jitter during the 10'000 messages was rather low; output was created for every 100th message.

The simple message evaluation included a callback to an OpenSER core function to get the recipient URI; the "simple regular expression" was:

```
my $ruri = $msg->getRURI();

my $oldhost = "172.16.1.200";
my $newhost = "bilbobox.collax.com";

$ruri =~ s/sip:([a-zA-Z]+)@($oldhost):([0-9]+)/sip:$1@$newhost:$3/;
```

The LDAP request was done with a remote LDAP server (ping time $\sim$ 0.2 ms); although the time for the request is long in comparison with the other timings, less than 7 ms is still a sufficiently short period.

The following run time estimations can be derived from these figures:

- Neither calling Perl nor callbacks to the OpenSER core (transition of "language boundaries") are expensive
- Perl processing (code execution) is not expensive
- Querying external sources – such as an SQL database, or the LDAP directory – are one to two orders of magnitude more expensive. This is not related to Perl.

Taken together, the results show that the Perl module will not have a negative performance impact on the running system.

For the VDB module, no full benchmarking tests were done due to two reasons: Firstly, the importance of the VDB module is expected to be significantly lower as the Perl module's, as explained in chapter 11. Secondly, the expected results do not differ too much from the results presented here.

During the evaluation of the Perl and VDB modules, benchmarks were compared. See section 11.3.1 on page 130 for the discussion of the results.

10.4.2. Invalid messages

One of OpenSER's core key features is its message parser. This subsystem is responsible to transform the text-structured SIP messages to a format that is easily interpretable by the modules.

As the Perl module relies on the parsing in the core, the module itself does not separately deal with invalid messages. Due to this, dedicated tests were not driven.

10.4.3. Invalid Perl code

A central aspect of the Perl module implementation was its behavior regarding invalid user scripts. The term "software quality", especially concerning correctness, comes into play here:

- User scripts can be syntactically wrong. During module initialization, the parser will not accept the code. The server will not start up.
- User scripts can generate uncaught runtime errors. In Perl, this leads to a "die". By calling Perl's API function `call_pv` with the `G_EVAL` flag set, the error is handled correctly. OpenSER does not crash; instead, the library's die-handler creates error logging messages. Internal script processing cannot be assessed here, however: Invalid internal states may be reachable if the developer does not explicitly take care of them.
- User scripts can hang or run forever. The "halting problem" prevents the system from detecting whether a script can reach hanging conditions. A "watchdog" could theoretically stop script processing after a specified time; this was not implemented.

Different "buggy scripts" were tested in OpenSER, both intentionally and unintentionally. Perl correctly handled syntactical errors on startup. Sensible handling of internal `dies` – explicitly via calling the `die()` function, or implicitly by creating wanted runtime errors – were handled in the expected ways.

10.4.4. Regression tests and coverage analysis

As described, tests were driven concurrently with the implementation. Regression tests were done after the implementation phase to re-validate the functionality of the Perl module. For the VDB module, testing and implementation were more tightly integrated, but concluding tests were also done.

During the regression tests, it became apparent that the `pseudoVariable()` extension function was incorrectly using the `malloc`/`free` and `pkg_malloc`/`pkg_free` functions: While memory was allocated through OpenSER's internal memory management (`pkg_malloc`), it was freed with the system call `free()`. This led to a segmentation fault when using certain variables, as OpenSER tried to re-use the memory that was ultimately freed by the module. This bug was interesting for two reasons: Firstly, it only cropped up with a subset of the available pseudo variables; secondly, it was incidentally discovered at the same time by a different person. At the time the problem of that user was understood, the fix was already in the CVS repository.

A test script with a single function was written that calls a large part of the API functions defined in the Perl extension and in the library. The GNU program/library "`gcov`" and the Perl module `Devel::Cover` provide a coverage analysis of these tests. It was attempted to reach a reasonably high coverage of the central components of the modules.

Part V.

Discussion

11. Discussion and Conclusion

Shortly after the Perl module had been published, people in different positions started using it. One of the users was working for a major U.S. American cable network that also provided IP and VoIP services. The company is planning to use the Perl module in their production environment.

Although this thesis' sponsor, Collax GmbH, is not yet involved in the implementation of a VoIP product, a white paper on possible options has been written. A crucial point in that paper was the integration of a VoIP environment with an LDAP service which is a key component in Collax products. The Perl module provides features for this integration.

The positive practical implications of the Perl module can be seen in these two use cases. It remains to be shown that the original tasks of this thesis are solved. Additionally, the Perl and VDB modules will be compared with each other and with other approaches to the EUD and LDAP topics.

11.1. Revisiting Use Cases

In section 5.2, a set of data categories in use in a VoIP system was defined, derived from a number of use cases. Later, these data were related to entities in the OpenSER server. At this point, these topics will be revisited to check the system's conformance with the initial requirements.

The following table contains the same categories as the one developed in the requirements analysis in section 5.3.

Data	Solution
Identity information	As already mentioned, this term is not defined well. Caller IDs can be modified through the modification of SIP messages in various ways, e.g. by calling the module functions provided by the `textops` module. The location database and the `usrloc` module are central parts of OpenSER. There does not seem to be a reason to substitute the integrated databases for a different technology. This is especially true as the `usrloc` module features internal caching; dynamic data will not instantly become active. While it would be possible to use the VDB module as a back-end for `usrloc`, scenarios where this is reasonable will be rare. See below for discussions of the subscriber, user and alias databases.
Voice box	The newly available data back-ends can provide references to voice boxes. As OpenSER does not handle voice mail itself, the exact behavior heavily depends on the given infrastructure.
Address book(s)	Mapping of user identities with their addresses is possible through the `alias_db` and `speeddial` modules in connection with the VDB module. Directly accessing a Perl function through the Perl module is even simpler. The latter module also allows for arbitrary modification of the SIP message, including the Caller ID.
User database	The subscriber database may either be realized through current authentication modules backed by the VDB module, or by arbitrary new functionality in the Perl module. Using a dedicated Triple-A system such as RADIUS is the preferred idea; see the discussion in the next section.
Aliases database	The VDB module includes an adapter class for the `alias_db` module. Directly modifying the SIP message through the Perl module, however, is a lot simpler.

Data	Solution
Conference room database	As discussed, OpenSER only handles marginal properties of a conference room system. Databases that include information about conference rooms can be accessed through Perl functions. Arbitrary permission management for OpenSER is well possible by features of the Perl module.
User groups	Although the management of user groups through RADIUS is the preferred setup (see below), the group module could be used together with the VDB. An adapter has not yet been implemented.
Meta databases	As the VDB and Perl modules provide a separation from underlying database schemata and technology, meta data in databases accessed by OpenSER do not interfere with this software.
Registration/location information	The `usrloc` module could be used with the VDB. This is generally not a good idea. A Perl-only reimplementation is possible.
Status information	The presence information within OpenSER's `pua` and `presence` modules can be stored through the VDB module. Arbitrary Perl functions can make this information available for other entities of a VoIP system.
Accounting	Although an AAA-System is the preferred concept in the context of accounting, passing data to the VDB module is well possible. Under certain circumstances, it might be a good idea to pre-process the data passed from the `acc` module. The implementation of an accounting script using the Perl module will also be an option.

Data	Solution
Permission/ authorization database	In contrast to other modules, the `permissions` module reads data from specially formatted text files, rather than from a relational database. A database may be used as a cache only. To use the VDB module in this context, one might either make up fake cache entries or substitute the `permission` module's data access mechanism with an SQL back-end. Arbitrary dynamic permission management is very simple with the Perl module. A Perl function returning 1 for accepted and -1 for denied calls can take arbitrary decisions for authorization. A demonstration of such a function can be found below in section 11.2.4.
Authentication database	Before implementing a VDB back-end for the `auth_db` module, a user should always consider using a RADIUS server implementation. Using the VDB is possible, however; an adapter class is provided for demonstration purposes.

This list demonstrates that all data categories discussed in the requirements analysis can now be processed through either the Perl or the VDB modules. In most cases, both options are available.

The interaction of all of these categories with LDAP directories – originally the primary target of this work – is well possible through the `OpenSER::LDAPUtils::*` Perl class hierarchy.

11.2. Specific solutions

A number of the cases discussed above will be described here in further detail. Implementations for central tasks have been developed as examples. Their main aspects will be outlined.

11.2.1. Aliases

OpenSER's main task is the routing of SIP messages to the destined recipient. A central component in this concept is the mapping of addresses to others. This concept can be

realized through aliases. The task of alias mapping is relatively simple: Extract the relevant components of the SIP message and preprocess them; query the database with these information; if a result is found, postprocess it and rewrite the current RURI.

OpenSER's `alias_db` module provides a single function `alias_db_lookup` that queries the database for the recipient URI of the current message and, if found, substitutes the RURI with the one found in the database.

Being one of the smaller modules, the `alias_db` module consists of approximately 450 lines of C code. Most of that code implements the module infrastructure (function and parameter export, database interface binding, ...); the implementation of the `alias_db_lookup` function consists of about 130 lines of code. Additional 15 lines perform the rewriting of the RURI. The core function's task is the preparation of a database query, the execution of the query and the processing of the result.

11.2.1.1. alias_db and perlvdb

The VDB class hierarchy contains an adapter class for the `alias_db` module that allows for simple implementations of arbitrary data access for aliasing. A matching VTab implements a function that takes two parameters (original username/domain) and returns a hash with two elements (new username/domain). Sample VTabs for LDAP access as well as for Perl hashes are included. The following code excerpt shows the LDAP version:

```
sub query() {
	my $self = shift;
	my $alias_username = shift;
	my $alias_domain = shift;

	my $uri = "$alias_username\@$alias_domain";
	my $ldap = new OpenSER::LDAPUtils::LDAPConnection();

	my @ldaprows = $ldap->search("(&(ObjectClass=inetOrgPerson)(mail=$uri))",      10
		"ou=people,dc=collax,dc=com", "uid");

	if (@ldaprows[0]) {
		my $ret;
		$ret->{username} = @ldaprows[0];
		$ret->{domain} = "voip";
		return $ret;
```

```
        }
        return;
}                                                                        20
```

This implementation expects the LDAP schema to use addresses with the pattern user@host, while the `alias_db` module explicitly extracts the user and host part of these addresses. The `alias_ldap` VTab shown above thus has to concatenate the strings again. A considerable amount of time is lost on pointless string operations.

11.2.1.2. Aliasing in Perl

As described above, the necessary operations for finding aliases are relatively simple. A Perl-only implementation providing aliasing information similar to the one above could look like this:

```
sub ldapalias {
        my $m = shift;

        my $uri = $m->getParsedRURI();
        my $user = $uri->user();
        my $dom = $uri->host();

        my $ldap = new OpenSER::LDAPUtils::LDAPConnection();

        my @rows = $ldap->search("(&(ObjectClass=inetOrgPerson)(mail=$user\@$dom))",    10
                "ou=people,dc=collax,dc=com", "uid");

        if (@rows) {
                my $newuri = "@rows\@voip";
                $ret = $m->rewrite_ruri("sip:$newuri");
                return 1;
        } else {
                return -1;
        }
}                                                                        20
```

This version has a number of advantages over the VDB variant described above: data does not have to traverse the database API, unnecessary string operations are omitted, and it is more legible than the relational wrapping shown above.

Both versions solve the task equally well.

11.2.2. Authentication

In section 4.2.4 (Authentication services), it was explained that secure network services should be implemented based on dedicated authentication mechanisms. In this context, RADIUS servers will be the most common choice; its successor DIAMETER is not yet widely available[1], while Kerberos is not specified for use with SIP.

In small installations, setting up a RADIUS server can possibly be an overkill. OpenSER therefore contains the `auth_db` module that authenticates against a relational database through OpenSER's DB API. This module can be used in conjunction with the VDB adapter that was implemented for this thesis. The downside of this method is that this technique requires access to the unencrypted plain text passwords. A sample VTab that stores accounts in a Perl hash has been written for this thesis.

Comparing the other topics described in this chapter, SIP authentication mechanisms involve some more intelligence, as different kinds of cryptographic hashes have to be calculated and compared. A good understanding of the underlying processes as well as the concerned standards, especially the Digest Authentication defined by RFC 2617, is crucial. It is of course well possible to implement these cryptographic functions in Perl, but the risk of creating security leaks is real – even for experienced software developers.

This leads to a gradation of preferred setups: The most secure option is a AAA system (RADIUS or DIAMETER); if that is not possible, OpenSER's standard `auth_db` module should be used. If a non-standard database back-end is absolutely necessary, the administrator should choose the VDB variant. The Perl reimplementation should never be necessary and thus should rather not be considered.

11.2.3. Accounting

SIP does not always provide sufficient information to resolve the obvious telephony accounting information "user A called number B; the call began at time X and lasted for T"[2]. Due to this reason, reliable accounting in common environments should be done on an entity that intercepts the transported media streams – such as an Asterisk server.

Yet, OpenSER's `acc` module provides accounting for SIP calls that do not involve a media gateway. Again, the recommended setup is using a dedicated accounting back-

[1] In contrast to OpenSER's RADIUS implementation, the DIAMETER module does not rely on well-established libraries but implements the protocol on top of raw IP traffic. This technique is at least questionable.

[2] Imagine an established call; then, one plug is pulled. Although the call is interrupted, a "BYE" message is never transmitted. SIP-only accounting will not be able to realize the end of the call.

end – such as RADIUS or DIAMETER. For these systems, the `acc` module provides `acc_rad_request` and `acc_diam_request` functions. Relational databases may be used by calling the `acc_db_request` function.

A – fairly simple – VDB adapter for the `acc` module has been written. As accounting only writes to the database through the "insert" operation, a VTab only needs a single function that takes an array of values. The sample `flatstoresimulator` VTab writes the passed values to a text file; adding the values to a different type of data back-end would of course be possible.

If a user A calls user B who in turn forwards the call to user C, regular accounting will not detect a relation between the two "legs" of a single call. The `acc` module provides so-called multi call-legs accounting which will identify a connection of two messages. If this technique is not used (it is turned off by default; its implementation is not too sophisticated, either), the database accounting function just extracts some pieces of information from every SIP message processed. The `acc` module is not dialog-aware. As such, the `acc_db_request` function is quite simple and could easily be reimplemented in the Perl module.

11.2.4. Authorization

From the perspective of a SIP server, authorization means the acceptance or denial of SIP messages under certain circumstances. When a SIP server rejects INVITE messages, calls cannot be established. Some of the 40x status codes defined in the SIP protocol can be used to signal different aspects of failed authorization[3]:

- 401 - Unauthorized
- 402 - Payment required
- 403 - Forbidden
- 407 - Proxy Authentication Required

OpenSER uses conventional configuration statements to implement the concept of authorization depending on certain circumstances:

```
    ⋮
    if (method=="INVITE") {
```

[3]See the SIP RFC [46] for a more detailed description of these and other response codes.

```
        if (!arbitrary_permission_function()) {
                sl_send_reply("403", "Forbidden");
                exit;
        }
}
⋮
```

With the Perl module, arbitrary functions can be implemented that return -1 for forbidden, and 1 for allowed connections. The following sample demonstrates a function that will randomly allow 90% of calls and reject the other 10%:

```
sub arbitrary_permission_function {
        my $r = rand();

        if ($r <= 0.9) {
                return 1;
        } else {
                return -1;
        }
}
```

A Perl function will commonly evaluate the current time and date to allow or deny calls.

OpenSER features a `permissions` module that can authorize requests based on IP addresses/networks, URIs or SIP addresses. Unlike the majority of OpenSER's modules, this module does not operate on a relational database, but uses text files of a style similar to Unix `hosts.allow`/`deny` files. A relational database may be used for internal caching. Due to this design, a cooperation with the Perl module is not sensible. As the module does not use the database API to request its configuration, the VDB module cannot be used, either. The caching database could theoretically be relayed through the VDB, creating fake cache entries; this design was not implemented, as it is unclean.

The task of the `permissions` module thus can be described like this: evaluate a text file, compare data from the SIP message with this file, and return true or false. The module does not provide any level of abstraction from the technologies in use. This is not primarily due to bad design, but rather due to the fact that the "intelligent" part of the module is the processing of the input files. Querying an internally built data structure upon a request is a simple process.

Authorization based on the `group` module may be relayed through the VDB module.

11.3. Perl vs. ...

Based on the results, the modules can be compared with each other and with other projects with a similar scope.

11.3.1. Perl vs. VDB

During the End User Development discussion, it was found out that EUD environments should have access to internal and external interfaces of a system. This led to the development of two modules – the Perl module attaching to the module interface, and the VDB module, jumping in between the database API and arbitrary data sources.

In section 11.1, it was discussed which of the two new modules may be used to provide functionality in the different contexts. In almost all contexts, processing the necessary data is quite simple by using the Perl module. In a number of cases, the VDB module can be used to provide an arbitrary data source for other modules' requests.

The only case where a VDB implementation would be a better choice than a pure Perl implementation is OpenSER's database authentication module, `auth_db`. A Perl-only version would need complex handling of hashing and encryption functions – not a simple task. In all other cases, a simple and clean reimplementation of given functionality will be easier and shorter to develop, as can be seen in the comparison of the ldap aliasing implementations above. A benchmarking test on both of these versions was done. The following table shows average results for 5000 processed messages with simple aliasing implementations that directly fetch data from internal Perl variables:

Test	Time in μs
Perl, alias found	82.342600
Perl, alias not found	73.033400
alias_db on VDB, alias found	439.739800
alias_db on VDB, alias not found	342.038000
alias_db on mysql, alias found	353.686889
alias_db on mysql, alias not found	341.994200

These benchmarks as well as the discussion of the use cases above show that in real world environments the usefulness of the VDB module will probably be significantly lower than the Perl module's. The benchmark also shows that the performance impact of Perl is very low, compared to that of a database request. Last but not least, it shows that the processing speed of the VDB is significantly lower than the Perl module's. This

is not caused by design faults, but rather by the complex processing involved in the data transformation.

11.3.1.1. VDB usage

While the original requirements can mostly be fulfilled with the functionality provided by the Perl module, the VDB module may be the more interesting technology from an academic point of view. It extends the currently existing database layer instead of providing low level SIP message processing.

The following table describes the suitability of VDB relaying for all modules in the OpenSER release 1.2.0 that use the DB API:

Client module	Suitability	Reason(s)
acc	o	See discussion above.
alias_db	-	Aliasing in Perl module is trivial.
auth_db	-	LDAP should be implemented through RADIUS. Using other non-relational DBs will rarely be reasonable. In these cases, VDB can be used.
avpops	o	In most environments, AVPs will not be set manually. No reason for external databases.
cpl-c	- -	The DB is an internal storage for CPL scripts.
domain	++	Could query a list of all local domains which will commonly be present in LDAP directories.
domainpolicy	o	Module is only necessary in sophisticated setups. RDBMS can be used there.
group	+	Group data are not quite as security sensitive as user credentials. E.g., Unix groups could be queried.
imc	-	Internal database for IM conferences. No external interface.
jabber	- -	DB is for internal use only. Module is deprecated (xmpp should be used).

Client module	Suitability	Reason(s)
lcr	- -/o	The lcr module in release 1.2.0 requires the DB_CAP_RAW_QUERY capability for raw SQL requests. Not available in VDB. This restriction was removed shortly after the release; therefore using the VDB is now possible. The usefulness of this technique is questionable, as the requested data usually will not be available in long-established setups.
msilo	-	DB is used as internal data storage (for SIP "MESSAGE" method messages).
pa	?	Package regarded as "unstable" and obsoleted by presence module.
pdt	+	Reasonable (This module provides "prefix to domain translation")
permissions	- -	This module uses a DB as an internal cache, not as a data source. Authorization is simple in Perl module.
presence	-	Internal data storage
pua	-	Internal data storage
siptrace	-	siptrace simply stores full messages. The Perl function `getMessage()` will be simpler and faster; unlike with the siptrace module, however, callbacks from the tm/sl are currently not possible.
speeddial	+	Reasonable
uri_db	++	Rarely used module; if used, currently unregistered but existing users could be found.
usrloc	- -	The usrloc module uses the database as an internal storage; therefore, setting up different databases is questionable. The module's performance has a great impact on OpenSER's global processing speed; interference with its default behavior is not advisable.

Some modules can be found that may well be used with the virtual database back-end.

11.3.2. Perl vs. SEAS

In December 2006, a module called “SEAS”, the Sip Express Application Server, was contributed to the OpenSER code. This module provides the interface to the Application Server WeSIP. This server runs so-called SIP Servlets, which are Java Servlets very similar to web servlets e.g. running in a Tomcat web server. The `seas` module in OpenSER provides a single function that relays processed messages to the WeSIP server, using a proprietary binary protocol. Through this interface, SIP messages are transformed into the Java class javax.servlet.sip.SIPServletRequest or javax.servlet.sip.-SIPServletResponse. An API not unlike the Perl module’s provides access to the data within the message object.

WeSIP also provides an EUD environment, similar to the the Perl module. There are a number of significant differences, however:

Programming language While Perl focuses on simplicity and is widely used by system administrators, Java is more difficult to learn. Sophisticated data processing and the integration of web services may possibly be better implemented in Java; however, the simple requests and message modifications provided by the Perl module API are easier to use and to maintain.

Focus WeSIP was designed to create a converged SIP/HTTP environment, while the Perl module was meant to provide access to long-established data sources for OpenSER. The different focuses of the projects have had large impacts on the solutions, though the outcome – EUD environments for OpenSER – was similar.

WeSIP tries to integrate a SIP server with other existing resources. A (proprietary binary) message interface between OpenSER and WeSIP is used for this task. The Perl module does not inherently feature distributed networking mechanisms, although distant resources may be requested from within Perl.

Availability and license The WeSIP server as well as the SEAS connector are in a Beta state and thus currently not available for production use. Although the server will be available for free in non-commercial environments, commercial usage will require (yet unknown) license fees.

The Perl module is available as a stable part of the OpenSER 1.2 release and fully licensed under the GPL, while the VDB module may be part of the next release. Future steps have to be discussed with the OpenSER developer team.

11.3.3. Perl and VDB vs. ldap modules

During the requirements analysis, existing LDAP binding modules for OpenSER's ancestor SER were introduced.

The first one, implemented at the ETH Zürich, provides a one-on-one substitution of the recipient URI. By providing data in a conforming LDAP schema, aliasing is possible. Considering the log messages spread over the code, the module is not meant to be used as-is in production environments. It seems obvious that the module was developed for the special needs within the university, but adaption to other environments should well be possible.

The code contributed by Rogelio Baucells in 2006 is more sophisticated. Nevertheless, the underlying fundamental ldap module provides a new interface that cannot be used in the context of existing code; the only existing implementation uses the LDAP module as an authentication back-end. Porting existing modules to use the LDAP interface would be a laborious task.

Both modules provide access to LDAP, but do not consider other data sources. The VDB module, on the other hand, makes LDAP handling easy while not losing track of other data sources.

11.3.4. Perl vs. SIP-CGI

The SIP-CGI interface defined by RFC 3050 [32] shall provide a SIP analog to HTTP CGI scripts. While SIP-CGIs are passed full SIP messages, the Perl interface can rely on OpenSER's sophisticated parser and its functionality.

OpenSER does not have a SIP-CGI interface, but provides an `exec` module with a similar scope.

11.4. The VDB approach

The virtual database implemented for OpenSER represents an approach to access foreign data sources in software systems. Although the idea was developed for SIP environments, it seems reasonable to apply the same concept to other scenarios. A future investigation could examine the constraints on the projects on which the concept could be applied and evaluate in which software systems a virtual database could be used.

OpenSER uses a limited subset of relational database functionalities. Its requests use the database tables as two-dimensional arrays; subselects, joins and other sophisticated

database operations are not required. A future study could examine whether these types of requests can be relayed to functions in the sense of programming languages as well.

11.5. Perspective

The Perl and VDB extensions to the SIP router OpenSER that were developed during this thesis seem to provide a reasonable way to access long-established data sources in environments in which the program is to be installed. Additionally, the Perl module lets users implement arbitrarily complex routing decisions.

The combination of the Perl module and the virtual database were compared to other solutions with similar scopes – data access and EUD environments – in the last section. It was shown that they are able to circumvent shortcomings of other projects for different reasons.

In real world environments, the Perl module has to prove its usability and stability. The proposed modifications to the module fixup functions are expected to be implemented in the next months and will create an additional value for the functionality available for Perl scripts. All module functions can then be used from within Perl scripts. It could then be evaluated whether a full routing configuration can be implemented in Perl, using the normal configuration file only as a stub.

New functionality may be implemented in all parts of the code, i.e. in the Perl library as well as in the extension. The APIs provided by other modules, especially by the `tm` and `sl` modules, could provide valuable features for user scripts.

A comparison with other programming languages and possible integrations of them could show whether the choice of Perl as an EUD environment was the right choice. The embedding of Perl was the solution for the notation of functions. While this is well possible with an imperative language – including Perl –, a functional language might be used in an alternative implementation of an EUD environment.

Conceptionally, the more interesting module is the virtual database – despite the fact that it will probably not be used in as many real world setups. The concept of using a programming language for database mapping could be evaluated in contexts of completely other database related systems, too.

The integration of the VDB module in the public OpenSER repository is still an open task. Before doing so, more thorough testing is required, e.g. concerning its reaction on incorrect result sets. Additional adapters, especially for the ones marked with "+" or "++" in section 11.3.1.1, should be implemented.

Part VI.

Appendix

A. Accompanying CD and website

The CD included with this document includes the following media:

- A PDF version of this document
- PDF versions of some of the papers referenced
- OpenSER 1.2.0 including the Perl module
- The PerlVDB module
- The benchmark module

With the exception of papers under copyright, identical information is available from the website
`http://www.bastian-friedrich.de/study/diploma/`

B. Glossary

ACD: Automatic Call Distribution. In a call center, incoming calls are appended to a queue. As soon as a dispatcher becomes available, the oldest call is passed to this user

API: Application Programming Interface. Includes class, type, method and function interfaces of a software system

CGI: Common Gateway Interface. Web servers offer this interface to run programs that deliver dynamic content.

CPAN: Comprehensive Perl Archive Network. A repository of free, publicly available Perl modules. Almost every free existing Perl module can be downloaded from CPAN

CPL: Call Processing Language. An XML-based programming language specified in RFC 3880 "to describe and control Internet telephony services"

CPS: Calls Per Second. A metric used to describe the scalability of VoIP systems.

CTI: Computer Telephony Integration

CVS: Concurrent Versions System. A widely used open source versioning/revision control system

DBMS: Database Management System

DNS: Domain Name Service

FIFO: First In/First Out. A buffering approach. In this context, it refers to Unix FIFO buffers or named pipes to support a pipes and filters model of communication

FOKUS: Fraunhofer Institut für offene Kommunikationssysteme. The SIP Express Router (SER) was developed here

GNU: Recursive acronym for “GNU’s Not Unix”. This project provides software for Unix-like, but open source operating systems

GPL: GNU General Public License. The most widely used open source license. The Linux kernel as well as SER and OpenSER are licensed under the GPL

HTTP: Hyper Text Transfer Protocol. Used in the World Wide Web for requesting and transmitting web content

IAX: InterAsterisk eXchange. Proprietary network protocol used between multiple instances of the Asterisk VoIP server

IETF: Internet Engineering Task Force

ITU-T: International Telephony Union, Telecommunication Standardization Unit

IVR: Interactive Voice Response. The system automatically processes the first steps of user interaction, e.g. more detailled classification, customer ID, etc.

LDAP: Lightweight Directory Access Protocol. Today’s standard directory access protocol and sometimes used synonymously for the directory and the server program

MI: Message Interface. The administration interface of OpenSER. Multiple ways of access are possible, e.g. through FIFO or XMLRPC requests

MTA: Mail Transport Agent. A “mail server”, such as a postfix, qmail or sendmail installation

MySQL: One of the best known open source database management systems

ODBC: Open Database Connectivity. A standard software API that provides unified access for various SQL databases

OLE: Object Linking and Embedding. Microsoft’s variant of a component model. Implements interfaces for interaction of different programs

POD: Plain Old Documentation. An inline documentation format used by Perl.

POTS: Plain Old Telephony System. Refers to conventional, non-internet telephony

PSTN: Public Switched Telephone Network. While the term could theoretically include VoIP networks (which can be public switched, too), it usually is synonymiously used with POTS

RAS: Remote Access Service

RDBMS: Relational Database Management System. Most actively used DBMS belong to this category

RPID: Remote Party ID. A string representation of a user, e.g. his name. The same acronym is used for Rich Presence Information Data which can also be used in SIP environments.

RTCP: Realtime Transport Control Protocol

RTP: Realtime Transport Protocol

RURI: Recipient Uniform Resource Identifier. The destination address of SIP messages

SDP: Session Description Protocol. Is used e.g. for media negotiation in SIP based telephony

SIP: Session Initiation Protocol. Core protocol in most modern VoIP networks

SQL: Structured Query Language. The standard query language for relational database management systems

TCP: Transmission Control Protocol

TLS: Transport Layer Security. An encryption protocol, successor of SSL (Secure Socket Layer)

UAC: User Agent Client. Refers to the user agent that sends a SIP request, e.g. initiating a call

UAS: User Agent Server. Refers to the user agent that responds to a SIP message

UDP: User Datagram Protocol. A slim IP-based protocol for datagrams. SIP traffic is usually transmitted through UDP

UM: Unified Messaging. The integration of e-mail, fax, telephony/voice mail, SMS, MMS and others to allow user access to all of these media through a single interface

XS: eXternal Subroutine. Through this mechanism, native/C code can provide functions for Perl scripts

C. Tools

During the work on this thesis, the following meta tools have been used for text editing and evaluation purposes:

- Text processing: LyX 1.4.x, teTeX 3.0 (for LaTeX and PDFLaTeX)
- BibTeX and the GUI kbibtex for citations, references and bibliography
- Xfig for graphics
- Subversion for revision control
- POD::DocBook for the implemtation documentation
- luma and JXplorer for LDAP evaluation
- `http://tools.ietf.org/html/` for pretty prints of RFC documents

Bibliography

[1] ENUM-Testbetrieb bei der DENIC eG. http://www.denic.de/de/enum/allgemeines/trial/index.html.

[2] mod-perl 2.0 API. http://perl.apache.org/docs/2.0/api/index.html.

[3] Perldoc: Internals and C language interface. http://perldoc.perl.org/index-internals.html.

[4] SIOD: Scheme in One Defun. http://www.cs.indiana.edu/scheme-repository/imp/siod.html.

[5] The unixODBC Project home page. http://www.unixodbc.org/.

[6] voip-info.org: Open Source VOIP Software. http://www.voip-info.org/wiki/view/Open+Source+VOIP+Software.

[7] What is mod-perl? http://perl.apache.org/start/index.html.

[8] Wikipedia.org (german), Callcenter. http://de.wikipedia.org/wiki/Callcenter.

[9] Wikipedia.org, Information is not data. http://en.wikipedia.org/wiki/Information?oldid=81809486#Information_is_not_data.

[10] Gisle Aas. PerlGuts Illustrated. http://gisle.aas.no/perl/illguts/.

[11] Rogelio J. Baucells. SER: LDAP and LDAP authentication modules. http://www.iptel.org/ldap_and_ldap_authentication_modules.

[12] Marcel Baur. ldap SER Module. http://www.ethworld.ethz.ch/technologies/sipeth/ser_modules/ldap.

[13] Dieter Conrads. *Telekommunikation: Grundlagen, Verfahren, Netze*. Vieweg, Wiesbaden, 2004.

[14] Michel de Boer. twinkle softphone. `http://www.twinklephone.com/`.

[15] P. Faltstrom and M. Mealling. RFC 3761, The E.164 to Uniform Resource Identifiers (URI) Dynamic Delegation Discovery System (DDDS) Application (ENUM), April 2004.

[16] Gerd Flaig, Martin Hoffmann, and Siggi Langauf. *Internet-Telefonie. VoIP mit Asterisk und SER.* Open Source Press, München, January 2006.

[17] H. Fulton, G. Hurst, and H.E. Fulton. *The Ruby Way.* Sams Publishing, 2001.

[18] Erich Gamma, Richard Helm, and Ralph E. Johnson. *Design Patterns. Elements of Reusable Object-Oriented Software.* Addison-Wesley Longman, Amsterdam, 1 edition, March 1995.

[19] Giorgos Gousios and Diomidis Spinellis. A Comparison of Portable Dynamic Web Content Technologies for the Apache Web Server. In *Proceedings of the 3rd International System Administration and Networking Conference SANE 2002*, pages 103–119, May 2002. Best refereed paper award.

[20] Martin Grabmüller. Multiparadigmen-Programmiersprachen. Research report 2003-15 in *Forschungsberichte Fakultät IV – Elektrotechnik und Informatik*, Technische Universität Berlin, October 2003.

[21] Bernhard Gröne, Andreas Knöpfel, Rudolf Kugel, and Oliver Schmidt. *The Apache Modeling Project*, volume 5 of *HPI Technical Reports.* July 2004.

[22] Wolf-Dieter Haaß. *Handbuch der Kommunikationsnetze.* Springer, Heidelberg, 1997.

[23] M. Handley, V. Jacobson, and C. Perkins. RFC 4566, SDP: Session Description Protocol, July 2006.

[24] Mathias Hein, Michael Reisner, and Dr. Antje Voß. *Voice over IP: Sprach-Daten-Konvergenz richtig nutzen.* Franzis' Verlag, Poing, 2002.

[25] Olivier Jacques and Richard Gayraud. SIPp. `http://sipp.sourceforge.net/`.

[26] T. Johnson, S. Okubo, and S. Campos. RFC 3944, H.350 Directory Services, December 2004.

[27] Franz-Joachim Kauffels. *Moderne Datenkommunikation: Eine strukturierte Einführung.* Datacom, Bergheim, 1994.

[28] Rolf-Dieter Köhler. *Voice over IP.* mitp, Bonn, 2002.

[29] Vaclav Kubart. iptel.org: Vaclev's Performance Tests. Memory management in SER. `http://www.iptel.org/ser/doc/performance/vaclev`.

[30] Georg Lausen. *Datenbanken. Grundlagen und XML-Technologien.* Elsevier, Heidelberg, January 2005.

[31] Marc Lehmann. XS - Einführung und esoterische Anwendungen. `http://www.goof.com/pcg/marc/xs.html`, March 2000.

[32] J. Lennox, H. Schulzrinne, and J. Rosenberg. RFC 3050, Common Gateway Interface for SIP, January 2001.

[33] Dr. Thomas Letschert. Nichtprozedurale Programmierung in Python und anderen Sprachen. February 2005.

[34] Yossi Levanoni and Erez Petrank. An on-the-fly reference-counting garbage collector for java. *ACM Trans. Program. Lang. Syst.*, 28(1):1–69, 2006.

[35] Henry Lieberman, Fabio Paternò, and Voker Wulf, editors. *End User Development*, volume 9 of *Human-Computer Interaction Series.* Springer, 2006.

[36] Netcraft Ltd. Netcraft Web Server Survey. `http://news.netcraft.com/archives/web_server_survey.html`.

[37] Wenbo Mao and C. Boyd. Development of authentication protocols: some misconceptions and a new approach. In *Computer Security Foundations Workshop VII, 1994. CSFW 7. Proceedings*, pages 178–186, June 1994.

[38] M. Mealling and R. Daniel. RFC 2915, The Naming Authority Pointer (NAPTR) DNS Resource Record, September 2000.

[39] Daniel-Constantin Mierla. Build SIP-based VOIP Service With RADIUS AAA Using OpenSER And FreeRadius. `http://www.openser.org/docs/openser-radius-1.0.x.html`.

[40] P. Mockapetris. RFC 1034, Domain names - concepts and facilities, November 1987.

[41] P. Mockapetris. RFC 1035, Domain names - implementation and specification, November 1987.

[42] Athanasios Nianias. Master thesis: Passwort im Datenexpress - Mit LDAP(ower) durchs Netzwerk. http://www.ks.uni-freiburg.de/download/diplomarbeit/SS06/09-06-ldap-anianias/, July 2006.

[43] Nils Ohlmeier. sipsak - SIP swiss army knife. http://sipsak.org/.

[44] Norman Ramsey. Embedding an interpreted language using higher-order functions and types. In *IVME '03: Proceedings of the 2003 workshop on Interpreters, virtual machines and emulators*, pages 6–14, New York, NY, USA, 2003. ACM Press.

[45] C. Rigney, S. Willens, A. Rubens, and W. Simpson. RFC 2865, Remote Authentication Dial In User Service (RADIUS), June 2000.

[46] J. Rosenberg, H. Schulzrinne, G. Camarillo, A. Johnston, J. Peterson, R. Sparks, M. Handley, and E. Schooler. RFC 3261, SIP: Session Initiation Protocol, June 2002.

[47] Vespe Savikko. Generative and Incremental Approach to Scripting Support Implementation. In Ban Al-Ani, Hamid R. Arabnia, and Youngsong Mun, editors, *Software Engineering Research and Practice*, pages 105–111. CSREA Press, 2003.

[48] Andreas Schael. TeleFAQ.de: Anruferfilter - Schutz vor ankommenden Rufen. http://www.telefaq.de/anruferfilter.html.

[49] Maik Schmitt and Rainer Jochem. Voice over IP. Fortgeschrittenenpraktikum. Master's thesis, Universität des Saarlandes, 2004.

[50] H. Schulzrinne, S. Casner, R. Frederick, and V. Jacobson. RFC 3550, RTP: A Transport Protocol for Real-Time Applications, July 2003.

[51] Nandu Shah. Pod::DocBook. http://search.cpan.org/dist/Pod-DocBook/lib/Pod/DocBook.pm.

[52] Venkita Subramonian, Liang-Jui Shen, and Christopher Gill. The Design and Performance of Dynamic and Static Configuration Mechanisms in Component Middleware for Distributed Real-Time and Embedded Systems. *25th IEEE International Real-Time Systems Symposium*, 5 December 2004.

[53] Alistair Sutcliffe. Evaluating the costs and benefits of end-user development. In *WEUSE I: Proceedings of the first workshop on End-user software engineering*, pages 1–4, New York, NY, USA, 2005. ACM Press.

[54] T-Com. Das Sicherheitspaket Plus für T-Net und T-ISDN. `http://www.t-com.de/is-bin/INTERSHOP.enfinity/WFS/EKI-PK-Site/de_DE/-/EUR/ViewBrowseCatalog-Start?CategoryName=00011600005\&CategoryDomainName=EKI-PK-DefaultCatalog`.

[55] Prof. Dr. Peter Thiemann. Seminar: Anwendungsorientierte Programmiersprachen. 2000.

[56] Miklos Tirpak and Juha Heinanen. OpenSER permissions module. `http://www.openser.org/docs/modules/1.2.x/permissions.html`, 2003.

[57] Arie van Deursen, Paul Klint, and Joost Visser. Domain-specific languages: an annotated bibliography. *SIGPLAN Not.*, 35(6):26–36, 2000.

Index

AAA services, 33
Accounting, 50
ACD, 47
Answering machine, 49
Arrays, 94
Asterisk, 12
Attribute/value pair, 56
AVPs, 85

b2bua, 10
Back-to-back user agent, 10

Call forking, 49
Call hunt group, 47
Caller ID, 49, 122
Calls per second, 115
Capability flags, 59, 86, 107, 132
Capability matrix, 107
Challenge-response authentication, 32
Circuit switched network, 7
Client side scripting, 69
Coding guidelines, 93
Collax GmbH, 2, 104, 121
Computer Telephony Integration, 8
Conference room, 50
CPAN, 68
CPL, 131
CTI, 8
CVS, 92

DBMS, 30
dbtext, 52
DENIC, 35
Digest authentication, 32
Directory, 31
dlopen, 97
DNS, 34
DocBook, 106

e164.arpa, 34
Event, 58
Export flags, 108

FIFO, 93
FOKUS, 15
Forking, 111

Garbage collection, 95
getBody, 100
GNU, 91
Google Talk, 9
GPL, 15

H.323, 8
h2xs, 102
Hashes, 94
HTTP, 10
Hyper Text Transfer Protocol, 10

IAX, 12
IETF, 8
INI style, 56
Internet Engineering Task Force, 8
IPC, 99
iptelorg, 15
ISDN, 7
ITU-T, 8
IVR, 47

JavaScript, 69
Jingle, 9, 12

LDAP, 31
LDAP module for SER, 52
LDAP schema, 31
ldapalias, 126
Lines of code, 21
Location service, 13
Logging, 111

Malware, 74
Media gateway, 10, 12
Media server, 10
Method, 10
module_exports, 22
MTA, 28
Music on hold, 10
MySQL, 19

Namespace, 72, 86, 105

ODBC, 19
Out-of-band signalling, 8

Packet switched network, 8
PBX, 7
Perl data types, 94
Perl functions, 84
Perl interpreter, 94
perlresult2dbres, 110
Pingtel, 13
Plain Old Telephone System, 7
POD, 106
pod2docbook, 106
PostgreSQL, 19
POTS, 7
Prototypes, 92
PSTN, 7
Public Switched Telephone Network, 7

Quality characteristics, 114, 117

RAS, 33
RDBMS, 30
Reference counting, 95, 108
Registrar service, 13
Regular expression, 116
Relational algebra, 86
Relational calculus, 86
Relational schema, 53
Remote Access Service, 33
Request/Response Model, 10
Result set, 109
RPID, 49
RTCP, 9
RTP, 9
Runtime errors, 117
RURI, 10

Scalars, 94
Schema, 31
SDP, 9
Session Initiation Protocol, 8
Shared library, 94, 97

Shared memory, 99
SIP, 8, 9
SIP event, 58
SIP method, 10
SIP proxy, 10
SIPfoundry.org, 13
SipX, 13
Skype, 9
Small and Medium size Business, 2
SMB, 2
Speed dialing, 49
Subversion, 92

TCP, 10
Tekelec, 15
TLS, 10
Tool chain, 91
Triple-A services, 33
Type signature, 75
Type system, 68

UAC, 10
UAS, 10
UDP, 10, 115
UM, 8
Unified Messaging, 8
unixODBC, 19
User agent, 10
User groups, 50

VBScript, 69
Virus, 74
VOCAL, 13
Voice box, 49
Voice mail, 49

Watchdog, 117
Wrapper pattern, 84

xlog, 20, 102
XS, 91, 100
xsubpp, 92