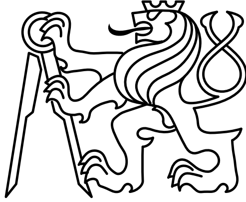

Czech Technical University
Faculty Of Electrical Engineering
Department Of Computer Science

SIP Proxy Server Effectiveness

Master's Thesis

Jan Janák

Prague, 21st May 2003

Assignment

Note: Real printed assignment goes on this page, this is just a placeholder.

Based on the analysis of bottlenecks in the SIP proxy server technology utilized in VoIP networks do design and implement modifications, and document improvement of effectiveness. Concentrate your effort to the protocol analysis, memory management and optimizing of caching strategies on single and multi-processor machines.

Abstract

Performance of SIP servers doesn't seem to gain much focus. Developers usually focus on implementation of new features that not standardized yet. Our goal was to develop a high performance SIP server that will be flexible and efficient. Result is the SIP Express Router, fast and efficient SIP proxy server. We briefly describe basics of SIP, then we describe architecture of the server and performance optimizations. Finally we present an overview of bottlenecks in the SIP protocol that make the performance tuning much harder.

Keywords: SIP, proxy, performance, bottlenecks

Anotace v českém jazyce

Výkonová optimalizace SIP serverů zatím příliš nepoutá pozornost vývojářů. Většina z nich se soustředí zejména na implementaci nových vlastností které zatím ani nebyly standardizovány. Našim cílem bylo vytvořit rychlý a flexibilní SIP server. Výsledkem je SIP Express Router, jeden z nejrychlejších SIP proxy serverů. V dokumentu nejdříve popíšeme základy SIPu, dále následuje popis architektury serveru a výkonových optimalizací. Nakonec shrneme několik problémů v návrhu SIP protokolu, které znesnadňují implementaci efektivního SIP serveru.

Klíčová slova: SIP, proxy, rychlost, výkon

Declaration

Hereby I declare that this Master's thesis was created entirely by myself, that I listed all sources used to create the thesis and that all quotations of the sources are complete.

I agree to using of the thesis by Department Of Computer Science, FEE, CTU for non-commercial purposes.

Acknowledgements

First of all I would like express my appreciation and thanks to my advisor Dipl.-Ing. Jiri Kuthan for his guidance and support, as well as many inspiring discussions and constructive criticism. He deserves recognition for his tremendous contribution to this work. He gave me an opportunity to learn from many great people and I will be forever in his debt.

Further, I would like to express many thanks to Dr.-Ing. Dorgham Sisalem, head of Mobis Competence Center at FhG FOKUS, who allowed me to work in such a pleasant and challenging environment. His support was very important to me during the development of this work.

My special thanks go to Doc. Ing. Jan Janeček, CSc., my mentor at the Czech Technical University in Prague, for all the valuable discussions, support, and understanding.

My gratitude goes to my entire family, especially my parents, who always supported and encouraged my studies over the years.

Contents

1 Introduction **1**
1.1 Motivation . . . 2
1.2 State of the Art . . . 3

2 Overview of SIP **5**
2.1 Purpose of SIP . . . 5
2.1.1 SIP URI . . . 6
2.2 SIP Network Elements . . . 7
2.2.1 User Agents . . . 7
2.2.2 Proxy Servers . . . 7
2.2.3 Registrar . . . 10
2.2.4 Redirect Server . . . 11
2.3 SIP Messages . . . 11
2.3.1 SIP Requests . . . 13
2.3.2 SIP Responses . . . 14
2.4 SIP Transactions . . . 16
2.5 SIP Dialogs . . . 17
2.5.1 Dialogs Facilitate Routing . . . 18
2.5.2 Dialog Identifiers . . . 20
2.6 Typical SIP Scenarios . . . 20
2.6.1 Registration . . . 20
2.6.2 Session Invitation . . . 21
2.6.3 Session Termination . . . 21
2.6.4 Record Routing . . . 22
2.6.5 Event Subscription and Notification . . . 22
2.6.6 Instant Messages . . . 23

3 The SIP Express Router **25**
3.1 Design Decisions 25
3.1.1 Targeted Operating Systems 25
3.1.2 Extensibility 26
3.1.3 Implementation Language 27
3.1.4 Concurrency Mechanism 28
3.2 The Server Architecture 30
3.2.1 The Configuration File 31
3.2.2 The Server Core 33
3.2.3 Modules 34
3.2.4 SIP Message Translations 36
3.3 Counted Strings 37
3.3.1 Performance Evaluation 39
3.4 Record Routing 39
3.4.1 r2 Parameter 40
3.4.2 Fast Host Comparison 41
3.4.3 Performance Evaluation 42
3.5 The Message Parser 43
3.5.1 Hand-crafted Parser Versus Yacc 44
3.5.2 Lazy Parsing 44
3.5.3 Incremental Parsing 46
3.5.4 32-bit Header Field Parser 47
3.5.5 Performance Evaluation 52
3.6 Registrar and User Location 53
3.6.1 Persistent Storage Cache 54
3.6.2 Persistence Modes 56
3.6.3 Hashing 57
3.6.4 Locking Optimization 58
3.6.5 Performance Evaluation 58
3.7 Memory Management 58

4 Performance Evaluation **60**
4.1 SIP Traffic Modeling 60
4.1.1 Transport Protocol 61
4.1.2 Authentication 61
4.2 Performance Metrics 61
4.3 Test-bed Overview 62

4.4 Performance Evaluation 62

5 SIP Protocol Bottlenecks **64**
5.1 Header Field Ordering 65
5.1.1 Order of Request Headers 66
5.1.2 Order of Response Headers 67
5.2 Hostnames Versus IP Addresses 67
5.3 Line Folding 68
5.4 Scattered Header Fields 69
5.5 Authentication 69
5.6 Display Name 71
5.7 SIP Over TCP 72
5.8 Careful Design of SIP Networks 73

6 Conclusion **75**
6.1 Achieved Results 75
6.2 Real Deployment 75
6.3 Future Work 76

Bibliography **78**

List of Figures

2.1 UAC and UAS . . . 8
2.2 Session Invitation . . . 10
2.3 Registrar Overview . . . 11
2.4 SIP Redirection . . . 12
2.5 SIP Transactions . . . 17
2.6 SIP Dialog . . . 18
2.7 SIP Trapezoid . . . 19
2.8 REGISTER Message Flow . . . 20
2.9 INVITE Message Flow . . . 21
2.10 BYE Message Flow (with and without record routing) . . . 22
2.11 Event Subscription and Notification . . . 23
2.12 Instant Messages . . . 24

3.1 The Server Architecture . . . 30
3.2 Data Lumps . . . 37
3.3 String Representation . . . 38
3.4 Lazy Parsing . . . 46
3.5 Incremental Parsing . . . 47
3.6 8-bit Parsing Automaton . . . 48
3.7 32-bit Parsing Automaton . . . 49
3.8 Registrar Cache . . . 55

5.1 Digest Challenge Message Flows . . . 70
5.2 SIP Message Read Over TCP . . . 74

List of Tables

5.1 Equivalent SIP Messages 66
5.2 Different SIP Messages 67

Chapter 1

Introduction

The Internet. What was originally an interconnection of supercomputers at universities became a global network connecting millions of computers all over the world. As of today almost every field of human movements is affected by the Internet. Internet is used to book flights, order books, buy goods, transfer money, people use e-mail more and more instead of the traditional mail.

The Internet is used in areas where nobody would expect it—during the War in Iraq in 2003 electronic mail was the major communication means used by the Czech soldiers to communicate with their families.

But there is still one area which hasn't been penetrated by the Internet yet—real-time multimedia communications. By real-time multimedia communication we understand communication of two or more entities that communicate in real time—like in a telephone conversation—and possibly use more than one type of media. For example users can use audio, video and text in a single session.

The reason why Internet is not so widely used in real-time communications yet is that such a communication imposes much higher requirements on the underlying network. There must be enough bandwidth available to carry multimedia data, delay of the network must be as short as possible and predictable, it must be possible to reserve required bandwidth or the network must be over-dimensioned so the reservation will be not necessary.

According to the [13] average round trip time between hosts located in the North America and Europe has been decreased to less than 500ms. Such a small delay already allows real-time communication. Delay higher than approximately 300ms in one direction practically prohibits real-time communication because people involved in the conversation start to experience unpleasant delay.

SIP (Session Initiation Protocol) is one of protocols used for signalling. The protocol has been developed within the IETF, it is open text-based protocol. SIP seems to be gaining accep-

tance during the past years. It is popular because it allows seamless integration of many existing services, including voice, video, and data. We believe that SIP is going to be widespread in the near future. The fact that this protocol has been adopted by Microsoft for its real-time communication platform means that the protocol will hit millions of users soon.

1.1 Motivation

Performance requirements laid upon a SIP server (i.e. how many requests per second a server should be able to handle) vary according to the population served. A server serving a small company with 50 employees doesn't have the same requirements as a core server serving millions of users, for instance in the next-generation mobile networks.

Today's GSM networks in Europe have millions of subscribers. If we consider a 3G mobile networks with SIP implementation in each handset, it is clear that core SIP servers in the network will serve huge population of subscribers. And the architecture of 3G networks contains many proxy elements.

Operators of such huge networks will definitely need to minimize operational costs and reduce per-subscriber capital. A network containing less hardware tends to be more reliable and requires less administrators to operate the hardware.

SIP signalling is mostly handled by SIP proxies, registrars and redirection servers.

SIP is going to be used as integrated signalling protocol for instant messaging, presence, voice calls, and multimedia conferences. Services integrated that way will probably have much broader audience and will attract many new users.

To do a rough estimation of how many people could possibly be using SIP in the future, let's take a look on how big population some well known services serve. As presented in [15]:

- **AT&T**—Connects 280 million calls per day.
- **Yahoo**—The Yahoo web servers serve 780 million pages per day.
- **AOL**—110 million e-mails per day.
- **AOL**—500 million instant messages per day.

If we consider that SIP could be used as a unified signalling protocol for most of the services then it is clear that the need for very fast and scalable SIP proxy servers will arise.

A SIP proxy with higher performance can also handle much better avalanches of requests caused by broken or badly configured user agents. Broken user agents sometimes tend to send SIP messages as fast as the underlying network allows. We have experienced many user agents flooding our public SIP server.

Thus, SIP server performance is important and should be considered when designing and implementing any SIP server.

1.2 State of the Art

Many companies developing SIP applications and servers unfortunately do not reveal performance numbers of their products or simply do not take care of performance measurements at all.

From SIP Interoperability events, that we have participated in, we know that most of the products available there can handle less than 800 Calls Per Seconds at sustainable rate. Measurement tools available at the event usually can generate low thousands of calls per second. It is important to say that not all companies developing SIP products attend those events. Usually there are 50–70 companies attending. Many of the companies are actually "big players" in the industry.

Vovida (later acquired by Cisco) has developed a free SIP Server. The SIP server is implemented in C++. From performance measurements performed by Vovida we know that the SIP server can handle about 40 calls per second on a 900 MHz Pentium III with 256 MB of memory and Linux as the operating system. This performance bottleneck is rooted in the design of the server.

Mockingbird Networks has announced a SIP server that can handle up to 200 calls per second at sustainable rate. The performance measurements were done using RadCom performance measuring tool. According to Mockingbird Networks the server was able to handle 2000 simultaneous connections.

The MegaSIP is a product of RadCom. The MegaSIP is a carrier grade SIP call simulator. The simulator provides a setup rate of 2000 calls per seconds. During one of the Sip-It events in 2002 we measured performance of our server using the call simulator. Our server running on an IBM laptop with common configuration shipped with our server fully saturated the call generator.

Sipstone is an ongoing project of the Columbia University and Ubiquity. Purpose of this project is to create a benchmark for testing of SIP servers and also to developed a utility implementing the benchmark. The benchmark attempts to measure the request handling capacity of a SIP server. It is a single implementation benchmark. The benchmarking is done in a controlled test bed. Sipstone tries to estimate how a typical SIP traffic will look like and generate traffic with similar characteristics.

Other available SIP server will be described only briefly because their parameters are not known.

- Cisco Proxy Server is a SIP server based on Apache web server. The performance numbers are not known, but this proxy is not very flexible which leads us to the assumption that the server can hardly be optimized for speed.
- Ubiquity has developed a SIP server and many other SIP products in Java.
- dynamicsoft Routing Engine is a SIP proxy written in Java. We know that this proxy server can do more than 500 calls per second on a multi-CPU machine with Solaris.
- HotSip's Application Server can be probably configured as a SIP Proxy. Nothing is known about performance but applications servers are usually very complex and the performance is a secondary concern.
- Indigo has a SIP Server along with SDK. Unfortunately nothing is known about the performance.

Companies and organizations presented here are the most important players in SIP area, our server was tested for interoperability with many of them during SIP-It events.

Chapter 2

Overview of SIP

Since the thesis deals with performance optimizations of a SIP server, readers should be familiar with the basics of SIP and should have an idea how SIP signalling looks like. Because the specification is very complex, we will present a brief overview here. The description is not meant to be exhaustive, that would be out of the scope of the document. Should the reader be interested in a more detailed description, he will find some documents to consider in the bibliography section.

2.1 Purpose of SIP

SIP stands for Session Initiation Protocol. It is an application-layer control protocol which has been developed and designed within the IETF. The protocol has been designed with easy implementation, good scalability, and flexibility in mind.

The specification is available in form of several RFCs, the most important one is RFC3261 [12] which contains the core protocol specification. The protocol is used for creating, modifying, and terminating sessions with one or more participants. By sessions we understand a set of senders and receivers that communicate and the state kept in those senders and receivers during the communication. Examples of a session can include Internet telephone calls, distribution of multimedia, multimedia conferences, distributed computer games, etc.

SIP is not the only protocol that the communicating devices will need. It is not meant to be a general purpose protocol. Purpose of SIP is just to make the communication possible, the communication itself must be achieved by another means (and possibly another protocol). Two protocols that are most often used along with SIP are RTP [14] and SDP [5]. RTP protocol is used to carry the real-time multimedia data (including audio, video, and text), the protocol makes it possible to encode and split the data into packets and transport such packets over the

Internet. Another important protocol is SDP, which is used to describe and encode capabilities of the session participants. Such a description is then used to negotiate the characteristics of the session so that all the devices can participate (that includes, for example, negotiation of codecs used to encode media so all the participants will be able to decode it, negotiation of transport protocol used and so on).

SIP has been designed in conformance with the Internet model. It is an end-to-end oriented signalling protocol which means, that all the logic is stored in end devices (except routing of SIP messages). State is also stored in end-devices only, there is no single point of failure and networks designed this way scale well. The price that we have to pay for the distributiveness and scalability is higher message overhead, caused by messages sent end-to-end.

It is worth of mentioning that the end-to-end concept of SIP is a significant divergence from regular PSTN (Public Switched Telephone Network) where all the state and logic is stored in the network and end devices (telephones) are very primitive. Aim of SIP is to provide the same functionality that the traditional PSTNs have, but the end-to-end design makes SIP networks much more powerful and open to implementation of new services that can be hardly implemented in the traditional PSTNs.

SIP is based on HTTP protocol. The HTTP protocol inherited format of message headers from RFC 822 [2]. HTTP is probably the most successful and widely used protocol in the Internet. It tries to combine best of the both. In fact, HTTP can be classified as a signalling protocol too, because user agents use the protocol to tell a HTTP server in which documents they are interested in. SIP is used to carry the description of session parameters, the description is encoded into a document using SDP (Session Description Protocol). Both protocols (HTTP and SIP) have inherited encoding of message headers from RFC 822 [2]. The encoding has proven to be robust and flexible over the years. But later we will show that the described advantages can easily turn into the weakness from performance point of view.

2.1.1 SIP URI

SIP Entities are identified using SIP URI (Uniform Resource Identifier). A SIP URI has form of `sip:username@domain`, for instance, `sip:joe@company.com`. As we can see, SIP URI contains of username part and domain name part delimited by `@` character. SIP URIs are similar to e-mail addresses, it is, for instance, possible to use the same URI for e-mail and SIP communication, such URIs are easy to remember.

2.2 SIP Network Elements

Although in the simplest configuration it is possible to use just two user agents that send SIP messages directly to each other, a typical SIP network will contain more than one type of SIP elements. Basic SIP elements are user agents, proxies, registrars, and redirect servers. We will briefly describe them in this section.

Note that the elements, as presented in this section, are often only logical entities. It is often profitable to co-locate them together, for instance, to increase the speed of processing, but that depends on a particular implementation and configuration.

2.2.1 User Agents

Internet end points that use SIP to find each other and to negotiate a session characteristics are called *user agents*. User agents usually, but not necessarily, reside on a user's computer in form of an application—this is currently the most widely used approach, but user agents can be also cellular phones, PSTN gateways, PDAs, automated IVR systems and so on.

User agents are often reffered to as *User Agent Server* (UAS) and *User Agent Client* (UAC). UAS and UAC are logical entities only, each user agent contains a UAC and UAS. UAC is the part of the user agent that sends requests and receives responses. UAS is the part of the user agent that receives requests and send responses.

Because a user agent contains both UAC and UAS, we often say that a user agent behaves like a UAC or UAS. For instance, caller's user agent behaves like UAC when it sends an INVITE requests and receives responses to the request. Callee's user agent behaves like a UAS when it receives the INVITE and sends responses.

But this situation changes when the callee decides to send a BYE and terminate the session. In this case the callee's user agent (sending BYE) behaves like UAC and the caller's user agent behaves like UAS.

Figure 2.1 shows three user agents and one stateful forking proxy. Each user agent contains UAC and UAS. The part of the proxy that receives the INVITE from the caller in fact acts as a UAS. When forwarding the request statefully the proxy creates two UACs, each of them is responsible for one branch.

In our example callee B picked up and later when he wants to tear down the call it sends a BYE. At this time the user agent that was previously UAS becomes a UAC and vice versa.

2.2.2 Proxy Servers

In addition to that SIP allows creation of an infrastructure of network hosts called *proxy servers*. User agents can send messages to a proxy server. Proxy servers are very important entities in the

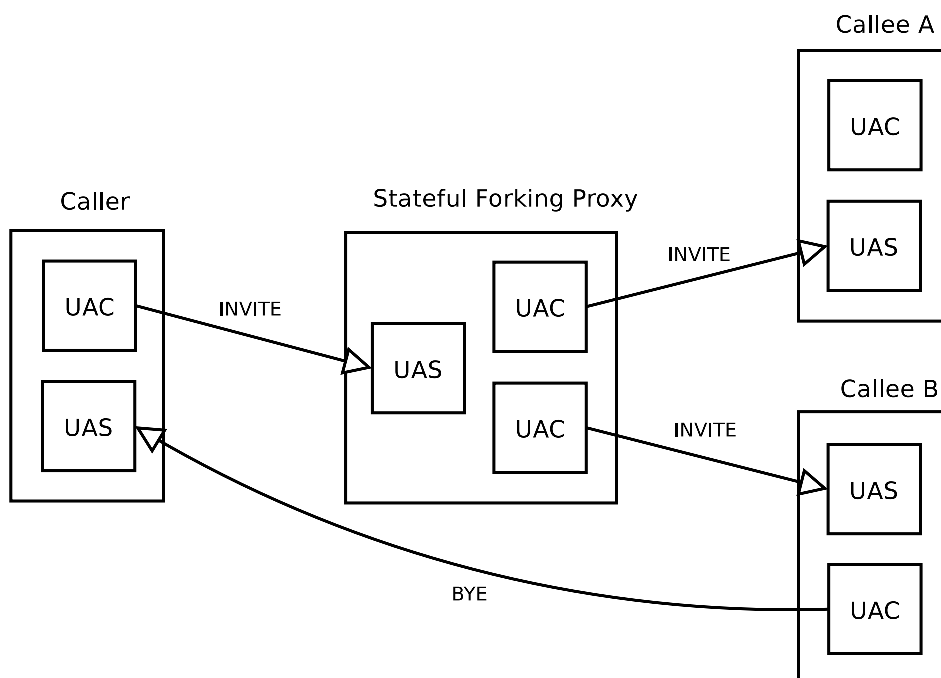

Figure 2.1: UAC and UAS

SIP infrastructure. They allow registration of user agents, perform routing of a session invitation according to invitee's current location, authentication, accounting and many other important functions.

The most important task of each proxy server is to route session invitations "closer" to callee. The session invitation will usually traverse a set of proxies until it finds one which knows the actual location of the callee. Such a proxy will forward the session invitation directly to the callee and the callee will then accept or decline the session invitation.

There are two basic types of SIP proxy servers—stateless and stateful.

Stateless Servers

Stateless server work as simple message forwarders. They forward messages independently of each other. Although messages are usually arranged into transactions (see section 2.4), stateless proxies do not take care of transactions.

Stateless proxies are simple, but faster than stateful proxy servers. They can be used as simple load balancers, message translators and routers. One of drawbacks of stateless proxies is that are unable to absorb retransmissions of messages and perform more advanced routing, for instance, forking or recursive traversal.

Stateful Servers

Stateful proxies are more complex. Upon reception of a request stateful proxies create a state and keep the state until the transaction finishes. Some transactions, especially those created by INVITE, can last quite long (until callee picks up or declines the call). Because stateful proxies must maintain the state for the duration of the transaction, it limits performance of the proxy.

The ability to associate SIP messages into transactions gives stateful proxies some interesting features. Stateful proxies can perform forking, that means upon reception of a message two or more messages will be sent out.

Stateful proxies can absorb retransmissions because they know, from the transaction state, if they have already received the same message (stateless proxies cannot do the check because they keep no state).

Stateful proxies can perform more complicated methods of finding a user. It is, for instance, possible to try to reach user's office phone and when he doesn't pick up then the call is redirected to his cell phone. Stateless proxies can not do this because they have no way of knowing how the transaction targeted to the office phone finished.

Most SIP proxies today are stateful because their configuration is usually very complex. They often perform accounting, forking, some sort of NAT traversal aid and all those features require a stateful proxy.

Proxy Server Usage

A typical configuration is that each centrally administered entity (a company, for instance) has it's own SIP proxy server which is used by all user agents in the entity. Let's suppose that there are two companies A and B and each of them has it's own proxy server. Figure 2.2 shows how a session invitation from employee Joe in company A will reach employee Bob in company B.

User Joe uses address `sip:bob@b.com` to call Bob. Joe's user agent doesn't know how to route the invitation itself but it is configured to send all outbound traffic to the company SIP proxy server `proxy.a.com`. The proxy server figures out that user `sip:bob@b.com` is in a different company so it will look up B's SIP proxy server and send the invitation there. B's proxy server can be either preconfigured at `proxy.a.com` or the proxy will use DNS SRV records to find B's proxy server. The invitation reaches `proxy.b.com`. The proxy knows that Bob is currently sitting in his office and is reachable through phone on his desk, which has IP address 1.2.3.4, so the proxy will send the invitation there.

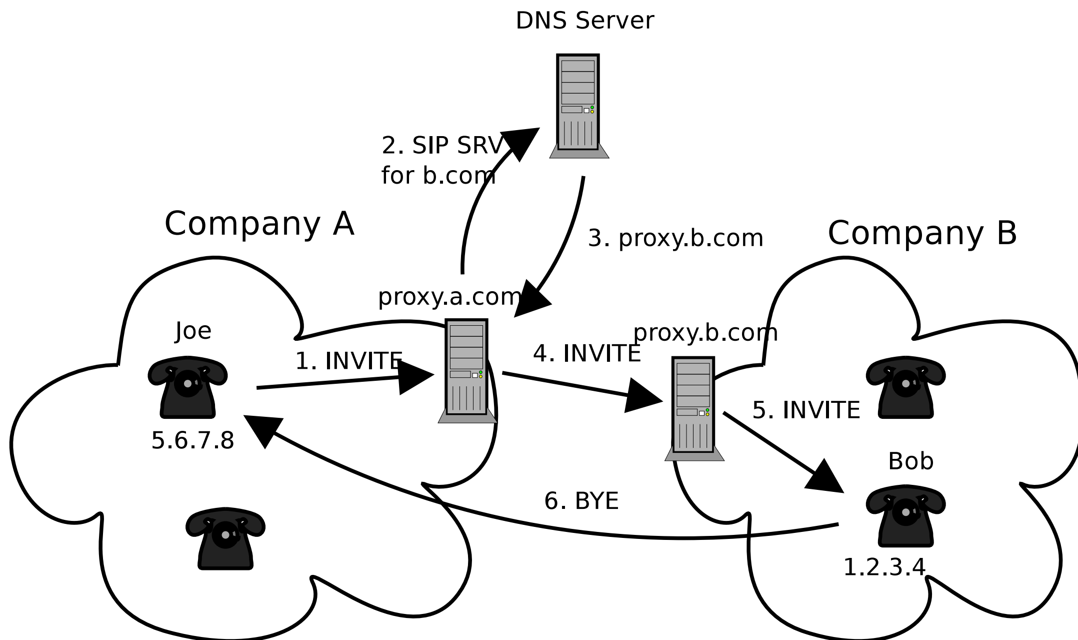

Figure 2.2: Session Invitation

2.2.3 Registrar

We mentioned that the SIP proxy at `proxy.b.com` knows current Bob's location but haven't mentioned yet how a proxy can learn current location of a user. Bob's user agent (SIP phone) must register with a *registrar*. The registrar is a special SIP entity that receives registrations from users, extracts information about their current location (IP address, port and username in this case) and stores the information into location database. Purpose of the location database is to map `sip:bob@b.com` to something like `sip:bob@1.2.3.4:5060`. The location database is then used by B's proxy server. When the proxy receives an invitation to `bob@b.com` it will search the location database. It finds `sip:bob@1.2.3.4:5060` and will send the invitation there. A registrar is very often a logical entity only. Because of their tight coupling with proxies registrars are usually co-located with proxy servers.

Figure 2.3 shows a typical SIP registration. A REGISTER message containing Address of Record `sip:jan@iptel.org` and contact address `sip:jan@1.2.3.4:5060` where 1.2.3.4 is IP address of the phone, is sent to the registrar. Registrar extracts this information and stores it into the location database. If everything went well the registrar sends a 200 OK response to the phone and the process of registration is finished.

Each registration has a limited lifespan. `Expires` header field or `expires` parameter of

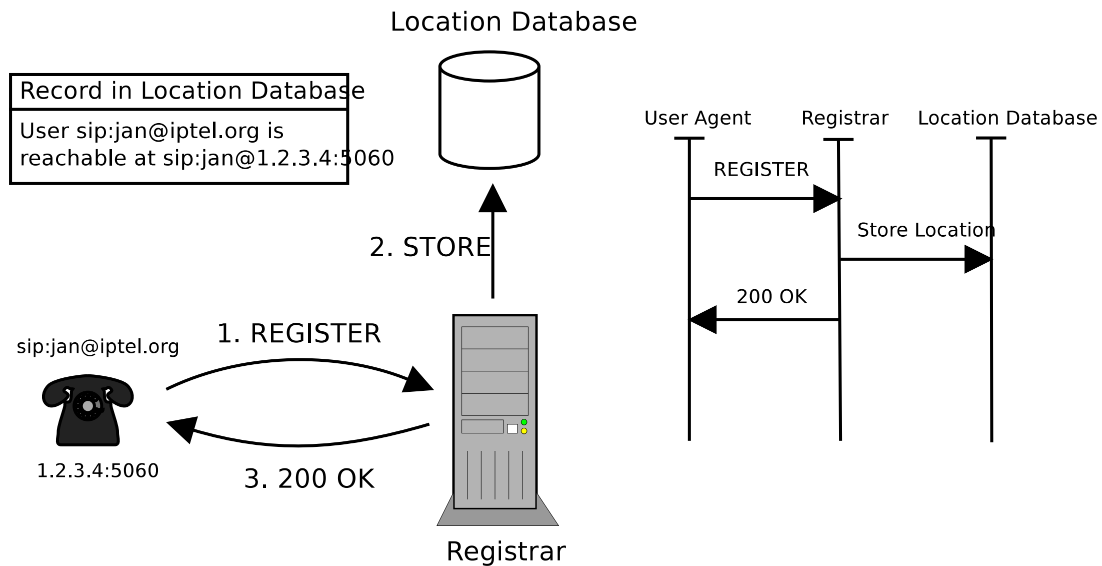

Figure 2.3: Registrar Overview

`Contact` header fields determines how long is the registration valid. The user agent must refresh the registration within the lifespan otherwise it will expire.

2.2.4 Redirect Server

The entity that receives a request and sends back a reply containing a list of the current location of a particular user is called *redirect server*. A redirect server receives a requests and looks up the intended recipient of the request in the location database created by a registrar. It then creates a list of current locations of the user and sends it to the request originator in a response with 3xx class.

The originator of the request then extracts the list of destinations and sends another request directly to them. Figure 2.4 shows a typical redirection.

2.3 SIP Messages

Communication using SIP (often called signalling) comprises of series of *messages*. Messages can be transported independently by the network. Usually they are transported in a separate UDP datagram each. Each message consist of "first line", message header, and message body. The first line identifies type of the message. There are two types of messages—*requests* and *responses*. A requests are usually used to initiate some action or inform recipient of the request

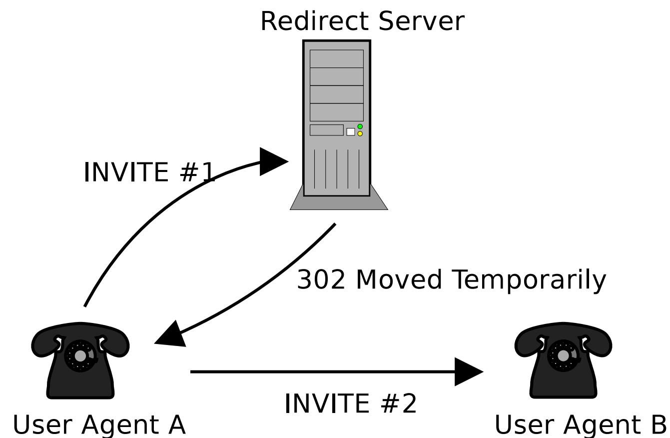

Figure 2.4: SIP Redirection

of something. Replies are used to confirm that a request was received and processed and contain the status of the processing.

A typical SIP request looks like this:

```
INVITE sip:7170@iptel.org SIP/2.0
Via: SIP/2.0/UDP 195.37.77.100:5040;rport
Max-Forwards: 10
From: "jiri" <sip:jiri@iptel.org>;tag=76ff7a07-c091-4192-84a0-d56e91fe104f
To: <sip:jiri@bat.iptel.org>
Call-ID: d10815e0-bf17-4afa-8412-d9130a793d96@213.20.128.35
CSeq: 2 INVITE
Contact: <sip:213.20.128.35:9315>
User-Agent: Windows RTC/1.0
Proxy-Authorization: Digest username="jiri", realm="iptel.org",
  algorithm="MD5", uri="sip:jiri@bat.iptel.org",
  nonce="3cef753900000001771328f5ae1b8b7f0d742da1feb5753c",
  response="53fe98db10e1074
b03b3e06438bda70f"
Content-Type: application/sdp
Content-Length: 451

v=0
o=jku2 0 0 IN IP4 213.20.128.35
s=session
c=IN IP4 213.20.128.35
b=CT:1000
t=0 0
m=audio 54742 RTP/AVP 97 111 112 6 0 8 4 5 3 101
a=rtpmap:97 red/8000
a=rtpmap:111 SIREN/16000
a=fmtp:111 bitrate=16000
```

```
a=rtpmap:112 G7221/16000
a=fmtp:112 bitrate=24000
a=rtpmap:6 DVI4/16000
a=rtpmap:0 PCMU/8000
a=rtpmap:4 G723/8000
a=rtpmap: 3 GSM/8000
a=rtpmap:101 telephone-event/8000
a=fmtp:101 0-16
```

The first line tells us that this is INVITE message which is used to establish a session. The URI on the first line—`sip:7170@iptel.org` is called *Request URI* and expresses next hop of the message. In this case it will be host `iptel.org`.

A SIP Request can contain one or more `Via` header fields which are used to record path of the request. They are later used to route SIP responses exactly the same way. The INVITE message contains just one `Via` header field which was created by the user agent that sent the request. From the `Via` field we can tell that the user agent is running on host 195.37.77.100 and port 5060.

`From` and `To` header fields identify initiator (caller) and recipient (callee) of the invitation (just like in SMTP where they identify sender and recipient of message). `From` header field contains a tag parameter which serves as a dialog identifier and will be described in section 2.5.

`Call-ID` header field is a dialog identifier and it's purpose is to identify messages belonging to the same call. Such messages have the same `Call-ID` identifier. `CSeq` is used to maintain order of requests. Because requests can be sent over an unreliable transport that can re-order messages, a sequence number must be present in the messages so that recipient can identify retransmissions and out of order requests.

`Contact` header field contains IP address and port on which the sender is awaiting further requests sent by callee. Other header fields are not important and will be not described here.

Message header is delimited from message body by an empty line. Message body of the INVITE request contains a description of the media type accepted by the sender and encoded in SDP.

2.3.1 SIP Requests

We have described how an INVITE request looks like and said that the request is used to invite a callee to a session. Other important requests are:

- **ACK**—This message acknowledges receipt of a final response to INVITE. Establishing of a session utilizes 3-way hand-shaking due to asymmetric nature of the invitation. It may take a while before the callee accepts or declines the call so the callee's user agent periodically retransmits a positive final response until it receives an ACK (which indicates that the caller is still there and ready to communicate).

- **BYE**—Bye messages are used to tear down multimedia sessions. A party wishing to tear down a session sends a BYE to the other party.
- **CANCEL**—Cancel is used to cancel not yet fully established session. It is used when the callee hasn't replied with a final response yet but the caller wants to abort the call (typically when a callee doesn't respond for some time).
- **REGISTER**—Purpose of REGISTER request is to let registrar know of current user's location. Information about current IP address and port on which a user can be reached is carried in REGISTER messages. Registrar extracts this information and puts it into a location database. The database can be later used by SIP proxy servers to route calls to the user. Registrations are time-limited and need to be periodically refreshed.

The listed requests usually have no message body because it is not needed in most situations (but can have one). In addition to that many other request types have been defined but their description is out of the scope of this document.

2.3.2 SIP Responses

When a user agent or proxy server receives a request it send a reply. Each request must be replied except ACK requests which trigger no replies.

A typical reply looks like this:

```
SIP/2.0 200 OK
Via: SIP/2.0/UDP 192.168.1.30:5060;received=66.87.48.68
From: sip:sip2@iptel.org
To: sip:sip2@iptel.org;tag=794fe65c16edfdf45da4fc39a5d2867c.b713
Call-ID: 2443936363@192.168.1.30
CSeq: 63629 REGISTER
Contact: <sip:sip2@66.87.48.68:5060;transport=udp>;q=0.00;expires=120
Server: Sip EXpress router (0.8.11pre21xrc (i386/linux))
Content-Length: 0
Warning: 392 195.37.77.101:5060 "Noisy feedback tells:
    pid=5110 req_src_ip=66.87.48.68 req_src_port=5060 in_uri=sip:iptel.org
    out_uri=sip:iptel.org via_cnt==1"
```

As we can see, responses are very similar to the requests, except for the first line. The first line of response contains protocol version (`SIP/2.0`), reply code, and reason phrase.

The *reply code* is an integer number from 100 to 699 and indicates type of the response. There are 6 classes of responses:

- **1xx** are *provisional* responses. A provisional response is response that tells to its recipient that the associated request was received but result of the processing is not known yet.

Provisional responses are sent only when the processing doesn't finish immediately. The sender must stop retransmitting the request upon reception of a provisional response.

Typically proxy servers send responses with code 100 when they start processing an INVITE and user agents send responses with code 180 (Ringing) which means that the callee's phone is ringing.

- **2xx** responses are *positive final* responses. A final response is the ultimate response that the originator of the request will ever receive. Therefore final responses express result of the processing of the associated request. Final responses also terminate transactions. Responses with code from 200-299 are positive responses which means that the request was processed successfully and accepted. For instance a 200 OK response is sent when a user accepts invitation to a session (INVITE request).

  A UAC may receive several 200 messages to a single INVITE request. This is because a forking proxy (described later) can fork the request so it will reach several UAS and each of them will accept the invitation. In this case each response is distinguished by the tag parameter in `To` header field. Each response represents a distinct dialog with unambiguous dialog identifier.

- **3xx** responses are used to redirect a caller. A redirection response gives information about the user's new location or an alternative service that the caller might use to satisfy the call. Redirection responses are usually sent by proxy servers. When a proxy receives a request and doesn't want or can't process it for any reason, it will sent a redirection response to the caller and put another location into the response which the caller might to try. It can be the location of another proxy or the current location of the callee (from the location database created by a registrar). The caller is then supposed to re-send the request to the new location. 3xx responses are final.

- **4xx** are *negative final* responses. A 4xx response means that the problem is on the sender's side. The request couldn't be processed because it contains bad syntax or cannot be fulfilled at this server.

- **5xx** means that the problem is on server's side. The request is apparently valid but the server failed to fulfill it. Clients should usually retry the request later.

- **6xx** reply code means that the request cannot be fulfilled at any server. This response is usually sent by a server that has definitive information about a particular user. User agents usually send a 603 Decline response when the user doesn't want to participate in the session.

In addition to the response class the first line also contains *reason phrase*. The code number is intended to be processed by machines. It is not very human-friendly but it is very easy to parse and understand by machines. Reason phrase usually contains a human-readable message describing the result of the processing. A user agent should render the reason phrase to the user.

The request to which a particular response belongs is identified using the `CSeq` header field. In addition to the sequence number this header field also contains method of corresponding request. In our example it was REGISTER request.

2.4 SIP Transactions

Although we have said that SIP messages are sent independently over the network, they are usually arranged into *transactions* by user agents and certain types of proxy servers. Therefore SIP is said to be a *transactional protocol*.

A transaction is a sequence of SIP messages exchanged between SIP network elements. A transaction consists of one request and all responses to that request. That includes zero or more provisional responses and one or more final responses (remember that an INVITE might be answered by more than one final response when a proxy server forks the request).

If a transaction was initiated by an INVITE request then the same transaction also includes ACK, but only if the final response was not a 2xx response. If the final response was a 2xx response then the ACK is not considered part of the transaction.

As we can see this is quite asymmetric behavior—ACK is part of transactions with a negative final response but is not part of transactions with positive final responses. The reason for this separation is the importance of delivery of all 200 OK messages. Not only that they establish a session, but also 200 OK can be generated by multiple entities when a proxy server forks the request and all of them must be delivered to the calling user agent. Therefore user agents take responsibility in this case and retransmit 200 OK responses until they receive an ACK. Also note that only responses to INVITE are retransmitted !

SIP entities that have notion of transactions are called *stateful*. Such entities usually create a state associated with a transaction that is kept in the memory for the duration of the transaction. When a request or response comes, a stateful entity tries to associate the request (or response) to existing transactions. To be able to do it it must extract a unique transaction identifier from the message and compare it to identifiers of all existing transactions. If such a transaction exists then it's state is updated from the message.

In the previous SIP RFC [6] the transaction identifier was calculated as hash of all important message header fields (that included `To`, `From`, Request-URI and `CSeq`). This proved to be very slow and complex, during interoperability tests such transaction identifiers used to be a common

source of problems.

In the new RFC [12] the way of calculating transaction identifiers was completely changed. Instead of complicated hashing of important header fields a SIP message now includes the identifier directly. Branch parameter of Via header fields contains directly the transaction identifier. This is a significant simplification, but there still exist old implementations that don't support the new way of calculating transaction identifier so even new implementations have the support the old way. They must be backwards compatible.

Figure 2.5 shows what messages belong to what transactions during a conversation of two user agents.

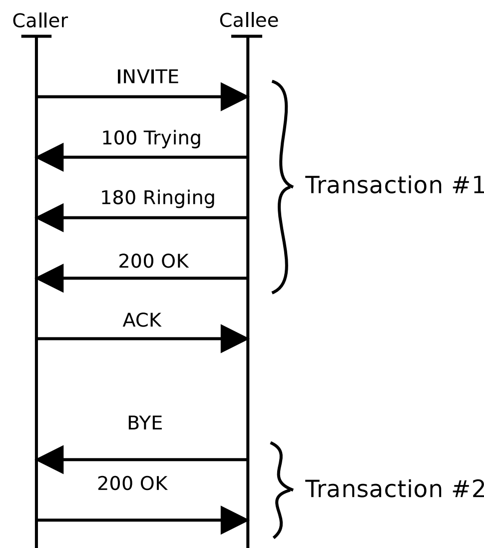

Figure 2.5: SIP Transactions

2.5 SIP Dialogs

We have shown what transactions are, that one transaction includes INVITE and it's responses and another transaction includes BYE and it responses when a session is being torn down. But we feel that those two transactions should be somehow related—both of them belong to the same *dialog*. A dialog represents a peer-to-peer SIP relationship between two user agents. A dialog persists for some time and it is very important concept for user agents. Dialogs facilitate proper sequencing and routing of messages between user agents.

Dialogs are identified using `Call-ID`, `From` tag, and `To` tag. Messages that have these three identifiers same belong to the same dialog. We have shown that `CSeq` header field is used to order

messages, in fact it is used to order messages within a dialog. The number must be monotonically increased for each message sent within a dialog otherwise the peer will handle it as out of order request or retransmission. In fact, the `CSeq` number identifies a transaction within a dialog because we have said that requests and associated responses are called transaction. This means that only one transaction in each direction can be active within a dialog. One could also say that a *dialog is a sequence of transactions*. Figure 2.6 extends figure 2.5 to show which messages belong to the same dialog.

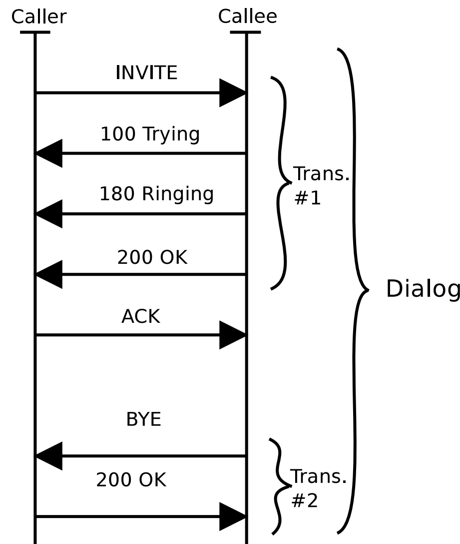

Figure 2.6: SIP Dialog

Some messages establish a dialog and some do not. This allows to explicitly express the relationship of messages and also to send messages that are not related to other messages outside a dialog. That is easier to implement because user agent don't have to keep the dialog state.

For instance, INVITE message establishes a dialog, because it will be later followed by BYE request which will tear down the session established by the INVITE. This BYE is sent within the dialog established by the INVITE.

But if a user agent sends a MESSAGE request, such a request doesn't establish any dialog. Any subsequent messages (even MESSAGE) will be sent independently of the previous one.

2.5.1 Dialogs Facilitate Routing

We have said that dialogs are also used to route the messages between user agents, let's describe this a little bit.

Let’s suppose that user `sip:bob@a.com` wants to talk to user `sip:pete@b.com`. He knows SIP address of the callee (`sip:pete@b.com`) but this address doesn’t say anything about current location of the user—i.e. the caller doesn’t know to which host to send the request directly. Therefore the INVITE request will be sent to a proxy server.

The request will be sent from proxy to proxy until it reaches one that knows current location of the callee. This process is called routing. Once the request reaches the callee, the callee’s user agent will create a response that will be sent back to the caller. Callee’s user agent will also put `Contact` header field into the response which will contain the current location of the user. The original request also contained `Contact` header field which means that both user agents know the current location of the peer.

Because the user agents know location of each other, it is not necessary to send further request to any proxy—they can be sent directly from user agent to user agent. That’s exactly how dialogs facilitate routing.

Further messages within a dialog are sent directly from user agent to user agent. This is a significant performance improvement because proxies do not see all the messages within a dialog, they are used to route just the first request that establishes the dialog. The direct messages are also delivered with much smaller latency because a typical proxy usually implements complex routing logic. Figure 2.7 shows an example of a message within a dialog (BYE) that bypasses the proxies.

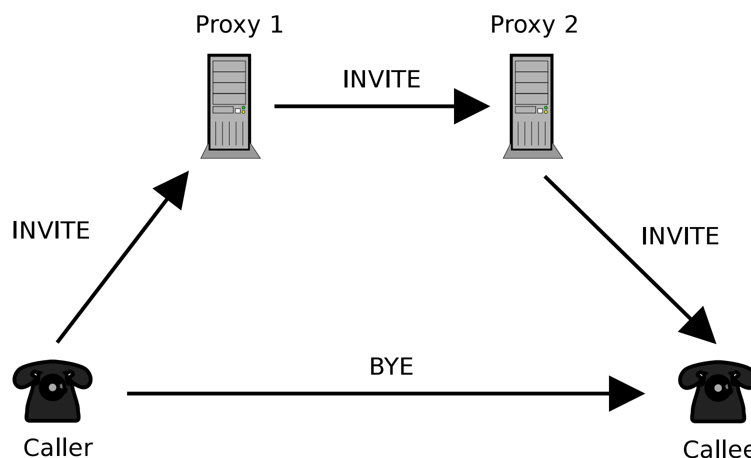

Figure 2.7: SIP Trapezoid

2.5.2 Dialog Identifiers

We have already shown that dialog identifiers consist of three parts, Call-Id, `From` tag, and `To` tag, but it is not that clear why are dialog identifiers created exactly this way and who contributes which part.

`Call-ID` is *call identifier*. It must be a unique string that identifies a call. A call consists of one or more dialogs. Multiple user agents may respond to a request when a proxy along the path forks the request. Each user agent that sends a 2xx establishes a separate dialog with the caller. All such dialogs are part of the same call and have the same `Call-ID`.

`From` tag is generated by the caller and it uniquely identifies the dialog in the caller's user agent.

`To` tag is generated by a callee and it uniquely identifies, just like `From` tag, the dialog in the callee's user agent.

This hierarchical dialog identifier is necessary because a single call invitation can create several dialogs and caller must be able to distinguish them.

2.6 Typical SIP Scenarios

This section gives a brief overview of typical SIP scenarios that usually make up the SIP traffic.

2.6.1 Registration

Users must register themselves with a registrar to be reachable by other users. A registration comprises a REGISTER message followed by a 200 OK sent by registrar if the registration was successful. Registrations are usually authorized so a 407 reply can appear if the user didn't provide valid credentials. Figure 2.8 shows an example of registration.

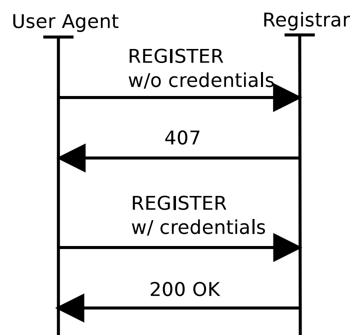

Figure 2.8: REGISTER Message Flow

2.6.2 Session Invitation

A session invitation consists of one INVITE request which is usually sent to a proxy. The proxy sends immediately a 100 Trying reply to stop retransmissions and forwards the request further.

All provisional responses generated by callee are sent back to the caller. See 180 Ringing response in the call flow. The response is generated when callee's phone starts ringing.

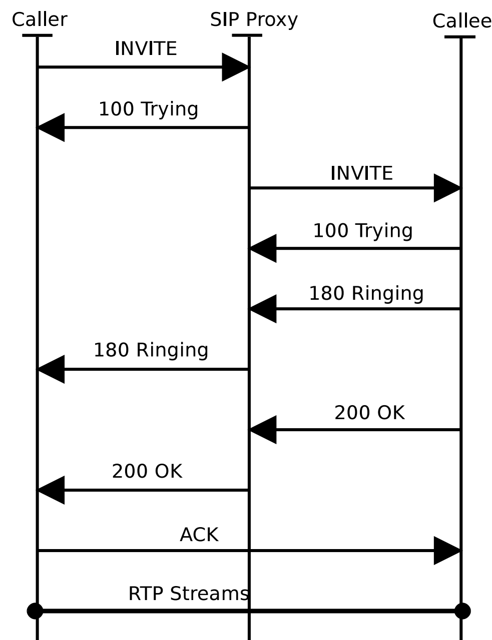

Figure 2.9: INVITE Message Flow

A 200 OK is generated once the callee picks up the phone and it is retransmitted by the callee's user agent until it receives an ACK from the caller. The session is established at this point.

2.6.3 Session Termination

Session termination is accomplished by sending a BYE request within dialog established bye INVITE. BYE messages are sent directly from one user agent to the other unless a proxy on the path of the INVITE request indicated that it wishes to stay on the path by using record routing (see section 2.6.4).

Party wishing to tear down a session sends a BYE request to the other party involved in the session. The other party sends a 200 OK response to confirm the BYE and the session is terminated. See figure 2.10, left message flow.

2.6.4 Record Routing

All requests sent within a dialog are by default sent directly from one user agent to the other. Only requests outside a dialog traverse SIP proxies. This approach makes SIP network more scalable because only a small number of SIP messages hit the proxies.

There are certain situations in which a SIP proxy need to stay on the path of all further messages. For instance, proxies controlling a NAT box or proxies doing accounting need to stay on the path of BYE requests.

Mechanism by which a proxy can inform user agents that it wishes to stay on the path of all further messages is called *record routing*. Such a proxy would insert `Record-Route` header field into SIP messages which contains address of the proxy. Messages sent within a dialog will then traverse all SIP proxies that put a `Record-Route` header field into the message.

The recipient of the request receives a set of `Record-Route` header fields in the message. It must mirror all the `Record-Route` header fields into responses because the originator of the request also needs to know the set of proxies.

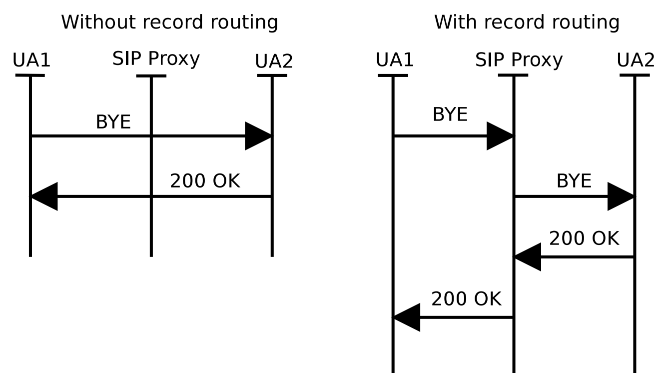

Figure 2.10: BYE Message Flow (with and without record routing)

Left message flow of figure 2.10 show how a BYE (request within dialog established by INVITE) is sent directly to the other user agent when there is no `Record-Route` header field in the message. Right message flow show how the situation changes when the proxy puts a `Record-Route` header field into the message.

2.6.5 Event Subscription and Notification

SIP specification has been extended to support a general mechanism allowing subscription to asynchronous events. Such evens can include SIP proxy statistics changes, presence information, session changes and so on.

The mechanism is used mainly to convey information on presence (willingness to communicate) of users. Figure 2.11 show the basic message flow.

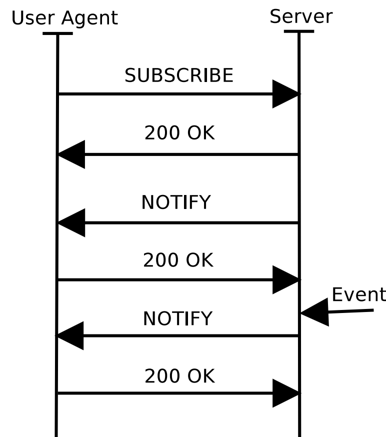

Figure 2.11: Event Subscription and Notification

A user agent interested in event notification sends a SUBSCRIBE message to a SIP server. The SUBSCRIBE message establishes a dialog and is immediately replied by the server using 200 OK response. At this point the dialog is established. The server sends a NOTIFY request to the user every time the event to which the user subscribed changes. NOTIFY messages are sent within the dialog established by the SUBSCRIBE.

Note that the first NOTIFY message in figure 2.11 is sent regardless of any event that triggers notifications.

Subscriptions—as well as registrations—have limited lifespan and therefore must be periodically refreshed.

2.6.6 Instant Messages

Instant messages are sent using MESSAGE request. MESSAGE requests do not establish a dialog and therefore they will always traverse the same set of proxies. This is the simplest form of sending instant messages. The text of the instant message is transported in the body of the SIP request.

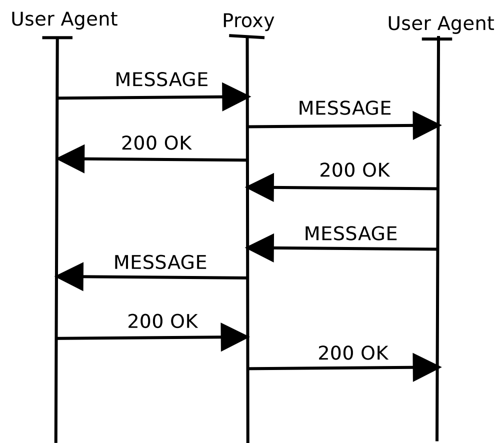

Figure 2.12: Instant Messages

Chapter 3

The SIP Express Router

The SIP Express Router is our attempt to develop a high-performance SIP server. Because there are only couple of SIP servers publicly available and none of them is suitable for performance optimizations—mainly because they are badly designed or written in some interpreted language like Java—we decided to implement a new server from scratch.

This decision gives us the possibility to do some design decisions that, we believe, will impact the performance a lot. We will briefly describe the design decisions and the architecture of the server and then we will describe individual performance optimizations and evaluate their performance. At the end of the chapter the performance of the whole server will be evaluated using the sipstone [16] benchmark.

3.1 Design Decisions

Because we designed the SIP server from scratch, we did many design decisions that could affect the performance of the server. Some of the design decisions will be described in this section.

It is not possible to evaluate impact of the design decisions on the performance of the server because that would require re-implementation of the whole server, so we will at least describe why we decided to implement it this way.

3.1.1 Targeted Operating Systems

The primary operating system for the development is Linux. Because the server will be implemented in C, it is easy to port it to other POSIX compliant operating systems, like FreeBSD, Solaris, OpenBSD and many others.

Because we use inline assembler to implement fast locks, it will be necessary to port the inline assembler instructions to all targeted hardware platform. If it is not possible for any reason then

the locking will be implemented using Sys-V mutexes.

3.1.2 Extensibility

A good SIP server must include many features of different kind. Administrators often request support for authentication, database access, accounting, user location, registrar, and many others. The proxy server has been designed to be fast. The speed is mainly important for core SIP proxies and we believe that such proxies don't need large set of additional functions, so not every part of the server has to be optimized for speed.

Because we were missing the operational experience during the development, we had to design the server to be extensible. That allows us to write just a minimum set of functions that are necessary at the moment and extend the server later when needed.

The extensibility is accomplished using *modules*. A module is a part of the server that is stored in a separate binary file and can be loaded into the memory when required. On UNIX-like systems modules are typically stored as the shared object files and the main application can ask the operating system to load a module using a system call. Symbols contained in a module can be then used by the main application when the module is loaded.

It is important to mention that most of the server logic is stored in modules. The server core contains only the minimum set of functions that will be needed by all modules and that are needed for the core itself.

The core of our server contains the following classes of functions:

- **Memory Management**—To speed up the server we have implemented our own memory management that all modules should use. The memory management is faster then the standard libc memory allocator because it knows the characteristics of our server.

- **Configuration Compiler**—Configuration compiler reads the configuration file in plain text form and compiles it into an internal binary representation that is used when processing SIP messages.

- **Message Parser and Builder**—Message parser gets a SIP message and produces a parsed structure containing pointers to the elements of the message. Message builder is used to convert the parsed structure and list of modifications back into a text message.

- **Module Interface**—Module interface contains functions necessary to load and use modules. Those functions include loading of modules, getting a list of exported functions, setting parameters and so on.

- **Transport Functions**—Transport functions hide the operating systems functions used to send or receive messages using the sockets API. They also include functions to open a TCP connection, perform a DNS query and many other related functions.
- **System Functions**—Wrappers used to hide differences among supported operating systems. That includes functions to perform fast assembly locking, forking, and signal handling.
- **System Log**—The server can generate debugging and error message either to the standard output (which can be redirected to a file) or syslog. In addition to that some additional functions and macros have been defined to support different kinds of log messages.
- **FIFO Interface**—The server can be controlled over a FIFO, which is a special character file. Data written to the file will be received by the server and data written by the server will be received by the user using the FIFO.

3.1.3 Implementation Language

Most of the companies developing SIP application have chosen Java as their implementation language. Success of Java is not surprising. Java seems to offer a mix of important features that are not so common in other languages. Code written in Java can be executed on any platform containing Java Virtual Machine (although people using Java often require that the applications run on Windows and Solaris only). Java is an Object Oriented language, object oriented approach is easier to understand and learn for developers. Java has garbage collector so developers don't have to take care of releasing unused memory. And last, but not least, there are plenty of Java developers available.

On the other hand, it is the garbage collector that can cause lots of troubles when developing SIP applications in Java. A heavily loaded server written in Java stops working when the garbage collector is cleaning the memory. The delay caused by the garbage collector can be even more than 10 seconds. Such delays are unacceptable, imagine that you have to wait more then 10 seconds for the ringing tone.

The delay can also trigger retransmissions of SIP messages when they are being sent over an unreliable transport like UDP. A user agent will retransmit a request if it doesn't receive any response within 500 ms. Round Trip Time between the U.S. and Europe is typically 150-200 ms [13]. Adding couple of milliseconds for processing at the user agent and proxy server, there isn't much time left for the delay caused by the garbage collector.

In addition to that Java is not very suitable for performance optimized applications because it's virtual machine adds too much unnecessary overhead.

Other languages to consider are C and C++. We decided to proceed with ANSI C because we mostly don't need features of C++. C-written applications can be easily ported to almost any POSIX-compliant system when written carefully and it is easy to optimize C applications for speed. ANSI C is also better supported among different compilers than C++.

3.1.4 Concurrency Mechanism

Heavily loaded SIP servers need some concurrency mechanism, because the processing of a single SIP message typically includes many slow operations, for instance, accessing a database over the network, radius authentication or radius accounting. Such operations are usually very slow compared to the processing logic. For acceptable performance the server must process several messages concurrently to exploit parallelism in the I/O subsystem. Because the slow operations would block the CPU, some concurrency mechanism will be necessary.

Processes

Processes are widely supported, they are the easiest way of achieving concurrency. Possibility to create separate processes is in every operating system and it is the most portable concurrency mechanism.

Processes are quite easy to program, communication among processes can be achieved using the standard inter-process communication mechanism, like shared memory or signals. Private memory doesn't require locking and it is much easier to develop programs without critical sections (but we will need the locking when accessing the shared memory anyway).

Each process has its own execution context and each process has its own private memory region. Because each process is scheduled independently by the scheduler of the operating system, it is not a very good idea to create a high number of processes by the server. That would overload the scheduler and the system would spend most of its time in the kernel scheduling processes in the runnable state.

Threads

Thread are similar to processes because they also have separate execution context, but, unlike processes, threads share the address space. Therefore it is necessary to use locking when accessing data structures stored in the memory. Such locks can easily become performance bottlenecks that are very hard to debug. A typical multi-threaded server consists of many threads of execution to exploit the parallelism contained in the application. A typical multi-threaded application contains many locks or conditions and that can lead to another problem—deadlock.

Also achieving high performance in a multi-threaded application is not easy. To make applications easy to program and understand, programmers often use simple locks, but simple locks yield low concurrency which leads to low performance.

To achieve high performance, fine-grained locking is often used, but that leads to complicated programs which are hard to implement and debug.

And there are few OS-related problems with threads as well:

- It is hard to port multi-threaded code
- Standard libraries are often not thread-safe
- There are only few debugging tools available

Events

Event is a mechanism that tries to avoid the concurrency wherever possible. Basic idea is to create an event loop. All slow operations like network I/O, filesystem access, database access are performed asynchronously and when finished they generate an event.

The event mechanism is promising, it is well suited for single CPU machines but doesn't scale well on multi-processor systems where real concurrency is necessary to utilize all CPUs available.

Another drawback is that the event mechanism is not very well supported across operating systems yet, mainly because not all operations can be performed asynchronously.

Conclusion

The decision whether to proceed with threads or processes is important. We have studied many papers that deal with performance of WWW servers, because there are no performance evaluations of SIP servers yet and WWW servers are similar to SIP servers. It is not clear whether multi-threaded processes are faster than process-based servers or vice versa. It is possible to find many papers stating that multi-threaded servers are faster than process-based servers and it is also possible to find many papers stating that the opposite is true.

The decision must also be based on the targeted operating system. Although most of them support threads of execution, implementation of threads is not always efficient.

Our main operating system is Linux. Implementation of threads in Linux is not very efficient. Threads, in fact, are implemented as separate processes with shared memory. That means all created threads are scheduled by the main kernel scheduler of the operating system. Creating a high number of threads can significantly decrease performance of the whole operating system, because the scheduler will have to schedule all of them and that would be slow.

Designing a multi-threaded application is very complex. Because threads share address space, it is necessary to use locking when accessing data structures stored in the memory. Multi-threaded programs usually require many locks. It is easy to create a deadlock which will occur rarely and such a deadlock will be hard to find.

Designing a high-performance multi-threaded application is not easy. To fully exploit parallelism contained in the application it is necessary to create high number of threads. Such complex systems usually suffer from unpredictable behavior from performance point of view.

Given all the facts we decided to use multiple processes to achieve the concurrency. It is easier to design such an application and also it is more portable. Nobody has proven yet that multi-threaded servers give better performance than process-based servers, it is always application-specific.

3.2 The Server Architecture

The server consists of the server core and extension modules. The core of the server loads the configuration file, parses it and converts into an internal binary representation. The binary structure is then used when a SIP message is being processed.

Upon startup the core creates several children processes (the number of the children is configurable). Most of the children simply wait for incoming SIP messages and when such a message arrives, one of the children will pick it up and start the processing.

There are also some additional children that take care of timers or handle the FIFO interface. Those children do not receive SIP messages from the I/O subsystem.

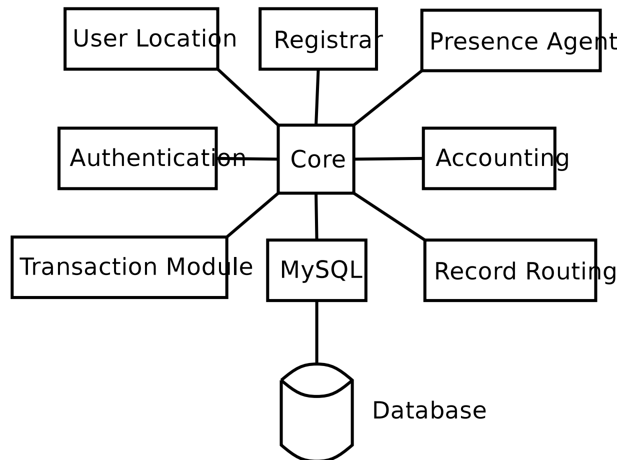

Figure 3.1: The Server Architecture

3.2.1 The Configuration File

Upon the start up the server reads and parses a configuration file. The configuration file contains the following information:

- Variable settings
- Modules that should be loaded
- Settings of module variables
- Description of routing logic

Configuration File Example

```
debug=3         # debug level (cmd line: -dddddddddd)
fork=yes        # Fork children
log_stderror=no # (cmd line: -E)
check_via=no # (cmd. line: -v)
dns=no          # (cmd. line: -r)
rev_dns=no      # (cmd. line: -R)
port=5060       # Listen on the default SIP port
children=4      # Fork 4 children
fifo="/tmp/ser_fifo"

# Load necessary modules
loadmodule "/usr/local/lib/ser/modules/sl.so"
loadmodule "/usr/local/lib/ser/modules/tm.so"
loadmodule "/usr/local/lib/ser/modules/rr.so"
loadmodule "/usr/local/lib/ser/modules/maxfwd.so"
loadmodule "/usr/local/lib/ser/modules/usrloc.so"
loadmodule "/usr/local/lib/ser/modules/registrar.so"

# Set module parameters
modparam("usrloc", "db_mode",   0)

# The main routing logic
route {
    # initial sanity checks -- messages with
    # max_forwards==0, or excessively long requests
    if (!mf_process_maxfwd_header("10")) {
        sl_send_reply("483","Too Many Hops");
        break;
    };
    if (len_gt( max_len )) {
        sl_send_reply("513", "Message too big");
        break;
    };

    record_route();
    # loose-route processing
```

```
    loose_route();

    if (uri==myself) {

        if (method=="REGISTER") {
            save("location");
            break;
        };

        # native SIP destinations are handled using our USRLOC DB
        if (!lookup("location")) {
            sl_send_reply("404", "Not Found");
            break;
        };
    };

    if (!t_relay()) {
        sl_reply_error();
    };
}
```

The beginning of the file contains settings of the core variables. The core variables control behavior of the server. In our example we set debug level to 3 to produce less debugging messages, instruct the server to spawn four children, log using syslog and so on.

The second part of the configuration file contains a series of `loadmodule` commands. This command loads required modules. Some modules depend on other modules so order of the loading is important.

The third part of the configuration file contains series of `modparam` commands (there is just one `modparam` command in our example). Those command set internal variables of loaded modules. The modules can be configured this way. Corresponding module must be loaded before the variables contained in the module can be set.

And the last part of the configuration file beginning with `route {` it the description of the routing logic. Commands contained in this part are executed sequentially each time a SIP message comes. The children processes execute this part independently of each other.

The Routing Logic

The most important and complex part of the configuration file is the routing logic. The routing logic describes how incoming SIP messages should be processed.

Although it doesn't may seem so, the routing logic is the *basic and most important mechanism for achieving good performance and scalability*. It is necessary to have full control over the server's behavior when we try to optimize the server and the control must be fine-grained.

Although performance tuning of the server's subsystems can improve the performance of the whole server, *disabling features that are not necessary* can improve the performance much more.

We can simply turn off features that are not absolutely necessary and tune the rest to achieve the best possible performance.

The routing language is similar to C. We believe that such a language is intuitive, easy to read, and easy to write. XML based configuration files seem to be very popular these days, but we are going to edit the configuration file by hand and thus a XML based format is not suitable for us. XML-based files are hard to read without special editors or viewers.

The routing part consists of several route sections, the route sections are distinguished by number. The main route section is without any number and this route section will be executed first. Other route sections can be called from the main section, the mechanism is similar to functions.

Each route section contains a series of commands which are executed sequentially. The commands can be either defined by the core or by one of the loaded modules. Each command can have up to two parameters. In addition to the command, route sections can also contain conditional statements and calls to other route blocks.

Every function called from the route section will get the SIP message being processed as the first argument. Our example does the following:

- Check if there is a `Max-Forwards` header field and if it contains zero. If so, send a 483 reply. If There is no `Max-Forwards` header field then add a new one with value 10.
- Check the size of the message and send back a 513 response if it is too big.
- Insert `Record-Route` header field at the beginning of the message.
- Process `Route` header fields.
- If the message is targeted to our server then save the location if it is a REGISTER message and perform location lookup if it is another message.
- If the message is not for our server then use `t_relay` to forward it statefully.

The configuration script is translated into a binary tree upon start up of the server. This speeds up the processing. All operations that can be done without knowing the SIP message like DNS queries or parameter conversions are done during the compilation.

3.2.2 The Server Core

The core of the server contains functions necessary for all modules or the core itself. The core receives SIP messages, parsers them and executes the routing script. The core must be small and fast, therefore all the additional functionality should be put into modules.

The core includes the message parser, memory management subsystem, transport subsystem (UDP and TCP support), message translator and so on.

3.2.3 Modules

Modules are separate units of the server that provide specialized functionality. The core communicates with the modules using the module interface which is common for all modules.

Each module can export a set of functions that can be used in various parts of the configuration file, set of parameter whose value can be set in the configuration file and that affect behavior of the module, and a generic set of callbacks and initialization functions that are called when specified event occurs or when a module is being initialized or destroyed upon the start-up or shutdown of the server.

Module Interface

Modules must implement the module interface so the core knows what functions from the module can be used and where and what parameters are available.

The module interface definition consists of three arrays which describe a list of available functions, parameters, and callbacks.

`cmd_export_t` array describes available functions. Each function is described by a structure that contains:

- Name of the function.
- Pointer to the function.
- Number of parameters
- Fix-up function. The fix-up function is used mainly to convert parameters of the function upon start-up of the server.
- A list of `route` blocks in which the function can be used.

```
/*
 * Exported functions
 */
static cmd_export_t cmds[] = {
    {"save",         save,         1, domain_fixup, REQUEST_ROUTE},
    {"save_noreply", save_noreply, 1, domain_fixup, REQUEST_ROUTE},
    {"lookup",       lookup,       1, domain_fixup, REQUEST_ROUTE},
    {0, 0, 0, 0, 0}
};
```

The module in our example exports three functions: save, save_noreply, and lookup. The very last element of the array must must contain fields with zero values. The element servers as a block for the core because it doesn't know number of elements in the array.

The second array is very similar to the first one. It contains structures describing available parameters and their type. The structure describing a parameter contains the following fields:

- Name of the parameter.
- Type of the parameter. It can be either `INT_PARAM` for integer parameters or `STR_PARAM` for string parameters.
- Pointer to the parameter variable.

```
/*
 * Exported parameters
 */
static param_export_t params[] = {
    {"default_expires", INT_PARAM, &default_expires},
    {"default_q",       INT_PARAM, &default_q      },
    {"append_branches", INT_PARAM, &append_branches},
    {"use_domain",      INT_PARAM, &use_domain     },
    {"case_sensitive",  INT_PARAM, &case_sensitive },
    {"desc_time_order", INT_PARAM, &desc_time_order},
    {0, 0, 0}
};
```

Our example shows a module exporting six parameters, all of them are integer parameters. This array must be also terminated by an element containing fields set to zero.

The last structure must have name `exports`. This structure is looked up by the core when it is loading the module. The `exports` structure contains pointers to the previous two arrays so that the core can find them easily and pointers to some other callback functions that the module can use.

```
/*
 * Module exports structure
 */
struct module_exports exports = {
    "registrar",
    cmds,       /* Exported functions */
    params,     /* Exported parameters */
    mod_init,   /* module initialization function */
    0,          /* destroy function */
    0,          /* oncancel function */
    0           /* Per-child init function */
};
```

The structure contains the following fields:

- Name of the module (registrar in our example)
- Name of array describing exported functions
- Name of array describing exported parameters
- Name of the module initialization function. This function is called once when the module is loaded.
- Destroy function. The function is called when the server shuts down.
- On-cancel callback function.
- Per-child initialization function. This function is called after the main process forks children. It is called in each child separately.

A module can provide zero value if it doesn't need any of the callback functions.

3.2.4 SIP Message Translations

SIP message being processed is stored in an array. The contents of the array must not be modified because subsequent functions want to see the unmodified message that was received over the network.

Because the message must be modified, we have to postpone the modification until the message is being sent out. Instead of modifying the message, we will create a structure that will describe modifications made to the message. The structure will be attached to the message and translated into real modifications when the server builds the message that will be forwarded.

Figure 3.2 gives a brief overview of the modification structure. In this example we delete the whole `Route` header field and add a new `Record-Route` field at the beginning of the message.

`sip_msg` structure which represents the parsed message contains a pointer to linked list of so called *data lumps*. A data lump is a basic building block which describes a modification that should be done to the message.

In our example there are three different lumps. First is `anchor`. The anchor specifies at which place in the message the modification should be done. It contains offset of the modification calculated from the beginning of the message. Each anchor has two pointers—before and after. `Before` is a linked list of data lumps that should be put before the anchor and `after` points to a linked list of modifications that should be put after the anchor. This approach allows us to insert a new lump between any two existing lumps.

Anchor is a lump that only marks a place in the message that should be modified. This type of lump carries offset only, it doesn't say anything about the modification that should be made.

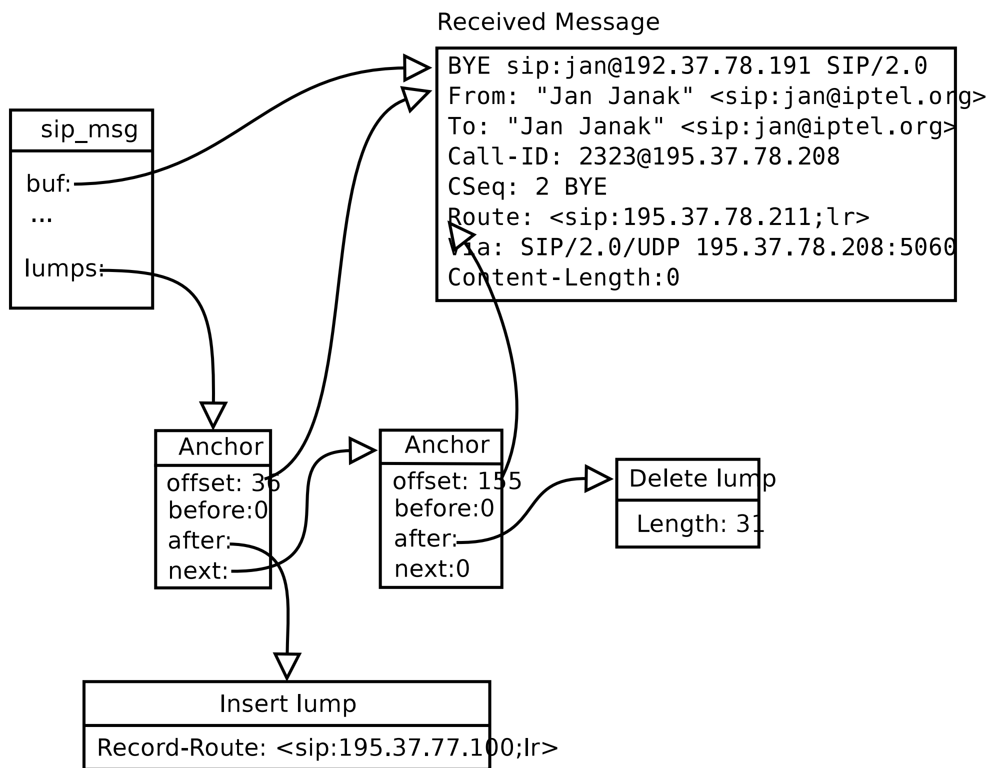

Figure 3.2: Data Lumps

There are several additional data lump types for that. The most important are `insert lump` and `delete lump`.

Insert lump inserts a piece of text before or after the anchor. The offset is taken from the anchor, the insert lump only contains the text that should be inserted.

Delete lump deletes a piece of text before or after the anchor. The region that should be deleted is specified by the offset in the anchor lump and length field which is contained in the delete lump.

3.3 Counted Strings

Because SIP is textual protocol, a SIP server has to deal with many strings. Most of the string functions need to know length of the string. Strings in C are zero terminated so `strlen` function has to scan through the whole string to find the zero termination. This operation is O(N) where N is length of the string.

Most strings that the server processes come from the SIP message being processed. But

because the server has already parsed the strings to be able to extract them, it must know their length and there is no need to calculate it again.

Therefore, for each string for which we know its length, we store it along with the string. For that purpose we create a new structure `str`:

```
struct _str {
    char* s;
    int len;
};

typedef struct _str str;
```

Most of the strings stored this way are not zero terminated. This is necessary because they are often stored in the buffer holding the SIP message and the buffer must be not modified. This approach prevents extra copying of the strings which saves CPU cycles. Drawback of this approach is that we had to re-implement most `glibc` functions that work with zero terminated strings. Our implementation of the functions is usually faster because it can be more specialized and *in-lined*.

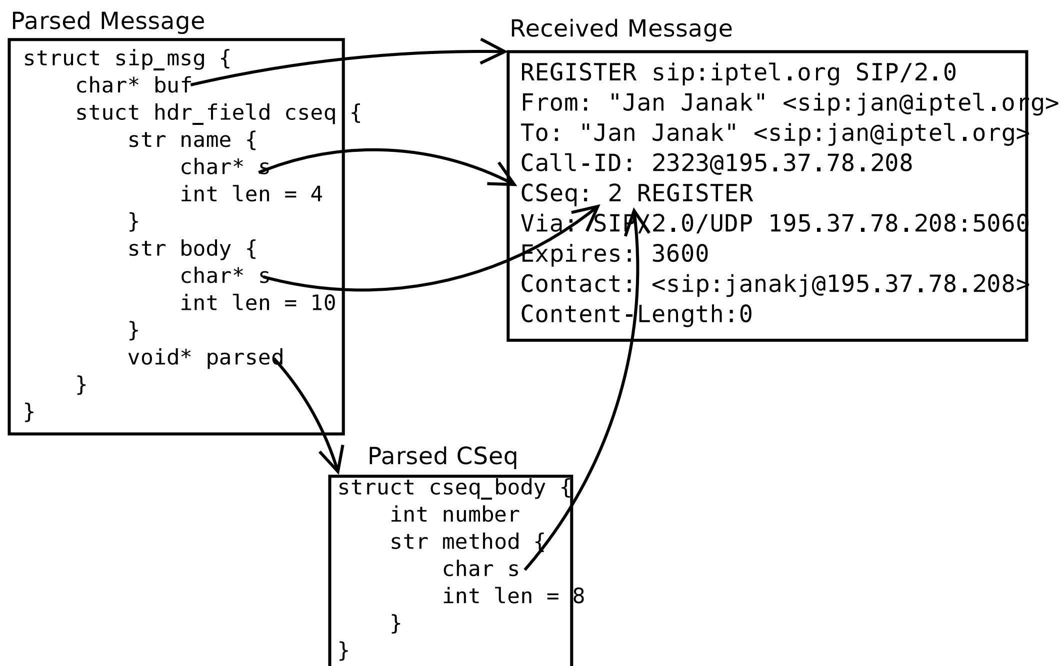

Figure 3.3: String Representation

Figure 3.3 shows how strings are stored in the parsed message. Field `buf` holds completely unmodified message as it was received. The message must not be modified because it will be later forwarded downstream or used to generate a response.

3.3.1 Performance Evaluation

Counted string have been implemented in the SIP Express Router from the beginning, thus it is not possible to compare performance of a server using zero-terminated strings and server using counted strings—to many functions depend on counted strings.

But as a side effect we re-implemented many string-related functions from the standard C library. This re-implementation gives us some performance improvement too, because our functions are simpler and can be in-lined—that eliminates the cost of a function call.

We tried to profile the server and found out that re-implementation of the system string functions like strchr, snprintf and others and using counted strings boosts performance from 5% to 10%, depending on the configuration and message type. The newly available CPU cycles are then consumed by other time-consuming functions, mainly the message parser.

3.4 Record Routing

The decision whether subsequent requests within a dialog should be sent to a SIP server is up to the server. Each server can express it's interest in the subsequent requests by adding `Record-Route` containing address of the server into the first message of a dialog. That way proxy servers can limit rate of messages they will see. For instance, a core SIP server may decide to process INVITE requests only because this is the only request that needs to be routed by the proxy server. On the other hand SIP servers controlling firewalls or NAT boxes will be probably interested in all requests within a dialog because they need to close previously opened pinholes when a session is being torn down.

Record routing has been available since RFC 2543 [6], but the algorithm described in the RFC was not the best one. Each proxy had to take the topmost `Route` header field, put it into the Request-URI and forward the message. It is necessary to save the original value of the Request-URI as the last `Route` header field because it gets overwritten.

The algorithm described in the previous paragraph has some drawbacks. First, it rewrites the Request-URI. Second, it saves the original Request-URI as the last `Route` header field, therefore the last `Route` header field contains URI which is not a proxy server along the path but the remote party.

To overcome the shortcoming, a new record route algorithm was proposed in RFC 3261 [12]. The new algorithm is significantly simpler and is similar to the source IP routing. Each SIP server

checks the topmost Route header field and removes it if it belong to the server. The request is then forwarded to the next Route header field if present or to the Request-URI.

As we can see this algorithm is very simple and straightforward, but because the new specification has to be backwards compatible with RFC 2543, it gets complicated again. A SIP server receiving a request should first check if the Request-URI contains URI generated by the server. If so then previous hop was a strict router (a server conforming to RFC 2543) and the proxy should put the last Route header field into the Request-URI. Otherwise the previous hop was a loose router (a server conforming to RFC 3261) and the proxy should proceed as specified in RFC 3261.

The server has also to check if the next hop is a strict or loose router. If the next hop is a strict router then the server must save the value of the Request URI as the last Route header field and put the topmost Route URI into the Request-URI. Otherwise the server simply forwards the request to the next hop.

Implementation of this algorithm is not very efficient. To find out if a URI belongs to a server, the server must compare hostname part and port with all hostnames, IP addresses and ports that belong to the server. Moreover, the server has to do the comparison twice (Request-URI and topmost Route) if it has inserted one Record-Record header field and three times if it has inserted two Record-Route header fields—there are certain situations in which a SIP server needs to insert two Record-Route header fields, for instance when the request is crossing transport protocol boundary. So, in the worst case scenario, a server has to compare three URIs to all local IP addresses, hostnames and ports.

3.4.1 r2 Parameter

When a SIP server inserts two Record-Route header fields, it could mark both of them so that later when the the server is processing Route header fields it could check if the header field is marked and remove also the following one if so.

To mark the header fields we append a new parameter—r2. The parameter must be added to both header fields because their order can be swapped so that the second will be processed first.

So a server marking Record-Route header fields will insert something like this

```
Record-Route: <sip:195.37.78.191;lr;r2>
Record-Route: <sip:195.37.78.191;lr;transport=TCP;r2>
```

Both header fields contain r2 parameter and when the server later identifies the URI as it's own and the URI contains r2 parameter then it will remove both of them and doesn't need to parse and compare the other one.

3.4.2 Fast Host Comparison

We have eliminated one Route URI comparison, but there are still two left, the server still have to compare the Request-URI and the topmost Route header field.

Comparison of the remaining two URIs cannot be eliminated, but we could, at least, try to make it faster. To avoid comparison of hostname and port parts of the URI with all local IP addresses, hostnames and ports, we can create a string and presence of the string in the URI will mark that it belongs to the server and no further comparison is necessary. The string can be put into username part of the URI which is normally not used and it will be easy to find there.

When a server receives a request containing

```
Route: <sip:1234abc@195.37.78.191:5060;lr>
```

It notices the presence of the "1234abc" string (which was generated by the server) and doesn't compare the URI anymore because it knows that it was generated by the server.

The generated string must be unique among servers. One of IP addresses of the server can be used for that purpose because IP addresses are unique. A SIP server can listen on several IP addresses and ports if it is running on a multi-homed host. In this case one of the IP addresses must be selected. Note that the selected address can be different from the real IP address that is put into the hostname part of the URI on a multi-homed host.

To make the comparison faster we will encode the IP address as a hexadecimal number and port number will be appended at the tail of the string.

Header field

```
Record-Route: <sip:195.37.78.191:5060;lr>
```

will then then look like this

```
Record-Route: <sip:bf4e25c35060@195.37.78.191:5060;lr>
```

One could argue that the string for the comparison is not shorter than the IP address itself. But a typical SIP server on a multi-homed host can have several IP addresses and each of the IP addresses can have associated several hostnames. In the worst case scenario it is necessary to compare all of them.

The described comparison is faster but it is not reliable. It is not guaranteed that a server will receive the same string it put into Record-Route also in a Route header. All SIP user agents are required to pass the content of the headers unchanged, but a broken user agent can remove the string.

Another problem is that a Route header field can be generated by a user agent when it is configured with a pre-loaded route set. In this case the Route header field will not contain the encoded string and it is necessary to perform the slow comparison.

Therefore the server first tries to compare the encoded string if it is present and if the string is not present or the comparison fails it tries to compare in the old way. Our experience shows that situations in which the server has to fall back to the slow comparison are rare.

3.4.3 Performance Evaluation

All the described enhancement have been implemented into `rr` module of the server. The module contains support for record routing. The interface of the module contains two functions: `record_route` and `loose_route`.

`record_route` function only inserts a `Record-Route` header field at the beginning of a SIP message. The text being inserted is pre-calculated when the server starts.

`loose_route` function is much more interesting because the function contains all the described improvements and also very complicated logic. To eliminate influence of the rest of the server code, we will measure delay caused by `loose_route` function. The measurement will be repeated 10000 times, extreme values will be removed from the sample set because they are usually cause by some CPU-intensive task of the operating system.

Because the logic of the function is complex, we will measure the following cases:

- SIP message containing no `Route` header field.
- SIP message containing just one `Route` header field of the server.
- SIP message containing two `Route` header fields, both belonging to the server.
- SIP message containing one `Route` header field examined by multi-homed server listening on two interfaces with the same port.

Each situation will be measured twice, once with `r2` parameter and fast host comparison enabled and second time with the improvements disabled.

There is no difference in speed when the message contains no `Route` header field. In that case the original function without the modifications is as fast as the new function. This is logical because the improvements work only when a `Route` header field is being processed.

When there is just one `Route` header field in the message, the original function is slightly faster then the new one. This is because the new function looks for the `r2` parameter which requires additional parsing of the header field. The old function doesn't know about `r2` parameter and ignores it.

When there are two `Route` header fields, the new function is approximately 1.5 times faster. The reason is that the old function has to perform two comparisons. The new functions performs just one comparison, it doesn't perform the second comparison because it finds `r2` parameter.

And in the last case, the new function is approximately 2.5 times faster than the original one. In this case both improvements will be used. The original function has to perform several different IP address comparisons because the proxy is multi-homed. The new function performs just one comparison because it can use the pre-parsed value stored in username of `Route` header fields.

Because there are cases in which the improvements do not help, we have made them configurable. An administrator can switch the improvements off if he knows that in his case it will not help.

3.5 The Message Parser

Message parsing is one of the most complex operations a SIP server has to perform. SIP messages are sent in text and the grammar of the messages is very complex. SIP Messages can also be very long—that is the reason why RFC 3261 mandates support of TCP as transport protocol.

Profiling of SIP Express Router shows that the message parser consumes from 10% to 40%, depending on the configuration of the server. A server with complex configuration script will not benefit from optimizations of the message parser because most of the time is spent elsewhere, the server has, for instance, to wait for a database to perform a query, calculate digest authentication parameters, perform radius accounting and authentication over network and so on. All the mentioned operations are extremely slow compared to message parsing and thus, the contribution of the message parser optimizations to the overall server performance is minimal.

On the other hand a SIP server with very simple configuration can benefit from the message parser optimizations much more because the message parsing will consume significant amount of time of the server.

We believe that a core SIP server will have simple configuration only. Typically all complex operations like authentication and authorization will be performed by outbound proxies. Outbound proxies usually serve smaller population of users and therefore performance of outbound proxies is not very important. Such proxies are usually feature rich. A core SIP server, on the contrary, has to serve a large population of users and will perform tasks that are not so time consuming and that are necessary to keep the SIP network running. Because of the large population of users such a proxy will have to be optimized for performance—that includes the message parser.

SIP Express Router is a configurable SIP server that can act as both. It can be either configured as outbound proxy containing many features or it can be configured as a very fast core SIP proxy. We did several message parser optimizations that are described in the following sections.

3.5.1 Hand-crafted Parser Versus Yacc

Most of parsers are auto-generated these days. Utilities like `bison` or `yacc` can generate an LR parser almost automatically. Using the utilities is very easy, a programmer should only supply a text file containing a description of the grammar and the utility will generate appropriate parser in C language. It is easy, fast, and reliable. Generated parser will contain no bugs if the input grammar is correct. On the other hand, auto-generated parsers tend to be big and slow. It is also hard to generate a lazy parser using `bison` or `yacc`.

A hand crafted parser takes more time to implement but can be made more efficient and faster then auto-generated parsers. It is possible to implement many performance improvements because we have some additional information that the generated parser doesn't have, it includes:

- *Target Platform*—We know on which platform the server will be running and we know the limitations of the platform.
- *Compiler Used*—Every compiler has it's strengths and weaknesses. We can design the parser to eliminate weaknesses of the compiler used.
- *Application Behavior*—It is possible to predict how the application will be using the parser and optimize it for a particular application (our server in this case).
- *Typical Message*—We can analyze existing SIP traffic from iptel.org and optimize the parser to be as fast as possible for the real traffic.

The SIP grammar is specified in RFC 3261. The grammar is written in ABNF (Augmented Backus-Naur Form). It is not complete—it contains references to other documents (mostly RFCs) that contain the rest of the grammar. It also contains some syntactical ambiguities which are probably result of the references.

Because we aim to make the server as fast as possible, we proceed with a hand-crafted parser. Indeed it will require much more man-power to write such a parser because the grammar is quite complex, but on the other hand it will be possible to do further performance improvements and also we don't have to write a complete parser. Many header fields are not processed by a SIP server and we can implement what is truly required.

3.5.2 Lazy Parsing

The basic method how to speed up the parsing is to parse only required header fields of the message, we call this approach *lazy parsing*.

A SIP server usually doesn't need body of the message, it is interested only in the message header so we don't parse message body (usually a document containing an SDP description of

media streams). When a SIP server finds an empty line delimiting message header from message body it stops parsing and further calls to the parser function return immediately without any action.

Because the core of the server doesn't know in advance which header fields will be needed and what detail of parsing will be requested, the modules will be responsible to call the parser whenever they need a part of the message that hasn't been parsed yet. The core parses only the very basic header fields which will be needed in almost any configuration of the server or which are needed by the server core itself.

Those always parsed header fields include `Via` and `To` header fields. Both header fields are used to generate responses by the core itself so the core parses them by default. The server must add `tag` parameter to `To` header field when generating a response and the topmost `Via` header field is used to route the reply back to the sender of the request. Other header fields are not parsed by default.

A module interested in a header field which hasn't been parsed yet calls `parse_header` function. The function takes three parameters:

- Pointer to the `sip_msg` structure.
- Header field identifiers that should be parsed.
- Flag telling if only the first occurrence of the header field should be returned or if the module is interested also in other occurrences.

The function will return immediately if the header field is already parsed and only the first occurrence was requested. If the caller specifies (using the third parameter) interest also in subsequent occurrences, then the function tries to find them and it stops parsing when it finds next occurrence or end of header (empty line).

A flag variable is kept in the parsed structure so it does know which header fields are already parsed. When called, the function first checks the flags variable and returns immediately if all requested header fields are already parsed.

Structures representing parsed header fields are stored in a linked list. That allows sequential processing of the header fields. In addition to that the `sip_msg` structure also contains series of variables called `hooks`. Hooks are variables that contain pointer to the first occurrence of a header field if the header field is already parsed. If it is not parsed then it contains null pointer. The hooks are used very often because most modules are interested only in the topmost occurrence of a header field. Figure 3.4 shows an overview of a parsed message structure.

`parse_headers` function does parse only header fields, nothing more. That means it can determine type of a header field from the name but it doesn't parse body of the header field. The

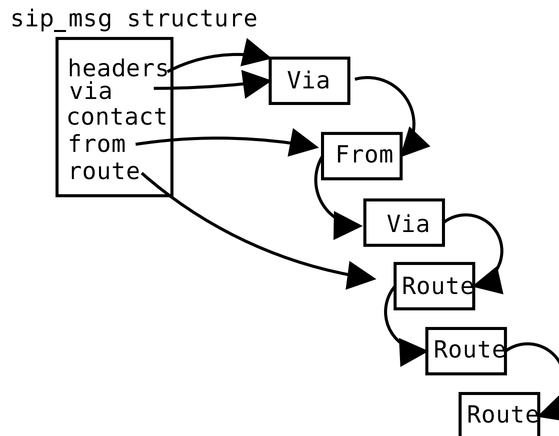

Figure 3.4: Lazy Parsing

structure describing the parsed header field only contains name of the header field, it's type and body of the header field as text.

This is another speed improvement, we have found that there are many situations in which a module needs to know just the whole body and is not interested in more details. Some header field bodies can be pretty complex and we will parse them only if a module really needs it.

3.5.3 Incremental Parsing

Another set of functions is necessary to parse header bodies on request. Those functions create additional data structures containing the parsed header field body and store it into the `parsed` field of the `hdr_field` structure. We call this technique *incremental parsing*. Incremental because at first only the header field is parsed and content of the header field is parsed only on request.

Some header fields are complex and some are simple. A common characteristics of complex header fields is that they can contain one or more SIP URIs. Examples of complex header fields are `From`, `To`, `Contact`. Because SIP URI are complex too and thus parsing of the URI is complex, we do not parse them by default. Again, modules interested in SIP URIs are supposed to parse them by themselves. There is a large number of modules that need URI as it is and are not interested in respective URI fields like username or hostname. Figure 3.5 depicts overview of incremental parsing.

As we can see a module first calls `parse_headers` function which will create `hdr_field` structure which contains only type, name, and body. To further parser the header fields the module identifies that it is a `From` header and calls `parse_from` function. If the module needs

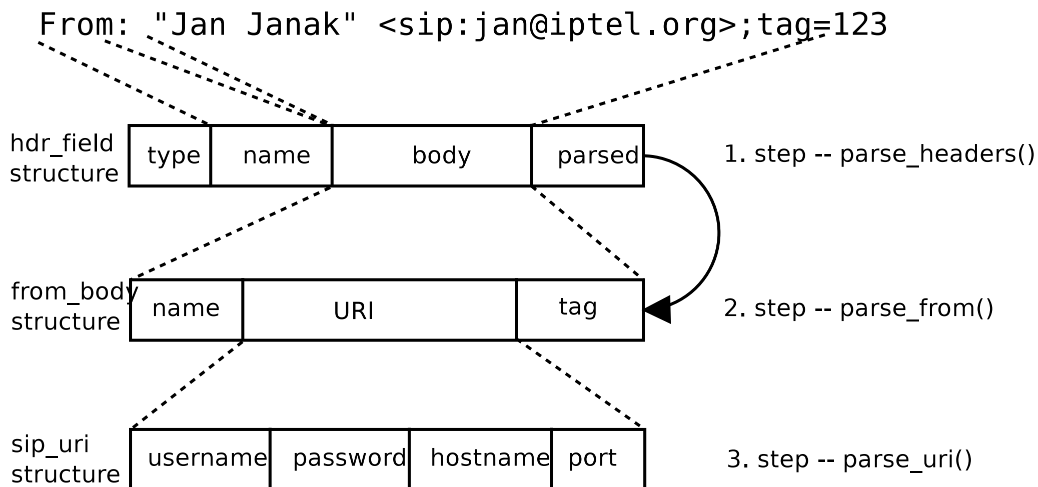

Figure 3.5: Incremental Parsing

just username part of the URI, it calls the third function which is `parse_uri`.

Next time, when another module needs to parse the same header field, it doesn't have to call any of the three functions because the header fields is already parsed and the parsed structures are stored. This prevents repeated parsing of the same data.

3.5.4 32-bit Header Field Parser

We have mentioned `parse_headers` function which can recognize type of a header field. To recognize the type of a header field it must parse it's name part. As this operation will be performed for any header fields in a message, we shall do it fast.

A naive approach would be to compare each header field name with all recognized names. The following algorithm shows case for three recognized header, `From`, `To`, and `Via`:

```
name = lowercase(name);

if (name == "from") {
    type = HDR_FROM;
} else if (name == "to") {
    type = HDR_TO;
} else if (name == "via") {
    type = HDR_VIA;
} else {
```

```
    type = HDR_OTHER;
}
```

This is very slow and inefficient, but we have seen some implementations doing that. More advanced approach is to design and use a finite state machine for that purpose. Figure 3.6 depicts the automaton.

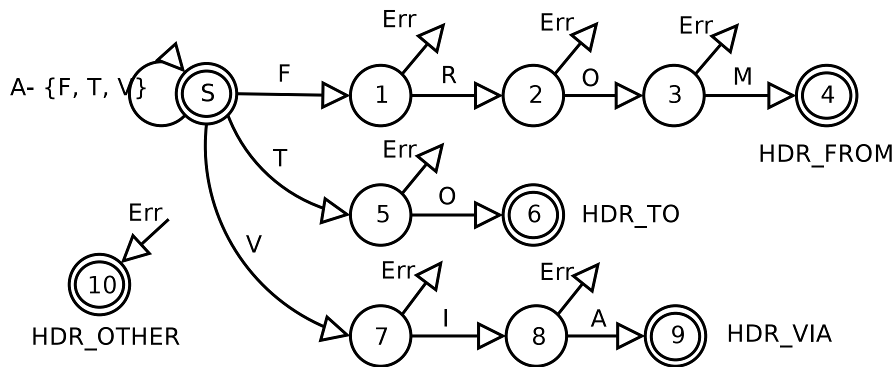

Figure 3.6: 8-bit Parsing Automaton

The automaton is not complete but it clearly shows the idea. The parser takes the characters one by one and compares them. If they match, the automaton will transit to another state until it reaches a final state.

This is faster the the naive approach, but it can still be made faster. We can use the following information to re-design the automaton:

- The parser will be running on a 32-bit architecture.
- A 32-bit or higher CPU can compare four bytes in one cycle.
- Most of the header fields are longer than four bytes.
- Only limited set of characters is used in header fields.

Given the information above, we can re-design the automaton to be 32-bit. That means the automaton will read and compare four bytes in one step instead of one. Header fields shorter than four bytes or remainders of longer header fields need to be compared again byte by byte. Figure 3.7 shows the basic idea.

As we can see, the parser reads four bytes from the memory and converts them to an integer. The integer is then compared at once—using one instruction because we are on a 32-bit architecture. This approach works very well especially for long header fields. In this case it is almost four times faster than the corresponding 8-bit automation.

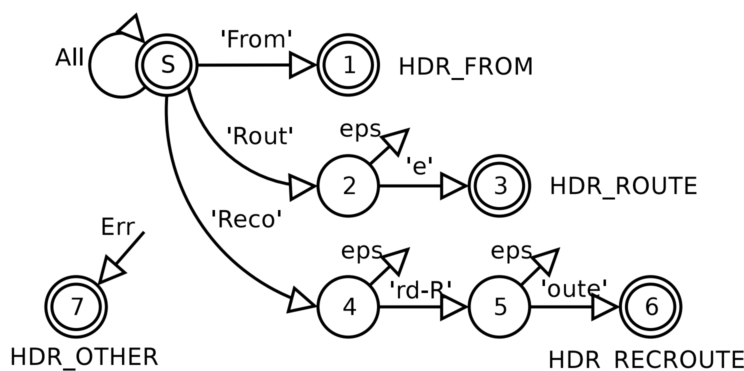

Figure 3.7: 32-bit Parsing Automaton

Header fields shorter than four bytes must be compared byte by byte again. This is one reason why the 32-bit automaton is not four times faster. Our measurements show that it is 2–3.5 times faster, depending on the SIP message header being parsed.

Hash Table Based Character Conversion

Because the comparison of the header fields must be case insensitive, we have to convert the bytes to lower or upper case first and then compare only with lowercase or uppercase patterns.

The first approach how to convert it to lower case without degrading performance was to create a hash table that will convert all four bytes at once. The table must be collision less to be fast, otherwise it would degrade the speed of the parser.

Another problem was how to generate keys and values for the hash table. We implemented a special generator that generated a C header file containing keys and values for all parsed header fields and also calculated size of the hash table so it was collision-less. The algorithm for calculating the size was very simple—brute force. It tried to create a hash table with a given size and tried to insert all the keys into the table. If there was a collision then the algorithm increased size of the table by one and tried again. That was repeated until a collision-less hash table was found.

This approach worked quite well, the case conversion was four times faster than the functions of the standard library because they convert byte by byte, but big drawback of this approach was that a special utility was needed to extend the table and it was very easy to do a mistake.

It is known that accessing the memory is significantly slower than the arithmetical operations. We also performed analysis of the SIP traffic going though our public SIP server and found out, that almost all SIP messages respect the letter case specified in the SIP specification. For example, the specification contains `Record-Route` and almost all SIP message contain the same letter case. There are hardly any messages containing, for instance, `record-route`, `RECORD-ROUTE`

or rEcOrD-rOuTe.

We can use this observation and design the parser to be even faster for this case. In this particular example, the parser first tries to find out if the header field is Record-Route, including the letter case. If not, than the compared header field is converted into the lower case and compared against record-route.

The next example in pseudo-C language gives a simplified overview of the algorithm.

```
quaternion = lower(read_four_bytes(name));
switch(quaternion) {
    case 'from':
        result = HDR_FROM;
        return;

    case 'rout':
        next = get_next_byte(name);
        if (next == 'e' ||  next == 'E') {
             result = HDR_FROM;
             return;
        }

    case 'date':
        result = HDR_DATE;
        return;
}
```

It is clear that the algorithm will be fast for messages respecting the letter case specified in the RFC and will be slow for all other messages. But as we have observed there are hardly any user agents constructing messages containing a different case of the header field names and therefore this algorithm improves speed of the parser.

Eliminating the Hash Table

Although the hash table described in the previous section was fast and worked well, it had some other drawbacks. It is hard to construct such hash table. Therefore we created a special utility which can calculate appropriate keys for all header fields to be parsed and also calculates size of the hash table so it is collision-less. But it is still not easy to use the utility and it is very easy to do a mistake. Therefore we tried to find a simpler solution.

A header field name can contain any character that can be encoded into UTF-8. That is the theory. Practically only a very limited set of characters is being used and only private extensions to SIP can potentially use the rest of the character set.

Because we need to parse the standard header fields only, the character set will be limited. Standard header fields use only the following characters:

- Lowercase alphabetical characters, 'a'–'z'.
- Uppercase alphabetical characters, 'A'–'Z'.
- Numbers, '1'–'9'.
- Delimiters, '-', ' ', ':'

All the mentioned characters can be converted to lowercase by resetting the sixth bit. Therefore one byte can be converted to lowercase using `(character | 0x20)` and four bytes can be converted to lowercase using `(four_bytes | 0x20202020)`.

This approach is significantly easier than constructing and using the hash table. We have replaced the hash table with this conversion because characters in header field names that we need to recognize can be converted this way.

We also convert all characters to lowercase before parsing and thus it doesn't matter if the parsed message contains `Record-Route` or `rEcOrD-rOuTe`. We believe that this is a significant improvement because it is much easier to implement than the hash table and also it is very fast.

Portable Memory Access

We have mentioned that the parser needs to read four bytes from the memory and compares them in one step. There are two ways how to read four bytes from the memory. We can either re-type the string buffer to integer and read them at once. The following example shows this approach.

```
char* message; /* Buffer containing the parsed message */
char* ptr;
int* num;

ptr = get_next_header(message);  /* Find next header field */
num = (int *)ptr;
```

Variable `num` will contain pointer to a 32-bit integer. The integer will be made from first four bytes of the next header field. Problem of this approach is that it is not portable. The mentioned code would work on Intel CPUs, but it will not work on SPARC, for example.

The problem is that the memory address is not aligned to 32-bits. Intel CPUs allow reading that way from un-aligned memory but the read is very slow. Running the same code on SPARC will crash with SIGBUS signal. SPARC doesn't not allow to read data from un-aligned memory.

We have to find another approach because we want the server to be portable. That means we have to read the number from the memory byte by byte. The following example will be portable.

```
char* message; /* Buffer containing the parsed message */
char* ptr;
int num;

#define READ(val) (*(val+0)+(*(val+1)<<8)+(*(val+2)<<16)+(*(val+3)<<24))

ptr = get_next_header(message); /* Find next header field */
num = READ(ptr);
```

It is interesting to compare speed of both examples on an Intel CPU (which allows both). Our measurements shows that there is no difference between reading four bytes in one step or reading byte by byte, reading byte by byte is as fast as reading four bytes together.

This only proves how extremely slow reading from unaligned memory is and how important it is that the compiler will align the data in the memory. Intel CPUs are a bit programmer-friendly and allow such reading even if it is slow.

3.5.5 Performance Evaluation

To evaluate performance of the message parser, it is necessary to find a message that will represent the SIP traffic. We analyzed the traffic from our public SIP proxy server and found out that more than 90% of users use Microsoft's MSN Messenger. Messenger generates two types of messages, INVITE and REGISTER. We will take the INVITE message as the representative sample.

32-bit Header Name Parser

Performance improvement of the 32-bit header name parser depends on the message being parsed. The worst case scenario is a SIP message containing only short forms of header field names (all important header fields have assigned a short form which consists of one character followed by :). In this case the 32-bit parser is as fast as the corresponding 8-bit automaton.

If the message contains many long header field names then the 32-bit header field name parser is approximately 3.5 times faster than the corresponding 8-bit automaton.

In case of the INVITE message generated by MS Messenger the 32-bit automaton is approximately 2.8 times faster.

We always parsed the same message 10 000 times and then calculated average time needed for the parsing and found out how many times is the 32-bit parser faster.

When profiling the server with 8-bit parser, the parser function was among top CPU-consuming functions. The 32-bit version is much more efficient because it the function is not among top CPU-consuming functions anymore.

Unfortunately this performance improvement affected the overall performance of the server less than we expected because the name parser consumes about 10% of time spent in the message parser.

Lazy Parsing

Lazy parsing is one of the most promising optimization techniques, especially for server with simple configuration. In this case parsing was one of the most CPU-consuming operations. Especially parsing of complex header fields containing a set of URIs seems to kill the performance.

Complete parsing of an INVITE message is approximately 6 times slower than parsing just header fields that are needed for stateful forwarding. Lazy parsing of a REGISTER message is in our case about 2.5 times faster than the complete parsing. The difference is caused by scattered header fields in the REGISTER message. Registrar has to find all `Contact` header fields and therefore must parse all the header fields in the message.

3.6 Registrar and User Location

To learn current location of users, the SIP server uses a location database which maps SIP URIs to current locations of users (more on this in section 2.2.3). REGISTER messages are usually used to construct location database entries.

We have implemented two modules—registrar and usrloc, which can turn the SIP server into a fully featured registrar.

Registrar module parses incoming SIP messages, extracts information that is stored into the location database and sends replies back.

Usrloc module implements the location database and exports functions that other modules can use to access the content of the location database.

Theoretically both the modules can be merged together but there are also some other modules that need to access the location database and thus it is beneficial to have a common API for accessing the database (for example Presence Agent needs to access the database).

3.6.1 Persistent Storage Cache

The data that is stored into the location database persists for some time. The lifespan is given by the `Expires` value received in the REGISTER message or it can be limited by the registrar. During this period of time a registered user is reachable through the server because the server knows his current location—IP address and port on which is the user reachable.

The contents of the location database must be stored persistently. This is one of the basic requirements. When the server crashes or is restarted, previously stored data must be retrieved from the database. Otherwise all previously registered users will be unavailable until they send a new registration.

The registrar uses MySQL database to achieve the persistence. The same database is used by the proxy server when it needs to route a request to a user. In that case it will retrieve the information previously stored by registrar and use the information when routing the request.

This approach has one big drawback—accessing the database is extremely slow. One could argue that MySQL server is one of the fastest database servers available. That is, maybe, true, but compared to the speed of the processing logic of the server it is slow.

The registrar has to access the database each time it receives a valid register request and the proxy server has to access the database each time it receives a request that is targeted to one of users the proxy is responsible for. Given the facts it is clear that the database could easily become the bottleneck of the whole system.

Moreover, the database is typically not running on the same host as the registrar or proxy server. Companies are usually running the database server on a dedicated host and the server is usually used by some other applications or servers too. It means that each SIP message that triggers some operation accessing the database would by translated into a command that will be sent to the database server over a TCP connection and the server must wait for the reply.

To overcome this problem we will implement a memory cache that will speed up the database operations.

Each registrar performs two basic operations:

- Contact lookup.
- Location storage.

Design

The heart of the cache is a hash table. Because the hash table will be not collision-less, each entry in the hash table contains a linked list of all entries that were hashed into the same field in the table. We call the hash table cells *collision slots*.

To allow sequential scanning of all entries in the hash table, all entries are also arranged into a doubly-linked list. This architecture allows us to implement fast lookup of entries based on their keys and also traverse all the entries sequentially, which is needed to perform some maintenance tasks.

Objects stored in the cache are called *records*. Each record in the cache consists of so called *Address of Record* which uniquely identifies every record and is used as key for hashing. Each record can have one or more contacts, which are again arranged into a linked list. Each contact in the linked list represents one contact address of the user represented by the address of record.

Each contact has assigned expiration value. When all contacts expires or are removed then the whole record is removed from the cache.

Figure 3.8 gives an overview of the cache design.

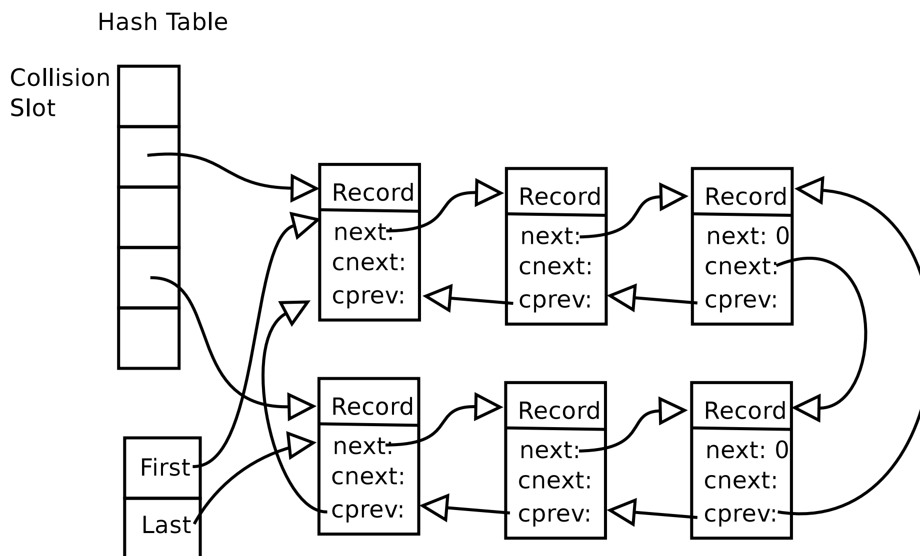

Figure 3.8: Registrar Cache

Basic operations performed on the cache are:

- Insert new record
- Delete record
- Update contact of a record
- Add contact to the record
- Delete contact of a record

- Flush the cache

First five operations are performed when a REGISTER request comes. The last operation is performed from a timer which periodically maintains the cache contents.

The whole cache is stored in the shared memory. This is necessary because subsequent REGISTER messages can by processed by different children of the server and all the children must have access to the structures created by other children. The data structures are protected by locks because they are shared among processes.

Clean-up timer

Each contact in each record has it's own expires value telling for how long yet is the contact valid. The contact must be removed from the cache after it expires.

Because the cache can contain many contacts, it is impossible to remove each contact at the same time when it expires. That would require too many timers (for each contact one timer) and that would generate too much overhead.

Instead of that, we will set up just one timer, which will hit periodically (the interval is configurable). The time scans the whole cache for expired contacts and removes them from the cache. It also flushes contacts that were updated into the database if it is enabled.

Drawback of this approach is that there are contacts which have already expired but haven't been removed from the cache yet. We call them *zombie contacts*. Zombie contacts must be ignored when accessing the database. Each module using the cache will get a list of contacts valid for a record, but the list may contain expired contacts which must be ignored.

3.6.2 Persistence Modes

Administrators can have different requirements on the registrar cache. For example for small installations of the server (serving only couple of users in a small company) it can be desirable not to use any database and keep the records in memory only.

But there can be situations in which it is absolutely necessary to make sure that all records are immediately stored into a database so all of them can survive crashes or restarts of the server.

To satisfy such needs, we have implemented three modes of operation of the case:

- Memory only
- Write through
- Write back

Memory only mode is the simplest one, it doesn't require any database to be available. All the records are stored in memory only and are not flushed to the database. This mode is the fastest one but records cannot survive reboots or crashes of the server.

Write through mode it the slowest one. The mode reflects all changes to the cache immediately into the database. In this case contacts will be persistent for sure (i.e. they will survive crashes and reboots.). The server will re-load all records from the database upon startup.

Write back mode is a compromise. All changed are primary made to the cache. Update of the database is done periodically by the timer. Speed and reliability of this mode is a compromise between the previous two modes. This mode is good to absorb avalanches of requests coming periodically. When updating the database it aggregates the updates so it is more efficient than write through mode. Drawback of this mode is that there is a window during which the records are in the memory and not in the database yet. If the server crashes or reboots during the window, records stored in memory only will get lost.

3.6.3 Hashing

To speed up lookup of records stored in the user location cache we use technique called `hashing`. Hashing is the fastest know technique. A perfect hash function is extremely hard to find. The basic idea in hashing is to scramble some aspects of the key and to use this partial information as the basis for searching.

Address of Record servers as the key in our case. Address of record is a string of form `username@domain`. We need to find a function that will hash such strings into the hash table and create as few conflicts as possible.

In addition to that the hashing function must be fast. Many hashing functions have been proposed to hash strings and they have something in common—all good hashing functions are simple [10].

In our case the domain part will be probably same all the time so we can ignore it when hashing. We just need to use the username part when hashing.

To create a good hashing function we need some data that will model characteristics of the real usernames that will be really used. Because SIP addresses are similar to e-mail addresses, we can use e-mail usernames to model the characteristics.

To find a good hashing function we took e-mail addresses from our mailing lists. Currently there are thousands of users subscribed. Then we tried to design a function with good distribution for usernames contained in the e-mail addresses.

One of the best and easiest functions is to take characters of the username from left to right and add them with descending priority. Such hash function gives reasonable results in terms of distribution and is simple and fast.

3.6.4 Locking Optimization

We have said that the cache is stored in the shared memory and therefore must be protected by a mutex. All operations on the cache will then be performed in a critical region which will prevent all other children from accessing the cache at the same time.

Because some of the operations need to access the database (especially flushing the cache from the timer), they can block the cache for a while.

To prevent this situation, we will divide the cache into *domains*. A domain is part of the cache which has it's own mutex to protect it. Different domains can be accessed simultaneously without blocking each other. Domains can be again partitioned, this approach allows us to keep reasonable size of domains and requests will not block each other.

The decision which domain to use can be made from the configuration script. It is possible, for instance, to put all usernames beginning with a to domain1, usernames beginning with b to domain2 and so on.

3.6.5 Performance Evaluation

To evaluate performance of the registrar, we will use REGISTER messages with different usernames and different expire values.

The measurements were done on an Intel Pentium III 850MHz CPU with 512 MB of memory connected to the generator of the messages using Fast Ethernet. The machine was completely dedicated to the server and there was on other traffic on the network.

In memory only mode the registrar was able to handle about 2000 messages per second. If more messages per seconds were generated then the registrar started to drop the messages.

In write through mode the registrar is able to handle about 1100 messages per second. This is significantly slower than the memory only mode because all operations are reflected directly to the database.

In write back mode the server is able to handle 1900 messages per second until the timer start to flush the cache. Registrar will slow down to 1100 messages per second during the cache clean-up.

3.7 Memory Management

The server uses private memory wherever possible, but some data structures must be stored in the shared memory because all children need to access it. Examples of such structures are registrar cache or transaction structures.

That means the server uses private memory and also shared memory. We needed some allocator for the shared memory segment so we re-implemented the memory allocator from the standard C library.

Our implementation is faster because it can exploit know characteristics of our server. Also we can eliminated locking from the allocator operating in the private memory segment because we do not use threads. The standard C library allocator has no way of knowing that the process doesn't use threads and therefore must use locking all the time.

Our memory allocator is also much more aggressive. It allocates bigger chunks of memory than needed because it is faster. Memory is very cheap nowadays so we can afford such wasting. A processing of a single SIP message requires more than 100 memory allocations and thus the memory allocator must be really fast, otherwise it can easily turn into the bottleneck of the application.

Our memory allocator tries to keep linked lists at the minimum because traversing of such linked lists consumes too much CPU-time.

Currently the memory allocator seems to work quite well although it allocates more memory than necessary. Further investigation will be needed to find out if it is possible to save the memory without loosing the speed.

Chapter 4

Performance Evaluation

In the previous sections we have described the architecture of the server. We have also described some individual performance improvements, their implementation, and evaluated their impact on the performance.

It is very complicated to evaluate impact of a particular improvement on the performance of the whole server, because other server subsystems affect the measurements.

In this chapter we will try to evaluate performance of the whole server with all the improvements using the sipstone [16] benchmark.

4.1 SIP Traffic Modeling

Call arrivals in PSTN networks are independent of each other. Time between arrival is a random value and therefore a Poisson process is used to model the call arrivals in PSTN networks. Call arrivals in SIP networks will be analogous to PSTN networks and therefore a Poisson process can be used to characterize SIP call arrivals too. SIP requests will arrive in random intervals and will be independent of each other.

Call holding time is an important metric in traditional PSTN networks. It is defined as time between setup and tear-down of a conversation. The call holding time is important for PSTN networks but is not important for most SIP servers. SIP servers are usually transaction-stateful which means that they do not keep state information during the whole call but only during the transactions that establish and tear-down the call.

Time between call arrival and answer is much more important for SIP servers because stateful servers maintain state during this period. Amount of memory that can be used to keep the transaction state and time between call arrival and answer will limit number of transactions that the server will be able to process concurrently (it is a very rough estimation).

Since Sipstone [16] is currently the only benchmark available, we will be using the statistics presented in the document:

- About 70% of calls are answered.
- It takes about 8.5 seconds to answer a call.
- Unanswered calls ring for 38 seconds.

Given the parameters above we can choose 20 seconds to be the average duration of an INVITE transaction. BYE transactions usually do not wait for action of a user and thus finish immediately.

4.1.1 Transport Protocol

RFC 3261 mandates that any compliant implementation must support both UDP and TCP as transport protocols. Although our server support also TCP, implemented performance improvements were mainly designed for UDP used as the transport protocol. This is given by the lack of freely available user agents supporting TCP. We have developed a testing utility implementing Sipstone benchmark and this utility currently support UDP only.

Thus, we will limit our performance testing to SIP over UDP only. The author of this thesis is aware of this shortcoming, performance of the server when TCP is used is a future work item.

When using TCP, other issues can easily pop up. A heavily loaded server will have to keep a high number of opened connections. Impact of high number of opened connection on the performance of the server will have to be evaluated and this is out of the scope of this document.

4.1.2 Authentication

Digest authentication is used as the authentication method in SIP. Authentication will decrease performance of the server. Unauthorized SIP requests will be rejected and the user agent is supposed to re-send the request including proper credentials.

Using of authentication depends on the role of the SIP server and its configuration. We believe that core SIP servers will not authenticate requests.

4.2 Performance Metrics

In order to evaluate performance of a proxy server a set of metrics needs to be established. We will use the following metric to evaluate our proxy server:

- **CPS**—Calls per Second. This is very important parameter. It shows how many calls a SIP server is able to establish and tear down within a single second. Two transactions and several SIP messages must be processed by a server when establishing and tearing down a call.
- **MPS**—Messages per second. This metric describes how many messages per second the server is able to process. This metric is good to evaluate performance of the registrar.
- **Message Latency**—This parameter is not so important unless a big latency causes retransmissions of UDP messages.

4.3 Test-bed Overview

The evaluated server was running on an Intel Pentium III 850MHz machine with 512 MB of memory available. The operating system was Linux, no other tasks were running on the machine. The machine was connected to load generators using fast Ethernet. There was no other traffic in the network.

To generate the load we wrote a utility that is able to generate load with characteristics described in the sipstone paper. We performed two kinds of tests:

- Proxy test. This test consists of INVITE followed immediately by BYE sent to the call receiver via the proxy.
- Register test. This test consists of a REGISTER message sent to registrar and followed by a 200 OK response sent by the registrar.

First, the load generator registers several thousands of users with the registrar. Then it tries to establish a session with call receiver running on another machine. Immediately after the session is established the call generator tries to establish another call and the whole test repeats.

The server was configured with the default configuration file. Authentication as well as message checks for a too big message were disabled. Record routing was enabled on the server.

When the server receives a REGISTER it stores the contact information into the location database. When the server receives an INVITE it tries to look-up corresponding contact in the location database and forwards the request.

4.4 Performance Evaluation

During the proxy test, the server was able to process about 2000 calls per second at sustainable rate. The server started to drop messages when the call generator generated more than 2000 calls

per second.

During the registrar test the server was able to register 1800 messages per second. This number is lower than the number presented in the section describing registrar because the configuration file was more complex in this case.

Chapter 5

SIP Protocol Bottlenecks

SIP is a relaxed protocol. SIP grammar is very flexible, user agents and proxy servers can create messages in many ways, they can decide whether to put IP addresses or hostnames into messages, how to arrange header fields, whether or not use line folding and so on.

Although this flexibility is one of the strongest features of SIP, it also makes it harder to write a highly optimized and powerful implementation. During implementation of the SIP Express Router, a free SIP proxy server, we tried to tune the server to be as fast as possible. Although we reached very good performance, further optimizing is not easy because some of the bottlenecks are not in the proxy but in the protocol itself. To make the implementation significantly faster the protocol itself would have to be changed.

Some of the problems are easy to overcome using optimized user agents that can generate optimized SIP messages. Some of them would require some changes in the protocol specification and that is not possible at the moment. During the implementation we have found that the writers of the specifications sometimes do not take care of implementations. Some aspects of SIP protocol are hard to implement efficiently.

Because success of each protocol is given by number of interoperable implementations, we will describe couple of protocol aspects that make the implementation harder in this chapter.

We (iptel.org) operate a public SIP Proxy Server that other users and developers can use. Users can apply for a free account and they will get a SIP address of form

```
sip:<username>@iptel.org
```

Developers often use our proxy server to test their SIP applications before they release it and are encouraged to do so. Although this may seem to generate unwanted traffic and impose higher requirements on our server, the opposite is actually true. We monitor traffic going through our proxy server and we can inform the originator of the messages if anything goes wrong. This

is beneficial for us because we see the state of the art of the industry and developers get early feedback from us which allows them to fix bugs quickly.

This public service serves also as a valuable source of the traffic characteristics which, among other things, were used in this thesis. Based on the characteristics we have put together a set of recommendations for SIP user agent developers that will allow them to write their software to be “proxy server friendly” from performance point of view. It is clear that the set of recommendations was created mainly for out proxy server. It means that other proxy servers not running the SIP Express Router will probably not benefit from them. But in our opinion some of the recommendations are worth implementing anyway because they either make the messages shorter or can make the server’s processing logic simpler.

We have experienced that many SIP application developers do not take care of SIP performance optimizations at all. This is probably due to fact that the SIP-related specifications are still being actively developed. It is often very hard to follow all the changes and updates and implement them quickly. The fact that the core SIP RFC is the longest RFC ever supports this presumption. We hope that once the protocol specifications get stabilized developers will more focus on performance and well-formed messages. We believe that performance optimizations gain focus once SIP gets mass-deployed which can actually happen in the near future because many Microsoft products distributed with MS Windows will use SIP for signalling. Also 3GPP initiative is going to use SIP in its IP based media communication in the 3rd generation mobile networks.

5.1 Header Field Ordering

SIP inherited its message header format from HTTP and RFC 822 [2]. Every SIP message header consists of several header fields, usually each line contains one header field, but header fields can also be spread over several lines.

The header field order is not significant for headers of different types. Order of header fields of the same type is important. Table 5.1 shows two messages that are equivalent.

As we can see order of many header field was changed but `Via` header fields are still in the same order. In other words if we remove all but `Via` header fields their order must not change.

Table 5.2 shows messages that are not equivalent. As you can see order of only two header fields has changed but because both header fields are of the same type the messages are different (in this particular case responses to the messages would traverse proxies listed in `Via` header fields in different order).

Because the order of different header fields is not important we can re-order the header fields so that the most important header fields (from SIP proxy point of view) will be close to the first

REGISTER sip:iptel.org SIP/2.0	REGISTER sip:iptel.org.org SIP/2.0
From: "Jan Janak" <sip:jan@iptel.org>	To: "Jan Janak" <sip:jan@iptel.org>
To: "Jan Janak" <sip:jan@iptel.org>	Via: SIP/2.0/UDP 195.37.77.100
Call-ID: dgasdfgasdf@195.37.78.208	Via: SIP/2.0/UDP 195.37.78.191
User-Agent: kphone 3.1	Expires: 3600
CSeq: 2 REGISTER	From: "Jan Janak" <sip:jan@iptel.org>
Via: SIP/2.0/UDP 195.37.77.100	CSeq: 2 REGISTER
Via: SIP/2.0/UDP 195.37.78.191	Call-ID: dgasdfgasdf@195.37.78.208
Via: SIP/2.0/UDP 195.37.78.208	Via: SIP/2.0/UDP 195.37.78.208
Expires: 3600	Contact: <sip:janakj@195.37.78.208>
Contact: <sip:janakj@195.37.78.208>	User-Agent: kphone 3.1

Table 5.1: Equivalent SIP Messages

line of the message. Which header fields are important to a proxy depends on configuration of the proxy. The proxy then doesn't need to parse the whole message header because header fields it is looking for will be found quickly and the proxy can stop parsing. This could be a significant performance improvement for proxies that implement “lazy” (see section 3.5.2) parsing, such as SIP Express Router.

5.1.1 Order of Request Headers

If a request contains `Max-Forwards` header field then it should be the first header field in the message. A proxy usually checks this header field before any other because value of zero means that the request looped and it makes no sense to process it any further, the proxy just generates 483 response and sends it to the originator of the request.

Another important header field is `Route` header. If present in a request they must be processed by any proxy routing the request. Therefore it makes sense to put them at the beginning of the message. A proxy will need up to three topmost `Route` header fields (two of them may be generated by the proxy if the proxy put two `Record-Route` header fields, for instance, to cross transport protocol boundaries, and the third will be used as next hop of the request).

If the message is REGISTER then `To`, `Contact` and `Expires` headers will be needed by the targeted registrar, so it is good to put them next.

If there is just one `Contact` URI in the message then it is better to use `expires` parameter of the `Contact` URI and not `Expires` header field because the proxy will not need to look for `Expires` header field. If there is more than one header field then it is better to use `Expires` and don't use `expires` contact URI parameters because the message will be shorter and parsing of

REGISTER sip:iptel.org SIP/2.0	REGISTER sip:iptel.org SIP/2.0
From: "Jan Janak" <sip:jan@iptel.org>	From: "Jan Janak" <sip:jan@iptel.org>
To: "Jan Janak" <sip:jan@iptel.org>	To: "Jan Janak" <sip:jan@iptel.org>
Call-ID: dgasdfgasdf@195.37.78.208	Call-ID: dgasdfgasdf@195.37.78.208
User-Agent: kphone 3.1	User-Agent: kphone 3.1
CSeq: 2 REGISTER	CSeq: 2 REGISTER
Via: SIP/2.0/UDP 195.37.77.100	Via: SIP/2.0/UDP 195.37.78.208
Via: SIP/2.0/UDP 195.37.78.191	Via: SIP/2.0/UDP 195.37.78.191
Via: SIP/2.0/UDP 195.37.78.208	Via: SIP/2.0/UDP 195.37.77.100
Expires: 3600	Expires: 3600
Contact: <sip:janakj@195.37.78.208>	Contact: <sip:janakj@195.37.78.208>

Table 5.2: Different SIP Messages

`Contact` header fields will be a little bit faster.

5.1.2 Order of Response Headers

The most important header field for any element routing a response is `Via`. This header field is used by proxies when forwarding a response or when generating a local request. Therefore it is wise to put `Via` header fields at the beginning of the message so they will be parsed first. `Via` headers are used by each SIP network element processing a response. Order of other header fields is less important because response routing logic is usually very simple and often only `Via` header fields are used.

5.2 Hostnames Versus IP Addresses

There are many places in a SIP message which contain address of a network element. Those header fields include:

- `Via`—Each header field contains one or more network elements to which a response should be forwarded.
- `Route`—Each header field contains one or more network elements to which a request should be forwarded.
- `Record-Route`—Each header field contains one or more network elements that want to stay on the path of any further SIP requests.

- `Contact`—The header field contains contact address of the network element that generated the message.

Address of a network element is typically an IP address when SIP is running on the top of IP protocol.

Moreover, SIP allows a hostname to be put in each of those header fields. Hostnames were created mainly for humans because IP addresses are hard to remember. All the mentioned header fields are processed by machines, not by humans. Therefore they should contain IP addresses and not hostnames.

A SIP network element processing a header field must perform a DNS lookup when the header field contains a hostname. This adds significant processing delay. In addition to that each hostname can resolve to several IP addresses, that can make the resolution potentially unambiguous.

On the other hand it is necessary to mention that the original specification recommended to use IP addresses, but it was changed to aid SIP traversing certain type of firewalls and NATs. Although it can help when certain NAT boxes are used, it can also harm. For example, it is impossible to do DNS query from the Linux kernel. This makes it almost impossible to write a good SIP Application Level Gateway for Linux iptables.

Therefore we recommend using of IP addresses in those header fields that are processed by machines and not by humans.

5.3 Line Folding

Long header fields can be split across several lines of the header. This is called *line folding*. Subsequent lines of a folded header fields must begin with a white space (space or tab) to be recognized as folded. The following header field

```
Route: <sip:janakj@195.37.77.101>,
       <sip:janakj@195.37.78.191>,
       <sip:janakj@195.37.77.100>
```

is an example of a folded header field. Purpose of line folding was to make the processing easier for applications using limited buffer for header lines and also make the messages easier for humans to read.

However line folding also complicates message parsers. Places at which header fields can be broken are not clearly specified so the result is that parsers must check for possible folding almost anywhere except URIs. This is a significant complication.

Line folding doesn't seem to be very useful because most messages we have seen don't use it anyway. Also special utilities like `ngrep`, `ethereal` or special message flow viewers are used to watch the SIP traffic and those utilities usually wrap too long lines.

Line folding is one of features of SIP that, we believe, deserves to be dropped and should not be used by any SIP network element.

5.4 Scattered Header Fields

As we have shown already a SIP message header can contain several header fields of the same type. Some of those header fields that can appear multiple times can be written in two ways:

- *Joined*—In this case there is just one header field of given type and this header field contains all components. An example of joined header fields follows.

```
Route: <sip:195.37.78.211;lr>, <sip:195.37.78.191;lr>, <sip:195.37.77.101;lr>
```

- *Scattered*—There are several header fields of the same type, each of them contains one component. The following example demonstrates scattered header fields.

```
Route: <sip:195.37.78.211;lr>
Route: <sip:195.37.78.191;lr>
Route: <sip:195.37.77.101;lr>
```

Scattered header fields are one of demerits of SIP. A proxy parsing a SIP message must automatically parse the whole message if it is looking for a header field that can appear multiple times, because it can be scattered down in the message header. The parsers don't know if a header field of given type is the last in the message header until they reach the end of the header.

For example a registrar processing a REGISTER request has to parse the whole message header automatically because it must extract all contacts from the message. If all the contacts were in the same header field then the registrar can simply find the first occurrence of the `Contact` header field and that can be very fast if the header field is at the beginning of the message.

5.5 Authentication

SIP uses HTTP Digest authentication according to RFC 2617 [4]. This is very simple authentication mechanism that allows server to verify that a user knows correct password. The authentication mechanism has been developed for the HTTP protocol and is widely used nowadays. Because SIP is very similar to HTTP, the same mechanism is also used here. Both

proxies and user agents can request authentication. A proxy performing authentication looks for `Proxy-Authorization` header field and if not present then it challenges the user agent using `Proxy-Authenticate`. A user agent performing authentication looks for `Authorization` header field and challenges user agent using `WWW-Authenticate` header field. `Authorization` and `Proxy-Authorization` header fields contain credentials that are used to verify authenticity of the user.

A proxy may challenge any SIP request except ACK and CANCEL. ACK and CANCEL requests cannot be challenged because ACKs must trigger no response and CANCELs must have same `CSeq` as the request being canceled—a challenge would result in retransmitted CANCEL which would have different `CSeq` number.

We have noticed that many user agents put credentials only in the request that is being challenged and do not put credentials in subsequent requests. Figure 5.1 shows this case (left message flow). A user agent sends a REGISTER which is challenged by the server. The user agent then generates credentials and sends another REGISTER containing the credentials which is accepted by the server. Later the same user agent sends an INVITE and doesn't include credentials computed previously for the REGISTER. The server again challenges the user agent and the INVITE will be re-sent.

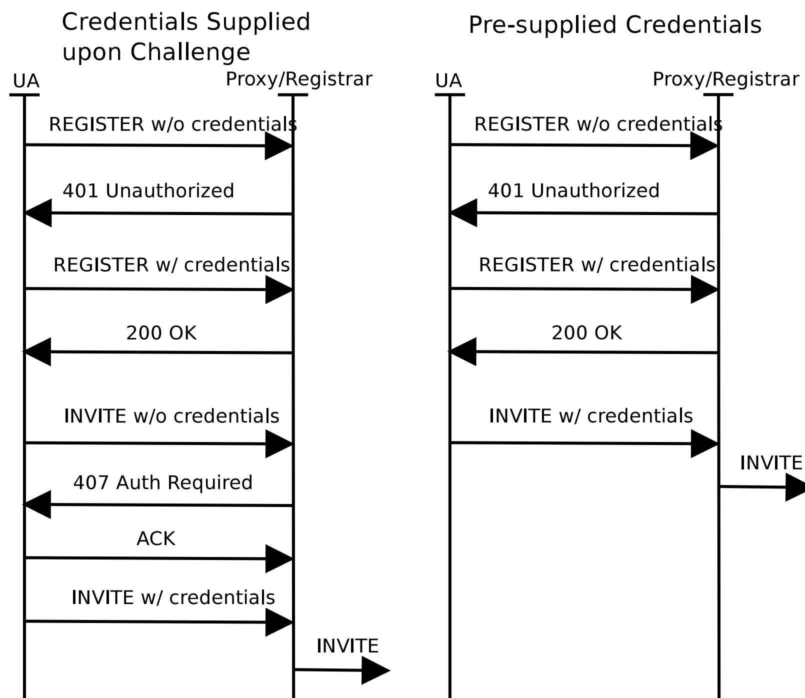

Figure 5.1: Digest Challenge Message Flows

This adds an extra delay of one RTT. To prevent this the user agent could include credentials calculated from the same username and password as for REGISTER also into INVITE request. Figure 5.1 shows call flow of such a situation (right call flow). In this case the credentials haven't expired yet so they are also accepted for the INVITE and the request is not challenged again.

It is important to mention that it is not guaranteed that the proxy will accept credentials computed this way. They can be rejected because the nonce provided by the server in the challenge has already expired or the server used the nonce to cover some header fields to protect them from modification. If this is the case then the server will challenge the user agent again and the user agent will have to recompute the credentials. In this case the user agent will have to re-send the request anyway but there is always chance the credentials will be accepted and so they should be included in all requests but ACK and CANCEL if the server has already challenged the user agent.

5.6 Display Name

Header fields containing a SIP URI can usually include also display name. The display name is usually rendered to user by user agents. It's purpose is to display something more human-readable and not just plain SIP URI.

For example if a user agent receives a message containing

```
From: "Jan Janak" <sip:jan@iptel.org>
```

it will display that Jan Janak is calling instead of

```
sip:jan@iptel.org}
```

. This makes perfect sense in `From` and `To` header fields which are usually rendered this way.

But there are also some other header fields that may contain the display name. They include `Route`, `Record-Route`, and `Contact` header fields. These header fields are processed by machines, i.e. proxy servers along the path or user agents. They are usually not rendered to users so it makes no sense to put display name in those header fields.

For example display name part of

```
Contact: "Jan Janak" <sip:jan@195.37.77.191>
```

will be not used anywhere and it only makes parsing of the harder more complicated. Instead of that use

```
Contact: <sip:jan@195.37.77.191>
```

which can be parsed much easier and faster. Note that we haven't removed the <> characters. This is because it clearly marks that the URI starts here. Otherwise it is unambiguous that `sip` is the beginning of URI and not display name (display name doesn't always have to be enclosed in quotes).

The same applies also for `Route` and `Record-Route` header fields, because they also allow using of the display name. In those header fields, however, enclosing of URI into <> is mandatory.

5.7 SIP Over TCP

SIP was designed to operate over UDP. Each UDP datagram can contain one SIP message. Such UDP messages are usually not fragmented. When a SIP network element receives a SIP message it knows how big the message is because it knows how long the corresponding UDP datagram is. We say that the underlying transport protocol provides *framing* for SIP messages.

Later TCP support was added into the new specification and the TCP support have become mandatory. It means that all SIP network elements compliant to the specification must support UDP and TCP. SIP messages can be sent over TCP streams, but because TCP doesn't provide framing for SIP messages, it is complicated to determine how big the SIP message is.

The specification mandates that SIP messages sent over TCP must include `Content-Length` header field which contains length of body of the message. So we know the length of the body but we still don't know the length of the message header. That means the implementation must perform algorithm according to figure 5.2 (left diagram) to find the end of the message header and then read `Content-Length` number of bytes to read the whole body.

To change this situation, the way how SIP messages are sent over TCP would have to be changed. A SIP network element knows how big the message being sent over TCP is—it already received the message or created the message from scratch.

As an aid to the entity receiving the message the sender could insert total length of the message at the beginning (i.e. before the first line). Existing SIP servers not supporting this extension would simply ignore it because it doesn't look like a first line of any message. Those that do understand this extension will use it to calculate how many bytes should they yet read over the TCP connection until the message is complete. The message sent over TCP would then look like this:

```
546
SIP/2.0 200 OK
Via: SIP/2.0/UDP 192.168.1.30:5060;received=66.87.48.68
From: sip:sip2@iptel.org
To: sip:sip2@iptel.org;tag=794fe65c16edfdf45da4fc39a5d2867c.b713
Call-ID: 2443936363@192.168.1.30
CSeq: 63629 REGISTER
```

```
Contact: <sip:sip2@66.87.48.68:5060;transport=udp>;q=0.00;expires=120
Server: Sip EXpress router (0.8.11pre21xrc (i386/linux))
Content-Length: 0
Warning: 392 195.37.77.101:5060 "Noisy feedback tells:
    pid=5110 req_src_ip=66.87.48.68 req_src_port=5060 in_uri=sip:iptel.org
    out_uri=sip:iptel.org via_cnt==1"
```

Where the number on the first line is the length of the message. This approach would be used only on TCP links. Figure 5.2 (right diagram) shows the algorithm for reading such messages over TCP. Compared to the previous case the new algorithm is much simpler.

It is generally a bad idea to insert the framing information into the application protocol. The framing information should be part of the transport protocol, for example SCTP.

5.8 Careful Design of SIP Networks

Ones of the aspects influencing speed and scalability of a SIP network is careful design of the network. A badly designed network can suffer from performance issues even if all elements in the network are really fast. Such network could also have serious scalability issues.

Every SIP message processed by a SIP network element costs CPU and memory (in case of stateful servers) so the basic idea is that not all SIP messages should be sent to SIP servers. In the other words—use record routing only when really needed.

In many cases a proxy needs to see only the first message in a dialog, for example INVITE or SUBSCRIBE, other messages can go directly from one user agent to another. This helps to decrease load of proxy servers and also such networks scale better, because proxy servers are usually the bottlenecks.

Of course there are situations in which record routing is necessary. Proxies performing accounting, for instance, need to see all messages within a dialog. Such proxies should be chosen carefully. They should be placed “closer” to the end points—user agents. Such proxies typically have a complex routing logic, but on the other hand these proxies usually serve smaller population of users and so the performance is not the most important aspect.

Almost all user agents can be configured to use outbound proxies. Outbound proxy is a proxy that will receive all traffic coming from the user. This is very dangerous, the outbound proxy will receive all traffic even if it doesn’t want to stay on the path. Outbound proxies should be used rarely, only when really needed. A good candidate for outbound proxy is a proxy controlling a firewall, the user will be otherwise unable to send the SIP traffic directly to the public internet.

One more thing that could improve performance of the network is using stateless SIP servers wherever possible. Usability of stateless servers is limited, but they can be used as simple load balancers, programmable message forwarders and validators. Using of stateless servers instead of stateful is recommended wherever possible, stateful servers are usually much faster.

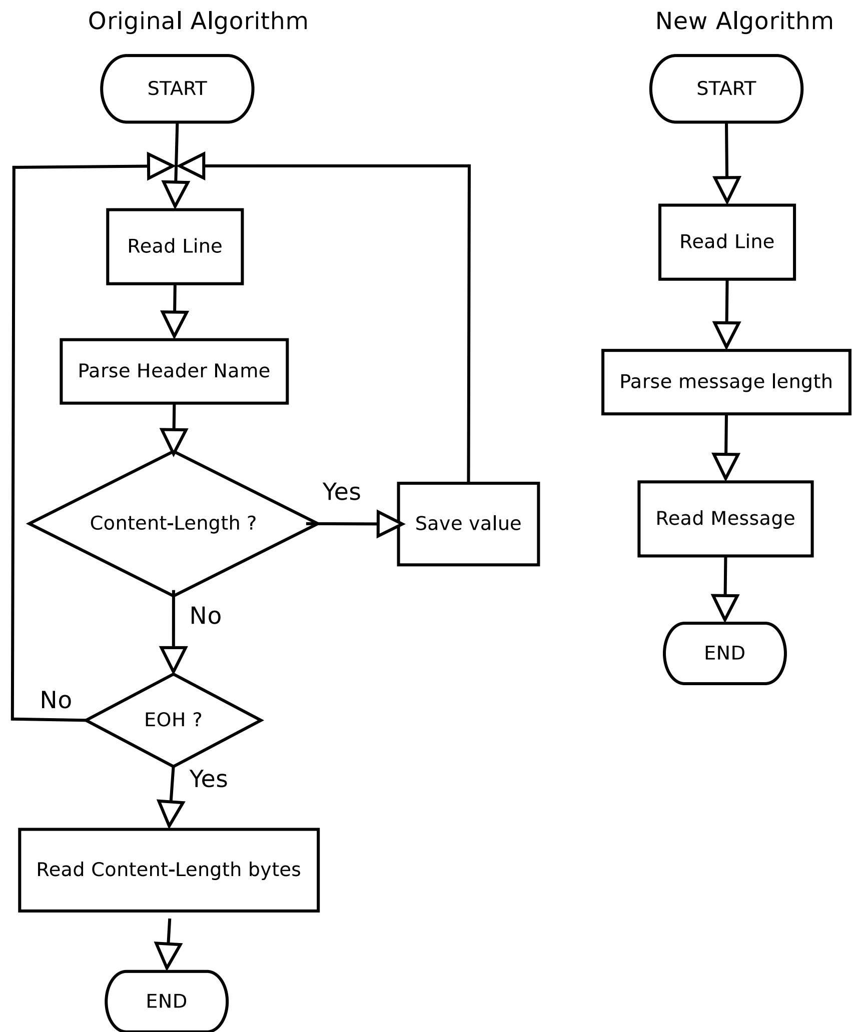

Figure 5.2: SIP Message Read Over TCP

Chapter 6

Conclusion

6.1 Achieved Results

Our SIP server is able to process almost 2000 CPS on a regular PC. This is quite good result acompared to performance of other SIP servers and call generators.

During one of the interoperability events we measured performance of our server using a professional performance measurement tool and our server running on a IBM laptop was able to fully saturate the tool.

Unfortunately performance measurement of SIP servers are not easy mainly because there is no operational experience yet. It is possible that the real deployment will require completely different tests, but currently the sipstone is the only benchmark available.

The server is available to the public and recently it started to attract wide audience. It is true that most users are not interested in the performance, but as a side effect of the performance optimizations we designed a quite flexible SIP server that gives it's administrator full control over the server's behavior. The flexibility seems to be one of the most attractive feature to users.

In addition to the SIP server optimizations we have described some bottlenecks of the SIP protocol itself. The bottlenecks were sources of lots of troubles during the development and we hope that writers of the specification will take care about implementations in the future.

6.2 Real Deployment

The described server was released under the GNU GPL license. So far the following institutions have been using the server:

- **iptel.org**—iptel.org is a know how company spun off of the German National Research Institute. Iptel.org operates a public SIP infrastructure that allows users' to sign up for an

account and use our SIP services, which are based on the SIP Express Router. Currently number of subscribers is in mid thousands. The SIP server has been running for almost 2 years, it proved to handle misconfigured user agents, broken implementations causing message avalanches. So far no misconfigured or broken client has managed to overload the server.

- **Yale University**—-The university uses the SIP Express Router to connect their PSTN network to the internet. Currently the server is serving more than 15 000 users.
- **Song Networks**—The SIP Express Router is used to provide Voice over IP services to subscribers. Song Networks is one of the largest ISP providers in Finland.
- **nic.at**—-The organization used the server to build an ENUM demo to demonstrate ENUM capabilities. The demo was successful so the organization decided to use our server to build a Voice over IP network at Technical University in Vienna. The VoIP network is going to have about 100 000 subscribers.
- **grandstream.com**—Grandstream is a manufacturer of HW IP phones. In addition to the phones the company also ships the SIP Express Router.
- **lindows.com**—Lindows is going to build a public Voice over IP network. The SIP Express Router was chosen as the SIP server.

6.3 Future Work

We plan to continue with the development of the server. Currently the biggest problem is the lack of operational experience which prevents us from more detailed performance testing. The only existing SIP benchmark is the Sipstone. The benchmark has many limitations. It is not possible to create better benchmark at the moment because we are still waiting for a large deployment of SIP.

It will be necessary to create a more sophisticated call generator that would allow us to do better performance measurements.

Bibliography

[1] Tim Brecht and Michal Ostrowski. Exploring the performance of select-based internet servers.

[2] David H. Crocker. Standard for ARPA internet text messages. Request for Comments 822, August 1982. Internet Engineering Task Force.

[3] T. Eyers and H. Schulzrinne. Predicting internet telephony call setup delay, 2000.

[4] J. Franks, P. Hallam-Baker, J. Hostetler, S. Lawrence, P. Leach, A. Luotonen, and L. Stewart. HTTP: Authentication: Basic and digest access authentication. Request for Comments 2327, June 1999. Internet Engineering Task Force.

[5] M. Handley and V. Jacobson. SDP: Session description protocol. Request for Comments 2327, April 1998. Internet Engineering Task Force.

[6] M. Handley, H. Schulzrinne, E. Schooler, and J. Rosenberg. SIP: Session initiation protocol. Request for Comments 2543, March 1999. Internet Engineering Task Force.

[7] Wenyu Jiang, Jonathan Lennox, Henning Schulzrinne, and Kundan Singh. Towards junking the pbx: Deploying ip telephony.

[8] M. Kaashoek, D. Engler, D. Wallach, and G. Ganger. Server operating systems, 1996.

[9] Poul-Henning Kamp. Malloc(3) revisited.

[10] Donald E. Knuth. *The Art of Computer Programming*, volume 3—Sorting and Searching. Addison Wesley Longman, 2nd edition, August 1999.

[11] Chuck Lever and David Boreham. malloc() performance in a multithreaded linux environment. In *USENIX Annual Technical Conference*, San Diego, California, USA, June 2000. The USENIX Association.

[12] J. Rosenberg, H. Schulzrinne, G. Camarillo, A. Johnston, J. Peterson, R. Sparks, M. Handley, , and E. Schooler. SIP: Session initiation protocol. Request for Comments 3261, June 2002. Internet Engineering Task Force.

[13] Jonathan Rosenberg. *Distributed Algorithms and Protocols for Scalable Internet Telephony*. PhD thesis, Department of Electrical Engineering, Columbia University, 2001.

[14] H. Schulzrinne, S. Casner, R. Frederick, and V. Jacobson. RTP: A transport protocol for real-time applications. Request for Comments 1321, January 1996. Internet Engineering Task Force.

[15] Henning Schulzrinne. Industrial-strength internet telephony.

[16] Henning Schulzrinne, Sankaran Narayanan, and Jonathan Lennox. Sipstone – benchmarking sip server performance. Technical Report CUCS-005-02, Department of Computer Science, Columbia University, New York, March 2002.