Computer Science

Per Sundberg, Karolina Åkerlund

Web Based Multimedia Communication with Session Initiation Protocol (SIP)

Bachelor's Project

2002:08

Web Based Multimedia Communication with Session Initiation Protocol (SIP)

Per Sundberg, Karolina Åkerlund

This report is submitted in partial fulfillment of the requirements for the Bachelor's degree in Computer Science. All material in this report which is not our own work has been identified and no material is included for which a degree has previously been conferred.

Karolina Åkerlund

Per Sundberg

Approved, June 4, 2002

Advisor: Eivind Nordby

Examiner: Martin Blom

Abstract

Session Initiation Protocol (SIP) is a signaling protocol, used, for example, in multimedia conferences. SIP is expected to be the dominating solution for telecommunication solutions over the Internet in a near future. This report describes a bachelor's project at Ericsson AB. The assignment was to investigate if a connection could be established between a web site user and a SIP user. That is, we were supposed to build a web site, where a web user could log on, and from there establish a connection with a SIP user. The major question that we had to ask ourselves, and to investigate, was how to link the HTTP protocol with the SIP protocol.

Even though we had some problems, we finally managed to establish a connection between the web user and the SIP user in the end. We almost managed to carry out all of the requirements that were set, but not quite since the signaling between the two parties was not fully completed. We came to the conclusion that a connection between HTTP and SIP is indeed possible.

Acknowledgements

We like to thank our advisor Robert Locke at Ericsson AB for his professional support and advice. You really helped us approach the problems in a structured way. Thank you.
We also like to thank our advisor Eivind Nordby at Karlstad University for his support with our report and way of analysing things. You made us think in a critical way that helped us to proceed in our work in the right direction. Thank you.

Contents

List of Figures

1 Introduction

This report describes a ten-credit bachelor's project in computer communication. Session Initiation Protocol(*SIP*) is an application layer protocol that is intended for establishing a multimedia connection between two or more users. The task involved creating a web application, that is, a web site where a user can log on to his own account and make an outgoing call to a SIP user with that user's *SIP address*. This report describes how we managed to implement this application on the basis of the requirements that where set up.

This introduction chapter will start with a description of the purpose and background of this project, which is followed by a SIP description. Next, the goal for this project and how the system should work, is described. Further the areas, that were decided not to be included in this project, are brought up. Finally, the structure of this report is explained.

1.1 The Purpose and Background of this Project

When this project started, Ericsson AB, former Ericsson Infotech(EIN), in Karlstad had already used SIP for some time, but wanted to extend their knowledge. Since SIP is expected to be the dominating solution for telecommunication over the Internet in a near future, they wanted to examine in what different ways the protocol could be used, starting with examining the possibilities of linking SIP with HTTP. Through our web application, which we call a Websip application, Ericsson AB wanted to make it possible for a caller to be independent of SIP and still have the option to call another SIP user. With our web application, a user should, from any computer be able to establish a session with a SIP user and communicate with him through, for instance Real-time Transport Protocol(*RTP*).

This project is just a start of developing such an application. We have prepared a foundation of ideas and constructions for future development.

1.2 SIP Description

SIP is the protocol that we based our project on. According to Ericsson AB, SIP will probably be the dominating telecommunication solution over the Internet in a near future.

SIP is an application-layer protocol for *signaling*, and can create, change or terminate multimedia sessions between two or more participants. A *SIP session* consists of a number of SIP connections. A SIP session is a SIP call from the beginning to the end involving two or more SIP users. A *SIP connection* is a connection between two users made using the SIP protocol. The protocol supports the setup of all kinds of media transfer, for example text, sound, or image.

SIP also makes it possible to add new connections dynamically within an already established session. For example, if two users have created a connection with sound as the medium, but want to add video to their session, this is possible through the creation of a new connection with video as the medium. This connection is then added to the already existing session. It is also possible to add more users to a session, which would create conferences.

When the SIP session is established, a multimedia protocol like RTP or any other Real-Time Transport Protocol(*RTCP*)can be applied, which takes care of the actual transmission of data. During the data transfer, SIP is waiting in the background until a user wants to end his call, add another medium/user etc. When this happens, SIP is reactivated to manage the desired change through negotiations.

Concepts that are involved in SIP can be seen in Figure 1.1 and are described below.

- To establish a **connection** a client is **connecting** to a server, and when the connection is terminating the two parties are **disconnecting**.
- Regarding the actual transmission of voice, text, mail, payments, ftp etc, **non SIP based** or **SIP based** programs may be used.
- The caller can have **multimedia programs** or **non-multimedia programs** on his

computer. SIP does not necessarily involve the use of multimedia.

- When communicating over the internet **SIP addresses**, alternatively **operator nets** or Signal System 7(***SS7***) may be used.

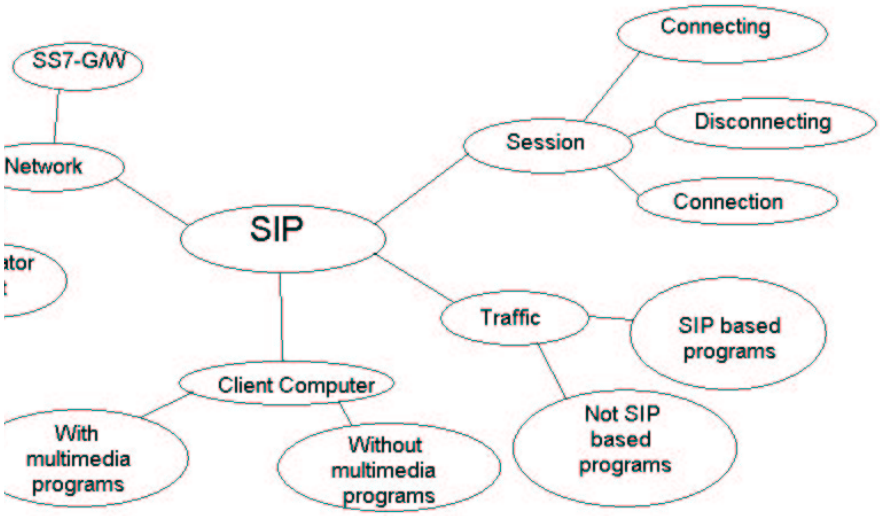

Figure 1.1: Overview of concepts that are involved in SIP

SIP is used with several others protocols. User Datagram Protocol(*UDP*) or Transmission Control Protocol(*TCP*) and Internet Protocol(*IP*) is used to transport the packages. To read more about these protocols we refer to computer communication books, for example [21].

The Session Description Protocol(*SDP*) is also used with SIP. This protocol transmits information about the media streams and other relevant information used in the SIP ses-

sions. This information is for instance what media type is used and the *session time*. For more information on SDP, please refer to RFC2327, which is found at [11].

As mentioned earlier, SIP is also used with a multimedia protocol to transmit the traffic between the participants. The protocol for this purpose is usually RTP or RTCP. Refer to [20] for more information about these protocols.

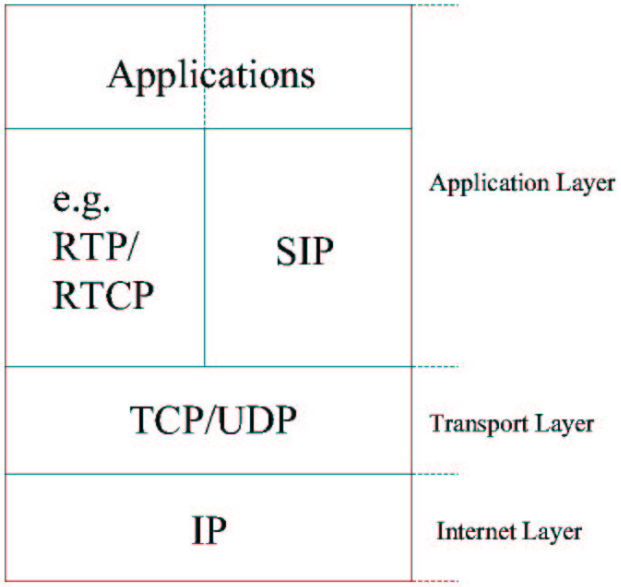

Figure 1.2: Internet protocol stack with SIP included

Figure 1.2 shows where SIP and its surrounding protocols are located in the Internet Protocol stack. The application layer is divided in two parts, where the left hand side protocols take care of the media applications, and the right hand side protocols take care of the signaling applications. It is on the right hand side, above SIP, that we have built our application. TCP and UDP take care of the transportation of the packages, and in the Internet layer, below them, the IP protocol is present.

1.3 The Goal of this Project

Our main goal was to implement a solution to be able to communicate between *HTTP* and SIP, and to examine different approaches to communicate from SIP to HTTP. The sub goals was to construct a web site, where a user could log on to a personal account with stored information about him, and from there establish an outgoing call to a SIP user with that user's SIP address. Further, we had to examine different solutions for the web user to receive an incoming call from a SIP user, through the web site.

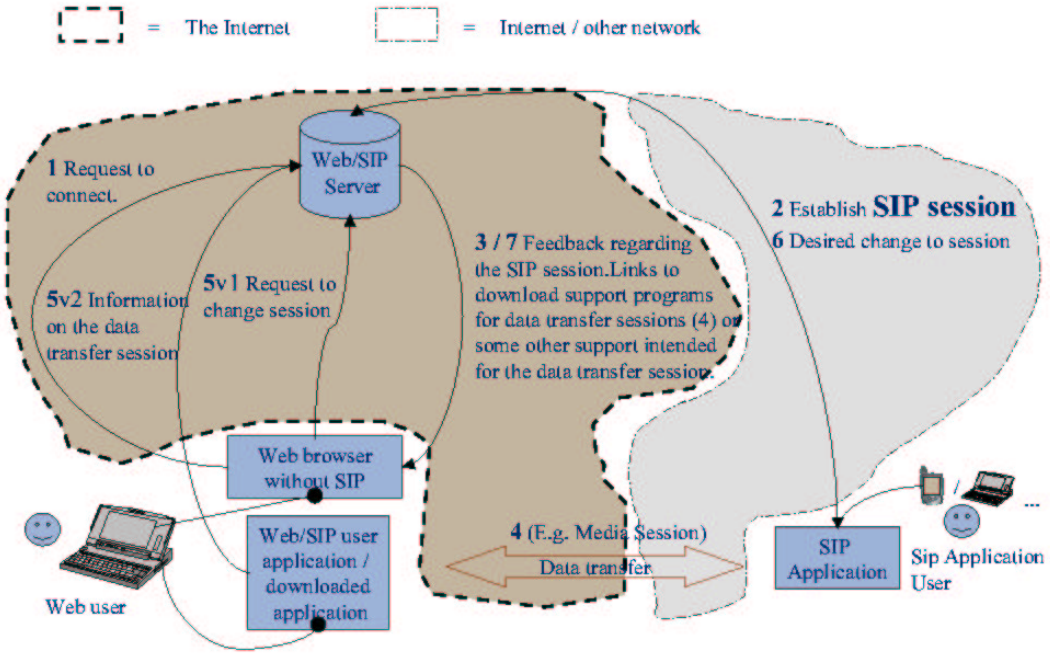

Figure 1.3: How the web user communicates with a SIP user

1.4 How a Fully Completed System is Supposed to Work

This section describes how the system, when it is fully completed, is supposed to work. So everything that this description includes is not implemented in our project. Please refer to Figure 1.3 when reading further.

A web user logs on to our web/SIP server, which is present at a web server. When the web user chooses to make an outgoing call to a SIP user, the web/SIP server establishes the SIP connection to a SIP user. The arrows 1 to 3 illustrate this. When the signaling between the two parties is done the web user and the SIP user can talk to each other directly. This is shown by arrow 4. This communication does not go via the web/SIP server. The reason is, that they are using some other protocol, like RTP, for the media transfer and SIP is not involved in that part of the session. If the web user application ends the data transfer or if the web user wishes to change the session, the SIP session is also supposed to be changed, The Web/SIP server has to be notified. This is done by the application notifying the Web/SIP server as seen in arrow 5 version 2(5v2) and the user clicking the desired options in the browser shown in arrow 5 version 1(5v1), respectively. As seen on the dashed markings, the web/SIP server is located on the internet, but the person whom to call can be located on any other net that can be reached through a SIP address.

1.5 The Scope of this Project

We have not put in much effort into constructing a good-looking web site with many functions, therefore only the most necessary functions are included. Further, we have not taken security into consideration when constructing our system. We have also limited the number of participants in a session to two.

1.6 The Chapter Structure of this Report

The document is intended to be understandable by a reader whom has at least two years of studies in the field of computer science. To give the reader a better insight in what we were supposed to accomplish with this project, we will discuss the preconditions and requirements put on this project in Chapter 2. In Chapter 3 we give an overall description on how to use our application so that the reader and possible user of the application gets an overview on how it works. This overall description serves as a background for Chapter 4 where we describe our whole system, and how the different parts in the system interacts with each other. Then Chapter 5 gives a more complete description of the different parts of the system and how we have designed them. In Chapter 6, we describe how we implemented and tested these different parts and how we made the whole system work. We examine the possibilities to receive incoming SIP calls from SIP users on the web site in Chapter 7 and in Chapter 8 problems that occurred during our work with the project is discussed. Chapter 9 discusses future development possibilities and finally we discuss our conclusions regarding the work as a whole in Chapter 9.

A clarification to words written that are written in italic during the report can be found in appendix A. Appendix B describes a tool called the GNU build system that was used developing parts of our application. Appendix C contains our first and our last time schedule, showing what we, when the project commenced, thought we would have time to accomplish, and what we really did have time to accomplish in the end. Finally, in appendix D we describe alternative tools that could have been used in this project, but never were.

To make this bachelor's project easier to read we have sometimes chosen to refer to **the user** of our application, with **he**, **him** or **his**.

2 Prerequisites and Requirements

In this chapter we discuss the prerequisite requirements concerning our development environment, which are especially important for the reader who has the intention to develop our application. Then we describe the requirements that were put on this project, to give the reader a greater knowledge about what exactly we were required to do by Ericsson AB. In this chapter there are references to Chapter 4.

2.1 Requirements on the Development Environment

These are the requirements that were put on our development environment:

- To have access to two UNIX/Solaris computers, with the possibility to log on into a *Windows Terminal Server*. We almost exclusively used UNIX when developing our application though.

- The graphical parts of the web site have to be constructed with HTML aimed at a Netscape Communicator environment.

- To Use CGI scripts to develop the engine of the web site.

- To use open source SIP(*oSIP*) as our SIP stack. oSIP is a help library to make an implementation of a SIP application easier. Please refer to Section 5.2 for more information about oSIP.

2.2 Requirement Specification

This is where we have documented the requirements that where put on our project. They are subdivided in requirements we must fulfil, should fulfil, and could fulfil.

2.2.1 MUST be Fulfilled

These are the requirements that had to be fulfilled to make our work meaningful.

- A web site must be constructed, where a user who is independent of SIP, is able to log on.
- Each user must have his own account, where his personal information and SIP addresses are stored.
- From the web site the user must be able to make an outgoing call to a SIP user by stating a SIP address, either manually or by selecting one from his address book.
- The web/SIP server(see Figure 4.1) must establish a SIP connection between these two users.
- After calling the SIP user, a simple data transfer from the SIP user to the web user must be carried out, and a confirmation message of this data transfer must be shown to the web user.
- The different possibilities, on how a web user can receive an incoming call from a SIP user must be examined.

2.2.2 SHOULD be Fulfilled

The following should be fulfilled, but it is not a disaster if we fail due to lack of time etc.

- One of our solutions, allowing a web user to receive an incoming call from a SIP user, should be implemented.
- After the SIP connection is established between the calling SIP user and the web user, a simple data transfer from the SIP user to the web user should be carried out, and a confirmation message of this data transfer should be shown to the web user.

2.2.3 COULD be Fulfilled

The following points could be implemented if time allows.

- A transmission of text/voice/images between the two parties could be implemented.
- A function to create new accounts on the web site could be implemented.
- A function to add new contacts to an already existing web user account could be implemented.
- A function to add/delete connections or users to an existing session

How we managed to carry out these requirements, can be read about in the report and especially in Chapter 8.

3 System Usage

This chapter presents the different use cases in the system, which we have implemented. A number of scenarios will also be described intended to give further insight and understanding to users of this system. A fully description of the system is given in Chapter 4 and 5.

3.1 Use Cases

In this section the use cases included in our web application are described. We will only describe the most important ones though, that is, the use cases we have implemented in our system. Figure 3.1 shows the different use cases.

These are the use cases for the web user:

- **Log on to the web site**: The web user can log on to the web site, either with a username and password of his own, or as guest if he does not have his own account.

- **Look at/Change user profile**: When logged on to the web site, the user can look at and/or change his personal profile.

- **Look in address book**: The user can look at the SIP contacts in his address book, and also choose a SIP address to establish a connection to.

- **Call SIP user**: When logged on to the web site, the user can call a SIP user, with the SIP users SIP address.

These are the use cases for the SIP user:

- **Receive call** : A SIP user can receive a call from a web user.

Note: The extend-arrow in Figure 3.1 means static dependency and is Unified Modelling Language(*UML*) standard.

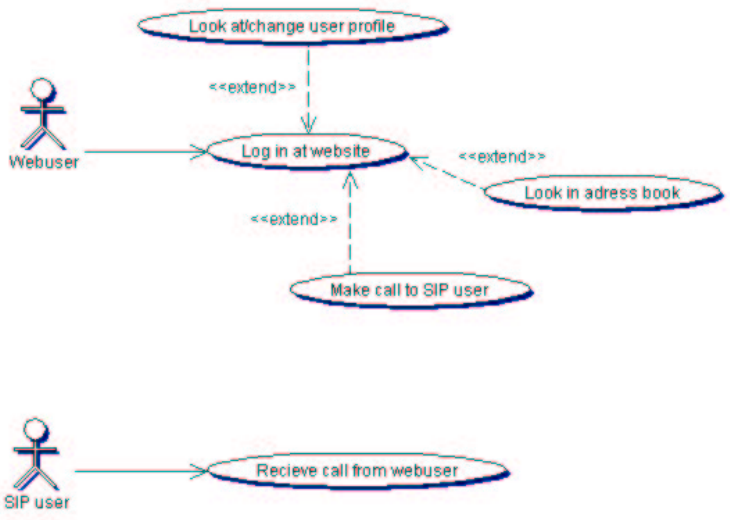

Figure 3.1: Use cases included in our system

3.2 Scenarios

A number of scenarios that can arise when using the web site will be described below. All scenarios shown, describe successful cases of its use case. References are made to user screens in Chapter 5.

3.2.1 Change User Profile

This scenario shows how the web user is to proceed to change his personal profile on the web site.

- Fill in user name and password and log on.
- Click and proceed to **Your Profile**. (Figure 5.5)
- Change the profile if necessary.
- Click **Ok** to confirm and to return to **Home**. (Figure 5.3)
- Log out.

3.2.2 Make a Call - Version 1

This scenario describes how the web user is to proceed to make a call to a SIP user. Assume that the user is already logged on.

- Click and proceed to **Make Call**. (Figure 5.4)
- Write the SIP address to the callee in the shown text field.
- Send request of connection.

3.2.3 Make a Call - Version 2

This scenario describes an alternative on how the web user can make a call to a SIP user, that is, through the address book. In this particular case the address book is opened via **Home**. The user is presumed to already be logged on.

- Click and proceed to **Address Book**. (Figure 5.6)
- Choose which SIP address to use, by clicking on **Use** next to the address. This takes the user automatically to **Make Call** with the chosen address written in the text field.
- Send request of connection.

3.2.4 Make a Call - Version 3

This scenario is similar to the previous one except that the address book is opened through **Make Call** instead of **Home**. The user is presumed to already be logged on.

- Click and proceed to **Make Call**.
- Click on Address Book-button in the **Make Call**-window to proceed to **Address Book**.
- Choose which SIP address to use, by clicking on **Use** next to the address. This takes you automatically to **Make Call** with the chosen address written in the text field.
- Send request to call

4 System Description

To give a better view on how the system is supposed to work we will describe how the system looks as a whole prior to explaining how our system is designed. We will also

describe the different parts in the system, which they are and how they interact with each other.

4.1 System Overview

Users are able to log on to a web site, which is located on a web server. We call the web server that is attached to a SIP server web/SIP server since they interact and ultimately function as one. When the web user wants to call a SIP user with a SIP address, the web/SIP-server establishes a connection to the correct SIP server, which is used by the correct SIP user. Please refer to Figure 4.1 for better understanding.

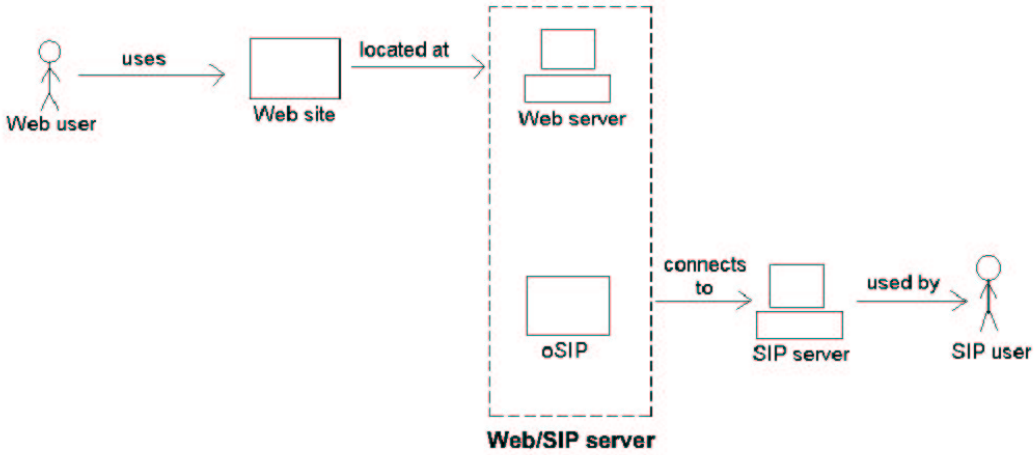

Figure 4.1: System overview

4.2 The Different Parts in the System and how they interact

Our system consists of three vital parts namely the web site, the SIP stack (oSIP) and the link between these two. Now follows a description of each part, and then we will also explain how the parts interact with each other to make the whole system work.

4.2.1 The Web Site

The web site is of course one of the main parts in our system, and this is where a user can log in and call a SIP user with the SIP users SIP address. The web site is just a normal web site implemented in HTML, and the functionalities is implemented with CGI-script written in Perl. Read more about the design and implementation in Section 5.1 and 6.1 respectively.

4.2.2 The SIP Stack

The other main part in our system is SIP. For a web user to be able to call a SIP user, two criterions have to be fulfilled. The first is that the web server, were the web site is located, has to use a SIP server, and the other criteria is that the SIP user also has to use a SIP server. The latter criterion is something we will not consider in our solution, but which is up to the user of our system to ensure. The former criterion though, has to be implemented in our system for it to work. For this purpose we used oSIP, which is an implementation of a SIP stack, and on this stack we built our own simple *User Agent* to be able to use it. Read more about our design of this User Agent in Section 5.2 and about our implementation of the same in Section 6.2.

4.2.3 The Link Between the Web Site and the SIP Stack

As mentioned in Section 1.3, our major goal was to make it possible for a web user to contact a SIP user. So the third major part in our system should be a link from the web site to the SIP stack. This part is maybe the most important part, since our system would

not work at all without it. In Figure 4.1 the web site and oSIP together is what makes the web/SIP server, and it was the link between the web site and oSIP that had to be designed.

To design the link we used a software named Simplified Wrapper and Interface Generator(*SWIG*). Read about SWIG and why we used it, the design of the link and our implementation of the same in Section 5.3 and 6.3 respectively.

4.2.4 The Interaction between the System Parts

Figure 4.2 shows how the different parts interact with each other. Each block represents a system component and the arrows represent in which way these components interact with each other. Each block will be explained further in Chapter 5 and 6, excluding the Transport layer component, since the reader of this report is assumed to be familiar with basic data communication.
The meaning of each block:

- HTML is the graphical part of our web site.
- CGI is our web site engine, which takes care of the functionalities of the web site.
- SWIG is the glue that connects our CGI script to our User Agent for oSIP.
- oSIP is an open source SIP implementation. It acts as a SIP library and parser in the behalf of our User Agent.
- Websip is our User Agent for oSIP.
- Transport layer is the layer that were used to send the packages.

The blocks HTML and CGI together give us the web site part of our system. oSIP, Websip and Transport layer is the SIP part and SWIG is the link between the web site and SIP.

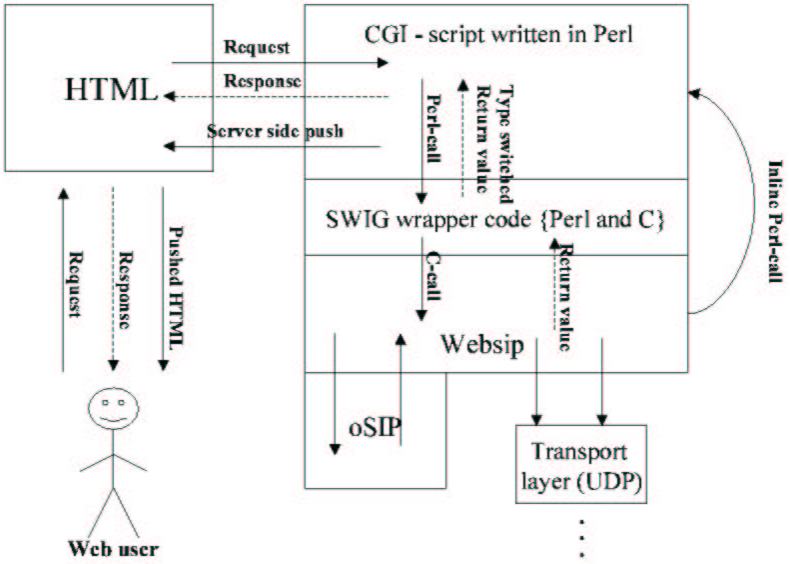

Figure 4.2: The Interaction between the parts in the system.

5 Description of the Design of our application

Before we describe how we implemented each part in the system, in the next chapter, we will describe how we designed them. For every part we will describe how we came to choose our way to design it and what tools we used for the purpose.

The chapter starts with a description of the design of the first main part, the web site. Next the second main part will be described in terms of how oSIP works and how we designed our User Agent for oSIP. The last section will describe the most important part in our system, namely the connection between the web site and the SIP stack. This section will explain why we needed to link C and Perl and how we proceeded to do this.

5.1 The Design of the Web site

This section explains the design of the web site part in our system, which are the shadowed blocks in Figure 5.1. At first, the section explains why we have chosen the particular layout of the web site. Secondly the design of the web site engine is examined, and this is where we, for example, will discuss why it involves certain functionalities.

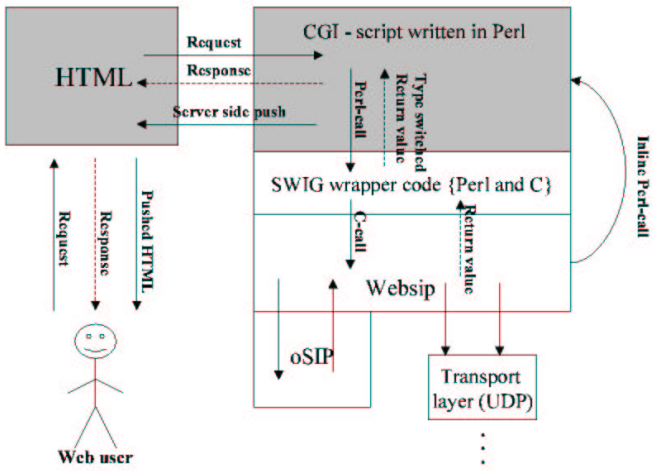

Figure 5.1: The web site part in the system

5.1.1 Graphical Design of the Web Site

Before we will go further into details regarding the design of the web site, we want to explain why we have chosen to pick our particular layout. Since we have focused on the functionalities of the web site, its appearance is not that extraordinary.

Figure 5.2 shows the first page of our web site where the user logs on. The reason why it looks as it does with name and login frame, is because its layout follows normal

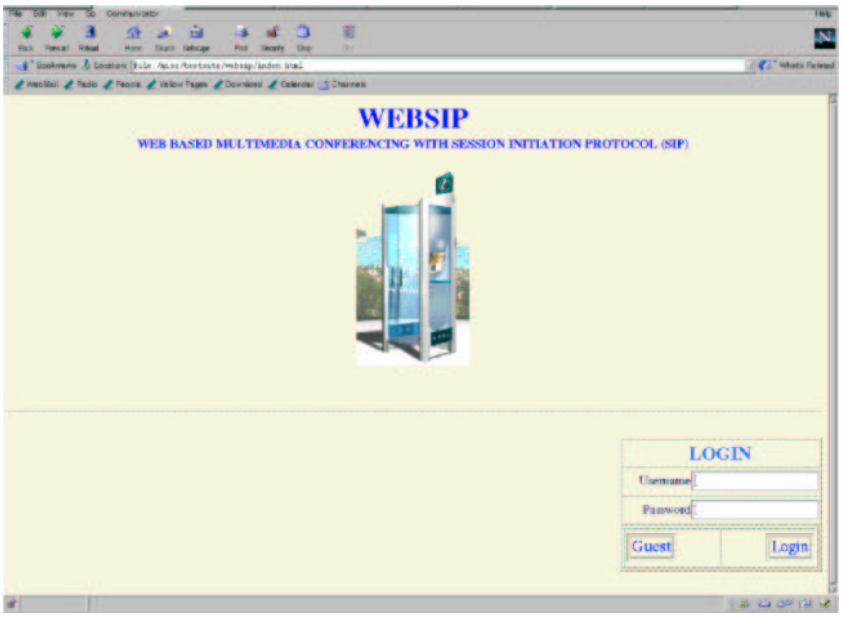

Figure 5.2: The window where the user logs on

standards seen on the Internet. It felt like the most natural way to design it.

Figure 5.3. shows the screen after logging in. A number of click-able buttons with underlying functionalities can be found here.

We have chosen to divide our web site into frames because of two reasons. Firstly most of the similar web sites on the internet makes use of frames e.g. web based email like Yahoo! Mail or Hotmail, and we did not want to change this concept since frames probably contribute to easier facilitation for the user. Secondly it also simplifies the usage of the web site since all the functions are constantly visible in one frame.

Figure 5.4. shows the screen where the user can **make a call** to a SIP user. The SIP address is written in the text field, the scrollbar shows the different kinds of alternatives of media that can be transmitted between the participants, and a button to transfer the user to the address book. The scrollbar for media is there because it is an important part in the SIP protocol. This function is not implemented though. This page is simple, intending not to make things difficult for the user.

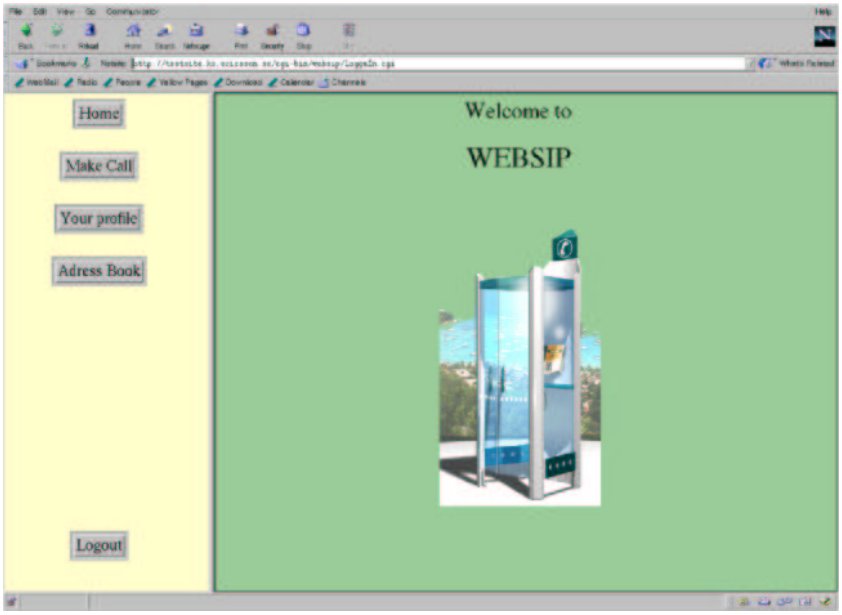

Figure 5.3: This is **Home**, where the the user is taken when he is logged on

The window **Your Profile** (please refer to Figure 5.5.) shows the user his profile where he also can change it with ease. The page includes the user profile in a numerous text fields and two fields intended for changing the password. The reason for including the functionalities **look at profile** and **change profile** in the same screen is for the sake of simplicity.

The users SIP-contacts together with their SIP addresses, is shown in the window found in Figure 5.6. We have chosen to include name and addresses in a table with use buttons on the side according to typical web based mail style.

5.1.2 Design of the Web Site Engine

In this subsection functionalities of the web site is addressed. One could imagine creating a web site where the user only could write a SIP address and request a call. That is, a web site that only consists of a page with this functionality and nothing else, since the main part of the work was to investigate the possibilities of establishing a connection between

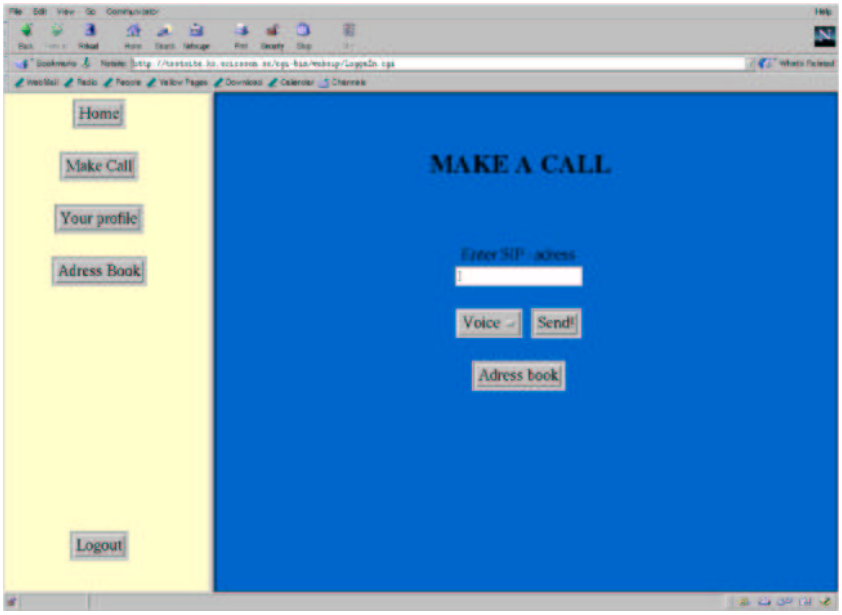

Figure 5.4: This is **Make Call**, where the user can make a call to a SIP user

HTTP and SIP. The following describes the functionalities that was required (except make a call to SIP user).

- The user can log on to the web site with a personal account consisting of different personal information. The information includes given names, surname, country, SIP addresses, username, and password. A non-registered user can log on as guest.

- After logging on the user can view and change his personal information.

- The user has a personal address book where his SIP contacts are stored.

Please refer to Section 9.1 to get further insight on more functionalities that could be implemented.

For implementation of the functionalities we used several CGI-scripts written in Perl. Below follows a short introduction to CGI.

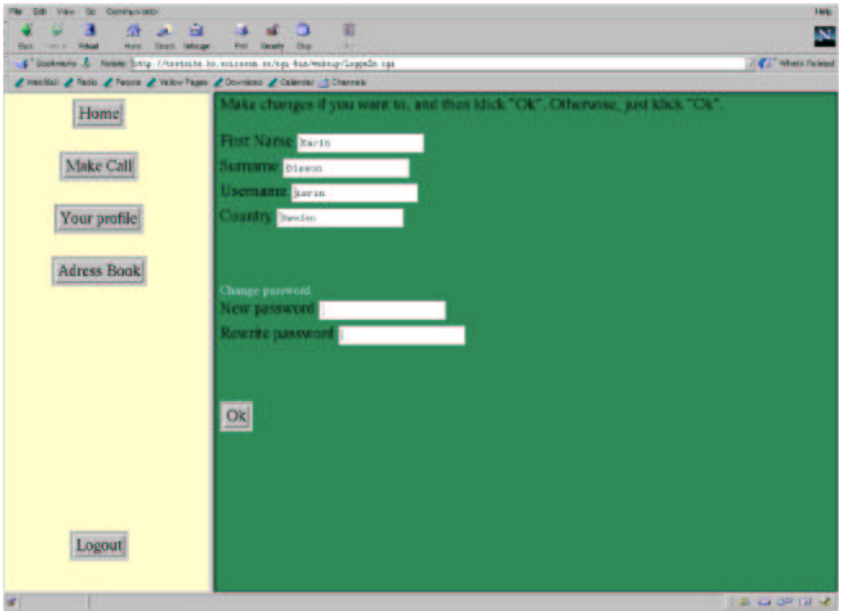

Figure 5.5: This is **Your Profile**, where the user can look at/change his profile

CGI-scripts CGI or Common Gateway Interface is a tool to construct web sites. A web site that only is based on HTML is static, meaning that it only has one state (e.g. compare to a text file). A CGI-script is executable and can send information dynamically. When a functional CGI-script is present at a web server it can accomplish two things:

- It can receive information from a web browser and process it in a desired way.
- It can process information and send information to a user that made a request.

Examples of this include reading information from a database and printing selected tuples on a web site, or receiving information filled in by a user from a web site form. CGI-script can be written in many different languages including Perl, TCL, Applescript and Visual Basic. It can also be written in C/C++ but it will make it more complicated. Further reading about CGI-scripts can be found in [22].

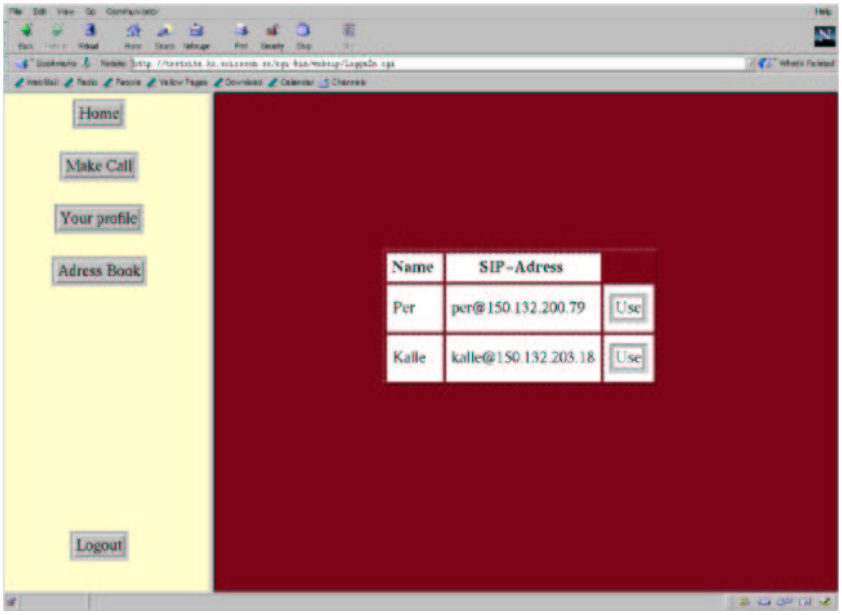

Figure 5.6: This is **Address Book**, where the user can look in his address book

Why we have chosen Perl as Scripting Language The reason for choosing Perl as scripting language was partly because we where recommended to do so, but also because Perl was the most familiar scripting language for us at that time. According to us, Perl is also one of the easier scripting languages, and it is well known to be used in CGI-scripts.

Section 6.1 gives more details about how the web site was implemented.
Now we have designed our web site, and the next step is to design the other main part of the system, namely the SIP part.

5.2 Design of the User Agent for oSIP

This section describes the design of our User Agent to oSIP, which contains the shadowed blocks of our system, shown in Figure 5.7. At first we will describe SIP a little more in detail and how a SIP connection is made, then we will explain oSIP and why we came to choose this particularly implementation of SIP. Finally we will describe how we used oSIP to design our User Agent.

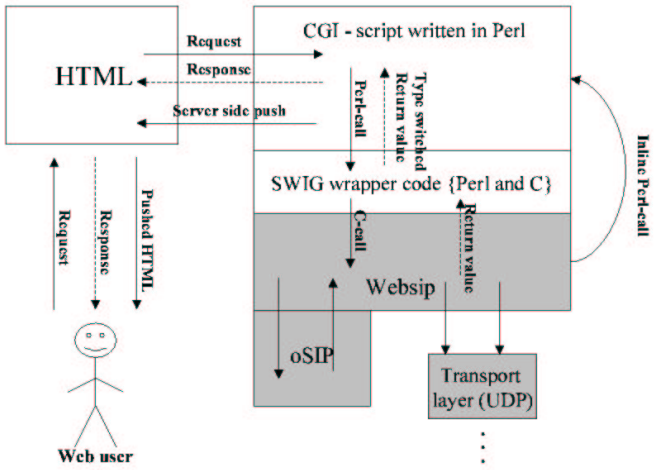

Figure 5.7: The SIP part in the system

5.2.1 SIP Functionalities

Before we describe oSIP and our User Agent, we will briefly explain some SIP functionalities.

A SIP session involves three parts, namely the connection between the parties, the data transfer and the termination of the session. Consider the following scenario, which is a session between two parties, to understand what a session looks like. No changes are made during the session, like adding another party or medium. See Figure 5.8 for better understanding.

SIP user A sends an INVITE message to SIP user B:s SIP address, for instance joe@ericssonSIP.com). The SIP server sends a TRYING signal to A, telling A that it is trying to locate the owner of the SIP address. Then the server sends an INVITE to all the devices such as SIP mobiles or hand held computers, that B has registered on the server, under this SIP address.

Further, B:s devices sends RINGING to the server which in turn sends RINGING to A. When the SIP user B has answered, a third message OK is sent to the server or to A directly. When A receives an OK message it sends an ACK to B, through the server or to B directly, and when the ACK is received at B, A and B have a connection. Now the two parties can start to transmit information between each other, in the chosen media type. The media type is actually negotiated about between the two parties, after being suggested by A, but this is not shown in the picture.

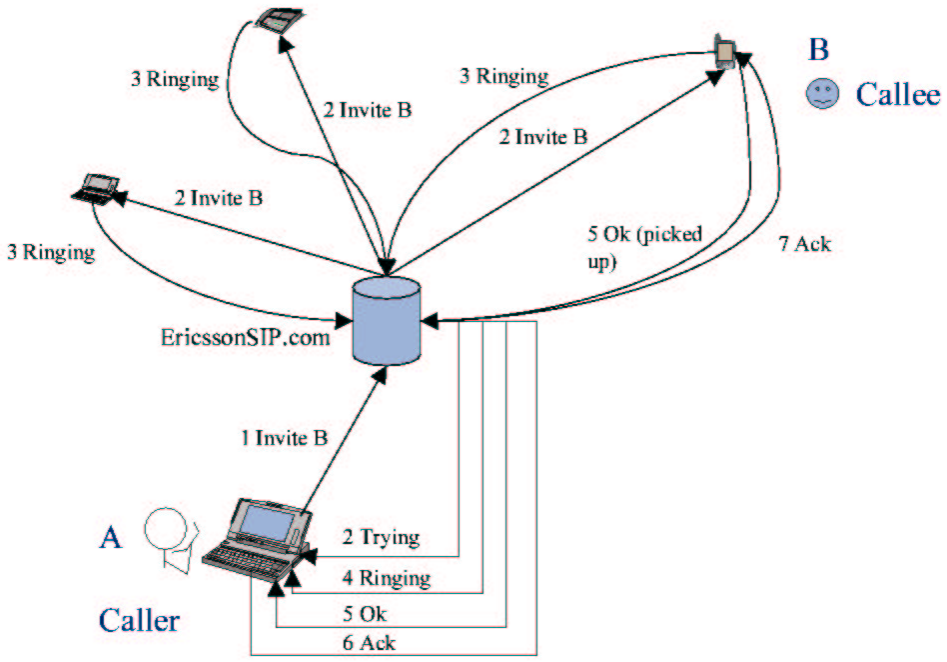

Figure 5.8: A SIP session through a SIP server

SIP does not take care of the transmission of data. Rather, a multimedia protocol like RTP or RTCP would do this. The data transfer does not go via the SIP server but directly between the two SIP users as shown in Figure 5.9. When one of the parties decides that

he wants to end the session, he sends a BYE to the other party, whom responses with OK. The BYE message does not need to be sent by the caller, that is SIP user A in this case, but by either one of the parties that wants to end the session.

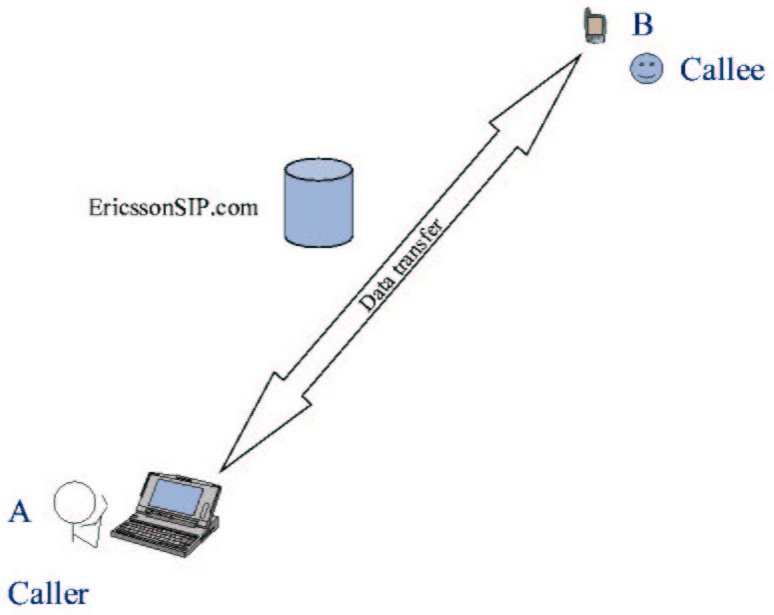

Figure 5.9: The data flow between the two SIP users in the session

A SIP message is either a request (e.g. INVITE) or a response (e.g. RINGING, OK). A request is sent from a client to a server, and a response is sent from a server to a client. Both types of messages has a start line, at least one header field, an empty line which indicates the end of the header fields, and an optional message body.

Request Messages The start line in a request message contains the following:

- **Method name:** This defines the method that the client is sending e.g. INVITE or

BYE

- **Request-URI:** This indicates the user to which the request is being addressed, that is the SIP address.
- **SIP-version:** The version of SIP that is being used, almost always "SIP/2.0".

These are the header fields that has to be included in a request message:

- **Request-URI:** Same as for the start line.
- **To:** Specifies the recipient of the request.
- **From:** Specifies the identity of the initiator of the request, that is, the client.
- **Call-ID:** A unique identifier to group together a series of messages. Each message within a session has the same Call-ID.
- **CSeq:** Consists of a sequence number and a method, and it identifies and orders transactions. The method is the same as for the request.
- **Max-Forwards:** Serves to limit the number of hops a request can pass by on the way to the destination.
- **Via:** Indicates the transport used for the transaction, and identifies where the response should be sent.

There is also one header field that must be included in an INVITE message and that is **Contact**. This field contains a SIP address at which the User Agent would like to receive the requests, and subsequent request, which are used to change a session.

Response Messages The start line in a response message contains the following:

- **SIP-version:** Same as for request messages.

- **Status-Code:** A 3-digit integer which indicates if the request was understood and satisfied.
- **Reason-Phrase:** Gives a short description of the Status-Code.

These are the header field that has to be included in a response message:

- **From** Contains exactly the same as the From header field in the request.
- **Call-ID** Contains exactly the same as the Call-ID header field in the request.
- **CSeq** Contains exactly the same as the CSeq header field in the request.
- **Via** Contains exactly the same as the Via header field in the request.
- **To** Contains exactly the same as the To header field in the request.

A request message could, except INVITE, BYE and ACK, also be CANCEL or REGISTER. A CANCEL request is used to cancel the previous request sent by the client. A REGISTER request is used to register locations for destinations, so SIP can find the host at which the destination user is located.

To register a SIP device, the client sends a REGISTER request to a special server called a Registrar, which acts as a domain's location service, and reads and writes mappings for the contents of the REGISTER requests.

SIP can also use two other servers, called proxy servers and redirect servers. A proxy server routes SIP requests to servers and SIP responses to clients. A redirect server sends a response back to the client, which include a new SIP address, to which the client should send the request to instead. The client is redirected.

SIP also has a state in the header, which tells the server or the client what kind of message that has been received. The different kinds of messages or transactions are *ICT*, *IST*, *NICT* and textitNIST. The state machine is set to ICT, when the client sends an INVITE request, and to IST when the server sends a response to an INVITE request. The

state machine is set to NICT when the clients send a request that is NOT an INVITE, and to NIST when the server sends a response to a request that is NOT an INVITE.

For more reading about SIP in general, and other SIP functionalities, please refer to RFC2543 and the newest RFC Internet-Draft, which are both found at [11].

5.2.2 oSIP

This subsection describes oSIP in general, an open source SIP implementation written in C. oSIP can be read about and downloaded at [1].

OSIP can be used for implementing a range of things including:

- A variety of SIP servers and proxy servers (conferencing servers etc).
- SIP registrar servers used to keep track of SIP clients locations.
- SIP User Agents to be used in phones, desktop computers etc.

oSIP is a library and provides an up-to-date implementation of the latest RFC draft 2543 found at [11]. The main responsibility of oSIP is to:

- Provide a parser for the SDP
- Provide a parser for the SIP
- Provide a library to handle SIP transactions.

oSIP advantages:

- It is a small and foreseeable library
- It is fast (one of the reasons is that it is written in C)
- OSIP is small and provides only functions needed to fulfil the RFC standard that in turn makes oSIP extensible to a variety of SIP usage areas.

- it is portable, that is, the code and the makefiles contain conditions aimed at several different OS platforms.

oSIP disadvantages:

- oSIP on its own is not usable, writing an extension is needed.
- As of this writing there is not any up-to-date documentation of oSIP other than the code itself that makes oSIP very hard to understand, thus also making it a lot more time consuming to implement an extension of oSIP.
- There are very few comments in the actual code, making it even harder to understand the oSIP stack.
- The security implemented in oSIP is low.

5.2.3 Our User Agent - Websip

In the beginning of our work with oSIP we used the oSIP edition 0.7.8, which had an example included. This example was not a real User Agent but it could send requests and receive responses from a server though. So we took the simple way and tried to use this example in our system, instead of writing our own User Agent. This was a mistake though, since this example included a lot more than we needed, and it was too big, so it gave us lot of problems with the linking with SWIG (see Section 5.3 and 8.2). So we reconsidered and started to build our own small User Agent, which could only send an INVITE and receive responses for the INVITE. Since oSIP is undocumented we opted to examine Jacks Open Source User Agent(*JOSUA*), which is a beginning of a User Agent, based on oSIP edition 0.8.3, to be able to understand the workflow of a typical User Agent. We extracted certain parts of the code which we then implemented in our own User Agent prototype. Our User Agent involved sending an INVITE and receiving responses for this INVITE (RINGING

and OK). We considered this to be enough to examine the possibilities of a web based User Agent. Other possibilities where also considered and can be read about in Section 9.1.

oSIP supports multiple threads, which in turn makes it possible to run several users on the same stack. Websip is at current not able to have several users, and supports only one user for test purposes.

The interaction between the User Agent Websip and the oSIP stack can be seen in Figure 5.10

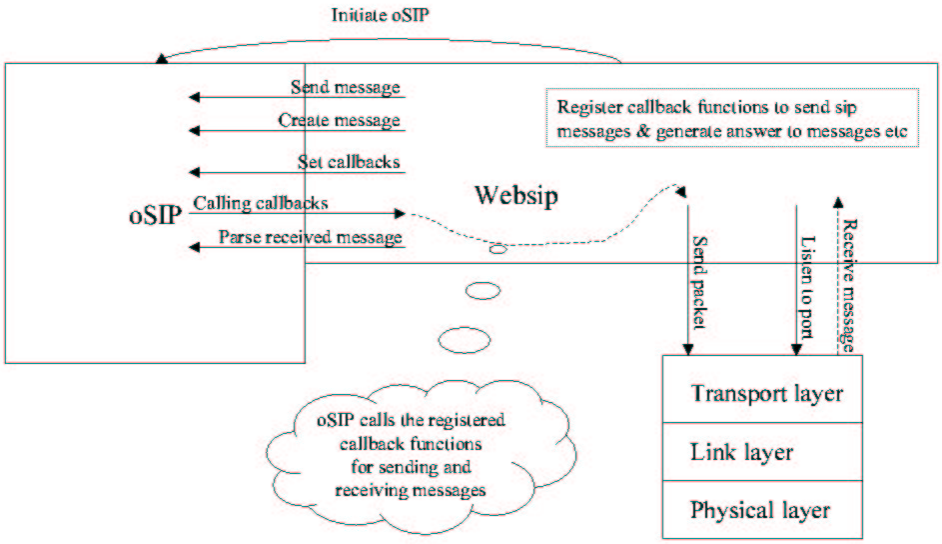

Figure 5.10: The interaction between Websip and oSIP

Calling *callbacks* Calls the registered functions when a specific event occurs in oSIP.

Websip functionalities:

Set callbacks Define which functions to send SIP messages, or be called when an event occurs etc. this way oSIP is independent of which transport layer to use, it only demands that the callback function has a specific form, for instance *cb(porttosend).

Parse received message Call the build in parser in oSIP with the message received from the transport layer. This will lead to the calling of a callback function specific for that message.

Send packet The callback function registered to send SIP messages sends messages to the transport layer (in our case UDP).

Listen to port A Websip function that listens for incoming messages.

Send message Send a message. It is up to the user of oSIP to decide when to send a message. The messages can be stored in a queue if it is a multithreaded environment and the oSIP user have a thread listening to the queue sending the messages as soon as they arrive.

Initiate oSIP Initiate and start up the oSIP stack.

5.3 Design of the Link between the Web Site and Websip

Now we have designed our web site and our User Agent for SIP, so now it is the link between them left, which is the shadowed part in Figure 5.11. This section describes the general solution for linking our web site with our User Agent Websip. We will discuss different possible solutions that we had in mind, which one we have chosen, why we picked it, and how it works.

5.3.1 Why We had to Link C and Perl

As explained earlier our web site's CGI-scripts is written in Perl and oSIP and our User Agent is written in C. Therefore, to be able to establish a link between the web site and

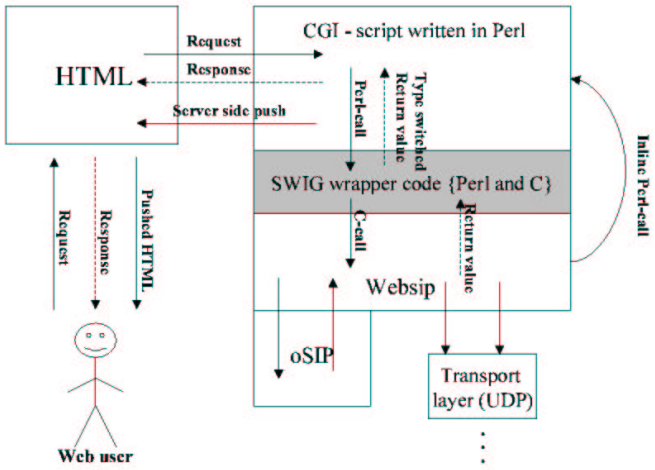

Figure 5.11: The part in the system for linking the web site with Websip

our User Agent, we have to call C-functions from the CGI-script. When we had concluded that we start digging in the Internet for different solutions, and Subsection 5.3.2 describes the ones we found.

5.3.2 The Tool for the Task - SWIG

We used the tool SWIG to wrap C code into Perl. Please refer to appendix E for other optional tools that we found, but did not use.

SWIG SWIG stands for Simplified Wrapper and Interface Generator and can be read about and downloaded at [8]. It is a tool for integrating Java, and script languages like Perl, TCL, python etc. with C/C++ in a seamless manner. With SWIG you have the ability to write powerful C/C++ modules and create interfaces easily.

The program itself is constructed based on XS (see appendix E), when wrapping Perl. It

takes care of the tedious task of construction wrapper code, which has to be done manually in XS. All that needs to be done is to define the C-functions which to be used by Perl in an interface file, for instance filename.i, which corresponds to the C file of the same name, filename.c. SWIG then generates wrapper code to this interface.

The wrapper code is stored in filename_wrap.c and contains all the functions that where specified in the interface file, and all the functions to convert types to and from the target language. After compiling, the wrapper file object is linked with the compiled C/C++ objects that contain the functions specified in the wrapper object and any other objects that needs to be linked. This file is a shared object that can be loaded from Perl. Please refer to Figure 5.12 for better understanding.

Subsection 5.3.4 describes an example on how to write Perl wrapper code for an c-program example.c.

Advantages:

- Platform independent.
- Powerful (Little has to be written to achieve something)
- Relatively easy to learn (We thought so).
- Interface files specifies clearly the functionalities, which can be used in the target scripting language.
- The interface file templates makes SWIG work on a higher more abstract level, which is easier to write and understand.
- SWIG supports wrapping C/C++ to other languages than Perl like TCL Python, Java , Guile , Mzscheme or Ruby.

Disadvantages:

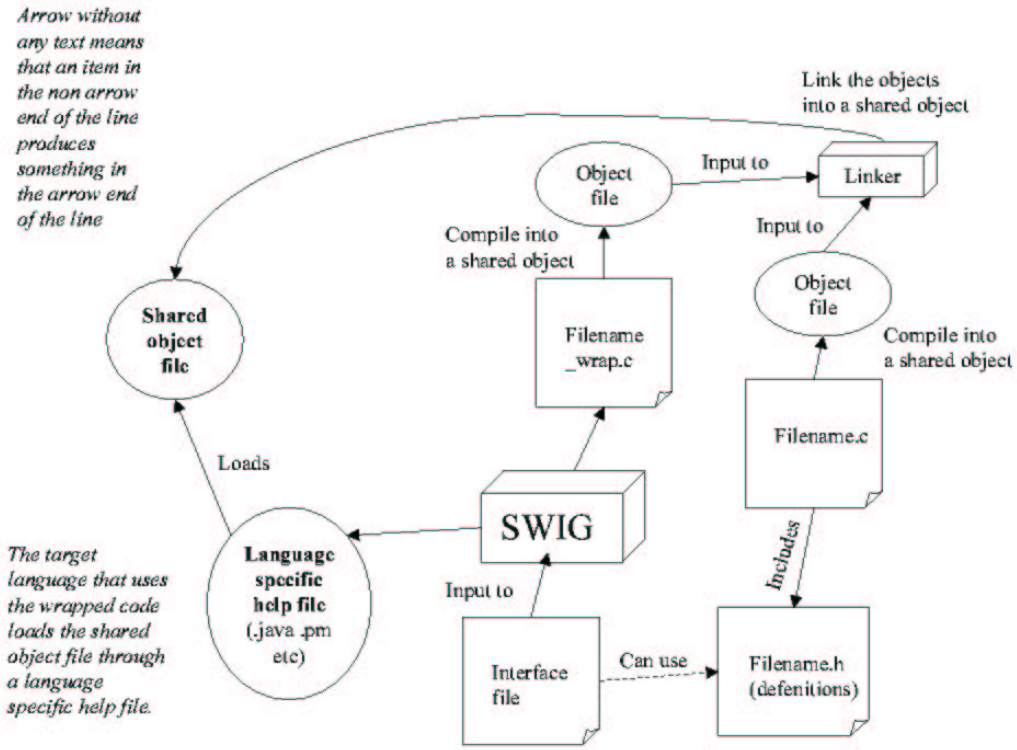

Figure 5.12: The procedure of creating a wrapped object in SWIG

- Takes some time to learn.
- Is not too flexible.

5.3.3 Why We have chosen SWIG

We have chosen SWIG because of the large amount of code that we had to wrap. SWIG also supports other scripting languages, which in turn can be integrated with each other. This results in more flexibility, when wanting to change the implementation of the Websip application. For example, if the developer was to choose Java instead of Perl, he would

only had to change a flag, -java instead of -perl5, perhaps the interface files and of course the web site code.

5.3.4 In Depth: SWIG

As mentioned SWIG demands the specification of an interface file that might look like:

```
%module example
%{
#include <example.h>
%}
void init();
char* getMyString();
```

%module tells SWIG what the name of the module and the C file for the intended wrapping is. In the above case the module/c-filename is 'example' and the wrapper code that will be generated by SWIG will be outputted as example_wrap.c. Also, in our case, a Perl module example.pm, will be generated. This is used to access and load the module in the Perl script.

All the code which is included inside %{ %} will be directly inputted into the modulname_wrap.c file. Everything else is a specification on what functionalities the interface between the C/C++ program should include. The developer could include a whole header file if desired by stating %include nameoffile.h

The functionality of this example include two functions void init(); and char* getMyString();. These two functions can be called directly from the target language. In the Perl module that will be created in this specific example can be accessed firstly by declaring use example; in the beginning of the Perl script and then accessing any in the interface file declared function or variable by typing example::function/variable, for instance example::init();

Figure 5.13 explains how the scripting language interacts with C/C++.

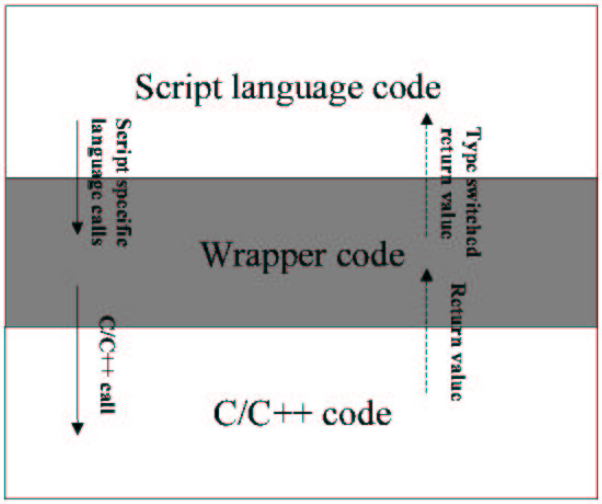

Figure 5.13: The interaction between scripting languages and C/C++ through SWIG generated wrapper code

The generated wrapper code takes care of all type conversions and function calling. With SWIG we where able to connect our CGI script to our Websip User Agent.

6 Implementation and Testing

This chapter describes the implementation of our system. We will explain the test phase, which tests was made and how they where made. Firstly we will explain the testing

and implementation of our web site, secondly our User Agent, and thirdly the connection between our User Agent and the web site.

6.1 Implementation and Testing of the Web site

We have used HTML for layout and CGI-scripts to implement the functionalities of the web site. Now we will explain how we implemented the layout and then the engine of the web site. Implementing the web site was not really any problem, but the problems that arose can still be read about in Section 7.1.

6.1.1 Implementation and Testing of the Graphical Part of the Web site

The implementation of our web site was carried out without any problems at all. We used the web tutorial [4] and [5], which gave us all the information we needed. There is not much to say about the test phase other than that we opened our html files in Netscape to make sure the had the correct layout, if not, we merely changed our code until the web pages were satisfying.

6.1.2 Implementing the Web Site Engine

An engine, which is present on the web site to provide functionality, is needed. A few examples of these functionalities include scenarios like logging on to the web site or accessing personal information of a user. The functionality of each scenario was provided through CGI-scripts written in Perl. Read about CGI-scripts and why we have chosen Perl in Subsection 5.1.2.

What every CGI script does is generally to perform certain tasks based on information received either from the user, a file, information stored locally etc. and then generate a web page which the user can view as a result. To make the web site functionalities work we needed six CGI scripts and these are described below.

Log On to the Web site The user needs a password and a username to be able to log on to the web site. The web server contains a database (a text file) holding information about registered users. Each row in the database is equal to one user and has the format: username:password:first name:surname:country:SIP-contacts. When a user has typed his username and password in the text fields and clicks **Login**, the CGI script is started. The first thing that happens is that the script checks whether the username and password the user has stated, match against any of the rows in the database. If a match is found, **Home**, which is a new web page, is opened. At the same time the user's information is stored in a new text file. If no match is found then a user is given a page with an error message.

Look at User Profile When the user clicks the Your Profile-button a script is executing which in turn examines the file where the logged on user's information is stored. The information in the file is parsed and a new web page is generated containing that information in editable text fields.

Change User Profile When the user is present in the Your Profile-window and clicks the ok button a CGI-script is executed intending to change the user profile. First a check is made to see if something has changed in each field compared to the old information stored in the user file. If nothing has changed nothing is made and the home page is sent to the user. Otherwise the following is carried out:

- The script check if something is written in the change password field. If so, the script checks whether the same thing is written in Rewrite password.
- A new row that will be saved in the database is made, that is, the new profile. Each field in the profile on the page is a string. These strings are appended to each other into a long one.
- The database is opened into an array.

- The database is truncated and opened again in append mode.
- One row at a time is now read from the array and compared to the old profile. If a row mismatch with the old profile the row is written to the database.
- Now the file where the information about the user logged on is stored will be opened in write able mode, which in turn mean that the file will be truncated. The new profile is written to this file.
- The new profile is also written to the database.
- The home page is generated and sent to the user. The profile for the user has changed.

View the Address Book When the user click on the contact button a script is executed that opens the file where the user is stored. The information in the file is read and parsed to separate the SIP addresses stored. The name on the person who owns the address is contained in front of each SIP address. When the parsing process is complete, an array is created where each position in the array contains a name and the correspondent address. Finally a page is generated containing a table where each contact and address is listed.

Use SIP Address through the Address Book When the user is present at the address book page and clicks on one of the use buttons, a CGI script will be executed. The address next to the use button will be sent as a parameter to the script, which in turn generates the page **Make Call** with the sent address already filled in.

Send INVITE to the SIP User This script is executed when the user clicks send INVITE on the Make Call page. The field on that page (with the address) is sent as a parameter to the script. The sent parameter is parsed by the script and used to send an INVITE. How the actual implementation works can be read about in Section 6.2 and Section 6.3.

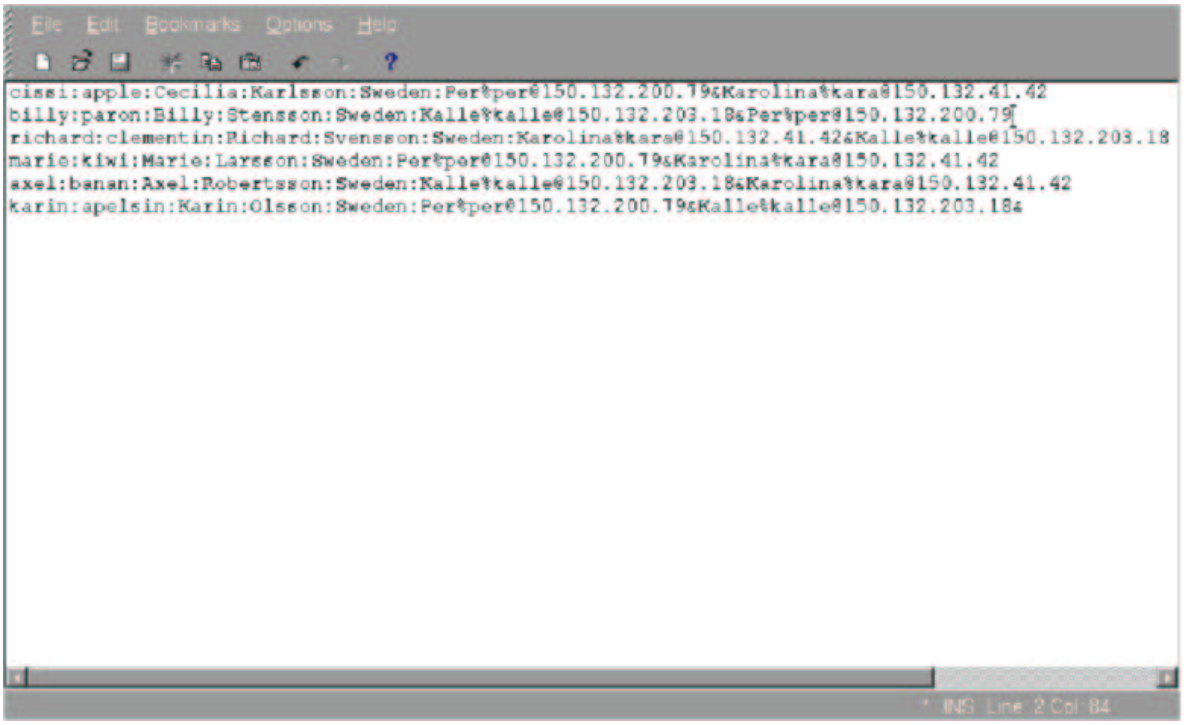

Figure 6.1: Example of a user database

6.1.3 Testing the Web Site Engine

To be able to test this web site in full, we created a database with made up users. This database can be seen in Figure 6.1. When we tested the procedure of logging in, we tried both right and wrong passwords to make sure everything was in order. There is not much more to say about the testing, other then, when testing the functionalities, we controlled if the correct actions where taken for each script, and if it were not, we located the problem and tried again.

6.2 Implementation and Testing the User Agent Using oSIP

This Section describes how we implemented and tested our User Agent Websip for oSIP. The implementation did not give us too much trouble since we, as said in Section 5.2, used

the User Agent JOSUA to see how an INVITE is supposed to be created and send.

6.2.1 Implementation of Websip

We started out by following the JOSUA code for creating and sending an INVITE. Then we started to create our own User Agent for sending an INVITE and it was done in the following steps:

Create and load configure file A configure file is created where useful information like username (for client), local IP and local port, is saved. The first thing that happens in Websip is that this configure file is opened and the posts is added to linked list. This part is something that we did not do ourselves, but copied directly from JOSUA, since we wanted the exact same code for this purpose.

Start the SIP stack, initiate oSIP and start the UDP layer The stack is started by calling the function oSIP_global_init(), oSIP is initiated by calling the function oSIP_init() and the UDP layer is initiated and started by calling the function udp_transport_layer_init(). The first two functions can be found in oSIP, and the UDP function is found in the udp layer. The UDP layer consists of two files which both we have copied from JOSUA. This because we wanted the exact same files.

Create a message for an INVITE First a new messages is created and initiated. The message has the form of a struct with members that forms the message header. Then the start line of the message is set: Method is set to **INVITE**, SIP-Version is set to **SIP/2.0**, and Request-URI is set to the SIP address at where the destination user is located. Status-Code and Reason-Phrase are both set to NULL, since you just need those in the response message. Then the header fields are initiated in the following way: Request-URI is set to the same as in the start line, To is set to the SIP address which was specified by the user, that is the same as Request-URI, from is set to the address from where the user is calling, Call-ID is set to local IP, CSeq is

set to **INVITE**, Max-Forwards is set to **70** and Via is set to a string that includes SIP version, transport protocol (in our case UDP), local IP and local port. When the startline and header fields are all set, the message is ready to be sent.

Send INVITE The next step is to send the INVITE, which starts with initiating a new transaction, then a new SIP event is created and a transaction id is set. Also the user decides what kind of transaction it should be. In this case it is a ICT. Now transaction_execute() is called, which is a function in oSIP, and it finds the right send method to call. When the method has been found it is called with the right callbacks for an INVITE. When this is done the INVITE message has been sent.

Create server The only thing a server needs to do from the beginning, is listen for new message. We found a small implementation of this in JOSUA that listens to the UDP layer. The implementation also includes parsing the message and distributing it to a correct transaction. With this implementation, all is set for receiving the INVITE.

Receive INVITE When the server detects an incoming message, the message is first parsed, and then analysed to see what kind of message it is. Further an event is created by oSIP, in this case an INVITE received. The event and the incoming old transactions are stored in a list in oSIP for future use. Now an IST transaction is created, since the message is an INVITE, and by calling ist_execute the event is carried out. In this case it calls the callback used when receiving an INVITE. Now the server prepares for sending a response to the client.

Create response to INVITE A response is created when the callback method for receiving an INVITE is called. Setting up all the necessary headers creates the message, and the most important one is the status code and the reason phrase that is to be set to 200 and OK respectively.

Sending the response By executing ist_snd_2xx(), instead of transaction_execute() as

when sending the INVITE, the message is sent by calling the callback function registered for this purpose.

Receiving response to INVITE (200 OK) When this message arrives, the osip_distribute_event does not actually create a new event, but it is up to the user of oSIP to manage the message and call the right functions. What should be done is that an acknowledgement of this 200 OK message should be sent. But we considered this to be enough since we now had sent SIP messages both from Websip to SIP and from SIP to Websip (SIP in this case refers to the simple SIP server that we implemented). What is done instead is that we check weather the message is an 200 OK response and then we make a call to the Perl stack to inform the Web user about the incoming response.

Note: We have chosen to implement both the server and our User Agent in the same binary. It is up to the user by giving input arguments to either start as a client or to act as a server. We have also chosen to just send one response to client from the server and that is OK. This because the purpose was just to see if we could create and send an response to an INVITE, because we did not have the time to do more than that.

6.2.2 Testing of Websip

For testing purposes we parted our User Agent into several parts, which are the same parts that we described in Subsection 6.2.1. We implemented and tested one part, and when it worked we started the implementation of the next part. For each part the testing was done the following way:

- The first step was to initiate and start the oSIP stack. Testing was barely needed, since we just had to call two functions in oSIP.
- Now we needed to create the message for an INVITE, and this part we tested by printing out several messages in all the functions, which were called. This way we could see which functions our program went trough and which it did not.

- When the INVITE message was created, the next step was to send the INVITE. This step included minimal testing and it was not any trouble since it was about calling one or two functions with the right, initiated parameters.

- The next step was register callbacks and start the UDP layer. Since we copied these from JOSUA the only thing that we needed to test was to check that the right send function was called for our INVITE and that the UDP layer had started. This we also tested by printing out messages in the functions that were supposed to be called.

- So now we could send a message for an INVITE but we had nowhere to send it to. Obviously some kind of server was needed, and this job was not hard at all since we found an implementation in JOSUA filling this purpose.

- The next thing we had to test was to see if we could send our INVITE from one computer and have our server receive it on another. This was actually a simple job cause with the right IP addresses in our configure file it worked right away.

- Further we had to make sure that our server called the right callback functions for sending a response to our client (the INVITE sender). At first, we had some problems with this, but we printed out messages in the functions which was called, and we could finally locate and take care of the problem.

- Sending the response was easy since we already knew how to set up a message. We simply created a new message, transaction and SIP event and used the built in function ist_snd_2xx(..) to send the response.

- Receiving the response was easy also. At first it was a bit confusing since the same transaction layer listener, which had parsed our incoming messages and distributed the events in the oSIP stack, did not do so with this message. The good thing was that we could still use the message parser to create an event and check weather it was an incoming response with EVT_IS_RCV_STATUS_2XX(..).

When we got this far the time did not allow us to move any further. So what our Websip can do right now is creating and sending an INVITE, the server receives the INVITE, and by printing a messages to standard out, we can see that the INVITE is received. Then the server creates a response and sends it back to the sender (it checks the From field) whom in turn parses and notifies the user of Websip.

Note: Websip has to be used from Perl since it includes calls to the Perl stack. Further the Perl program (CGI script etc) has to include the functions that Websip calls.

The possibilities to develop Websip can be read about in Section 8.2.

6.3 Implementation and Testing of the Link between the CGI-script and Websip

So far we have a web site and a User Agent for oSIP implemented, but for a web user to be able to send an INVITE from the web site to Websip, a link between Perl and C is needed as explained in Section 5.3. In this Section we will explain how we implemented the linking between the two languages with help from SWIG, and how we tested the same. Further we will describe how we connected the three parts and finally could send an INVITE from the web site on one computer, and receive the INVITE on another computer via a SIP server.

Please make sure you have read Section 5.3 before continuing.

Wrapping our User Agent was not hard at all. The wrapping was made according to Figure 5.12, shown and explained in Chapter 5.3.2. The following SWIG interface file was designed:

```
%module websip
%{
#include <websip.h>
%}
int init(char* SorC);
```

```
void call_printUID();
%typemap(perl5, in) SV *
{
$target=$source;
}
```

%module websip tells us about the name of the module. A file called websip.pm will be created when SWIG executes the wrapper generator for the interface file. This .pm file is actually a Perl module that loads the wrapped C/C++ code. The Perl module can then be included within, and loaded from a Perl script, which in turn can access our User Agent through the functions defined in the interface file. One string parameter can be sent to the function int init(char* SorC); defined in the interface, and in the User Agent. By sending the string **server**, a server is started. If any other string is sent an INVITE is sent to the specified address given in that string, and after the INVITE is sent the User Agent is listening for any incoming SIP messages.

The call_printUID() is a function that we want to call when receiving an INVITE. In this function we call a Perl function that exists in our CGI script that in turn notifies the web user through server push combined with a print statement (this will be discussed later). Unfortunately we have to call the Perl function through Perlcall, which is a way of calling Perl functions from C. This is only a temporary solution and should be replaced by wrapping functions to register callbacks from the target language. Sadly, since SWIG does not know how to wrap function pointers, this also means that you cannot send a Perl function as an argument. There are ways to work around this, but since we had to be experts on SWIG we have chosen not to. We could have chosen to ignore the problem of calling a function in Perl from C, since writing a simple printf-function in the C stack will give the same result in the CGI script as a print statement in Perl does. The purpose for not doing so is that we want our User Agent to be separated as much as possible from the web site implementation. We think that our User Agent should work as an object, which

offer services without the involvement of other languages. Unfortunately coding a Perlcall into the User Agent is necessary since wrapping callbacks is not yet supported by SWIG. If this was possible, and it is most likely to be in the near future, due to the demands, put on this functionality, that we have seen on the SWIG mailing lists ([9]), then the User Agent could work as a completely isolated object.

The reason why, we want to register callbacks from our target language that we use in our web site, is that if the developer of the web site wish to change the language to Java, TCL, Python etc, instead of Perl, and/or change the CGI script to JavaScript, implement Java applets etc, he would only have to change the language itself and would not have to worry about rewriting the User Agent. This gives the developer of the web site more freedom of choice developing his site, and ultimately this leads to better layout and functionality on the web site.

The %typemap function is a bit hard to explain but it takes care of type-conversion when calling a Perl function from C/C++. It is needed when calling Perl functions from C/C++.

Now we were able to call the stack from a CGI script to access our User Agent functions. To get feedback regarding the SIP session we have chosen to use server side push technique in our CGI script. This means that the connection between the page on which the CGI script is active and the user is kept open. The servers choose when to send information update to the user. The user of the web page is constantly listening for any web pages that might be sent to him. Obvious disadvantages with this technique is that the constant open connection between the Websip server and the user consumes a lot of bandwidth between the two, and it also puts a load on the Websip server and the Websip user since the connection has to be "kept alive". Other techniques that might be used instead of this one are discussed in Chapter 8 - future development.

The technique for server side update (or server side push) is simple. By writing a special delimiter (written: –anyspecialstring) in your script you tell the server when to send the

next page to the user. E.g. you could have a loop that sends a page every second. In this loop this delimiter would be written together with the page you want to send (and any other extra functionalities like a sleep-for-one-second function etc). In this way the user can be kept informed on how the SIP session is proceeding because each event in the SIP stack has a callback attached to it. Those callbacks can in turn call a Perl function in the CGI script, which then tells the server to send a page to the user. For example: an ok response to an INVITE is received at the web page intended for the user. The callback function for this INVITE is called which in turn tells the CGI script to send an update to the user.

Since our server can receive and respond to an INVITE, we where able to conduct a test on a full-scale scenario. The procedure of our test was to:

- Log on to the web site
- Go to address book
- Choose address to send to
- Click use
- Click send
- response is received from the SIP server and seen on the web site.

This leads us to conclude that there is, at least, one satisfactory technique to use SIP through html. Thereby, another part of our main goal with this project was achieved.

7 Possibilities for the Web User to Receive Incoming Calls

In our requirements specification the last thing we MUST do was to investigate the possibilities for a web user to receive incoming calls from SIP users. This chapter describes two

results of our investigation. Non of the solutions we have suggested have been implemented.

The problem was that a web site, written in HTML, is static, and needs to be reloaded for the web site user to get new information from the server. So a function to inform the web site user about incoming calls was needed. An example of a normal call scenario is shown in Figure 7.1, and our case, when calling the web site, is shown in Figure 7.2.

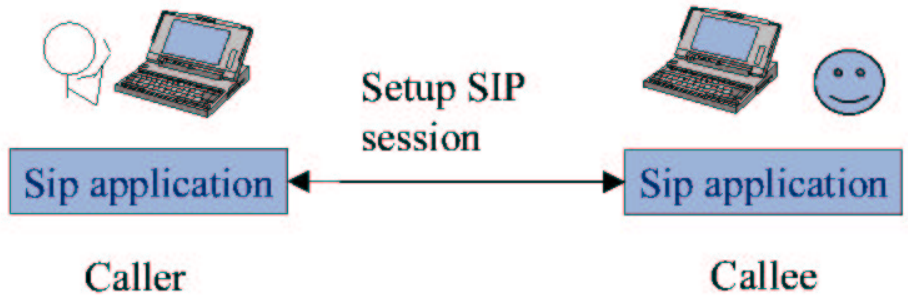

Figure 7.1: An example of a normal SIP session. The callee application and the SIP functionalities is present on the same computer

Through new techniques that have been developed throughout the years like Java, CGI, Dynamic HTML, .NET, XML, and other programming languages or scripts, the Internet has been taken to a new level. There are many ideas to consider, and below follows two of them:

- As shown in Figure 7.3, a small Java applet could be loaded when the user logs on

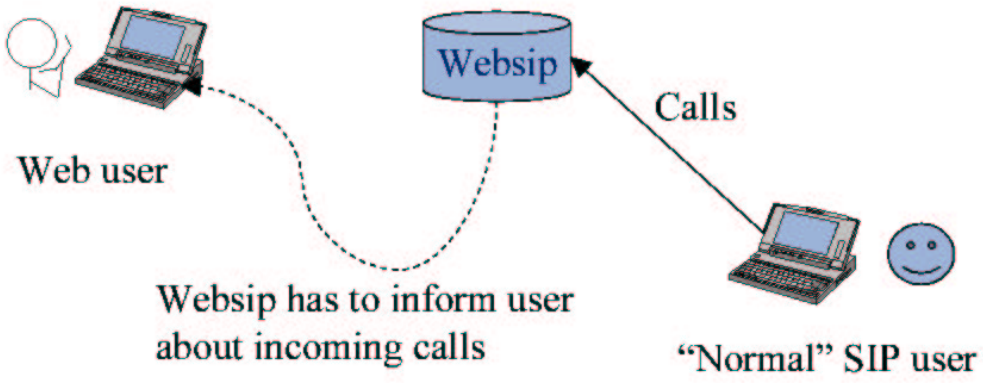

Figure 7.2: Scenario where a normal SIP user tries to call a Websip user. The SIP functionalities is present on the web site

to Websip. This applet could in turn be used to inform the user of incoming calls. The applet would accept incoming messages from the server and should not have to have a constant open connection. In turn it could also inform about or redirect the user to an ”incoming call” web site where he can handle the incoming calls. Applets are able to handle these situations. To support the fact that applets could work in this way please refer to [14] that shows a simple example of a client-server model in Java (which applets are written in). Refer to the standard Java Messaging Service (JMS) [17] if you want to know more. Please also refer to Section 8.2.1, which discuss pushlet techniques.

- A window that the server pushes information into could be used to inform the user about incoming calls. This window could also be used to redirect the user. We know

this idea could be implemented since we have successfully used server push technique through our CGI script to inform the user about incoming SIP messages. One could simply write a link to the site that the user should be redirected to (although he has to click on it himself). As of this writing, the technique we used only works with Netscape browsers, and requires a constant open connection between the server and the client, and is therefore, not recommended.

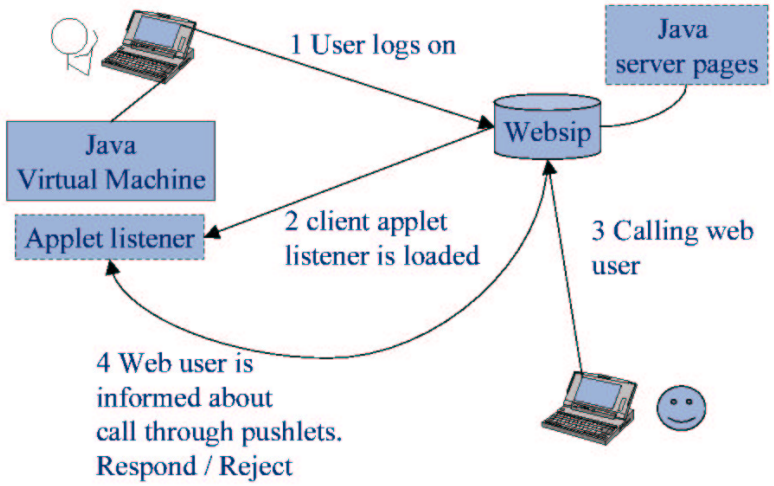

Figure 7.3: Client being notified of an incoming call through Java

Note that the server would take care of all the incoming SIP calls and distributes them to the users.

8 Problems

This chapter will summarise problems involved in this project. The end of this chapter discusses what we could have done differently.

8.1 Problems that have Occurred

In this Section we will talk about the bigger problems we had when developing our application, so readers that are interested in further developing, can see what also may be a problem in the future.

8.1.1 Problems with the Web Site

As said earlier, the design and implementation of the web site, did not give us much trouble. With the graphical part, we hardly did have any problems at all. With the web site engine we had some minor problems and one big. The big one has to do with how many users that can be logged on at the same time to the web site. At present only one user can be logged on at the time (see why in Subsection 6.1.2), and this is of course a big problem, cause it is not how a web site should work. We have not been able to solve this problem during our working period though and leave it to future developers.

There is also one problem that has to do with the graphical part of our web site, but is caused by the CGI-scripts. As mentioned in Section 5.1, we parted our web site in frames, and when the user is clicking a button in the left frame the correct side should open in the right frame. The problem occurs after a button that is connected to a CGI-script is clicked. When the user clicks a button the next time the new side is opened in a whole new window, and the frames are all gone. This is a problem we have not been able to figure out, all we know is that it has to do with the CGI-scripts, cause the problem does not occur without them. So this to we leave to future developers to solve.

8.1.2 Problems with Websip

When implementing our User Agent Websip, we did not have to many problems, since as mentioned, we used JOSUA for help. The smaller problems we did had were solved quickly, and does not seem relevant to bring up in this report.

8.1.3 Problems with Linking C and Perl

It was this part that gave us the big problems, and made our time schedule to crash. The problem was not the program SWIG itself, cause it worked fine for smaller examples, but the size of oSIP. As mentioned earlier, oSIP is not big, but it is not small either, and it includes many files. So the problems occurred mainly when we tried to link all of these files with the SWIG specific files. And if the linking went through, the next problem was to manage to call a function in Websip from a Perl program, and then it could not find all the files in oSIP. So obviously there was still a problem in the linking.

Short about types of linking: **Static:** Libraries are copied into each target of the project that has a dependency on them. This often results in huge executables and much memory consumption, since the whole executable is loaded into the memory.
Dynamic: Each object is linked to its particular dependencies. The good thing about dynamic linking is that objects are loaded into memory as they are needed, and they do not need to be copied. The bad thing might be that linking faults can go undetected since the libraries are loaded at runtime (with ld.so.1 in Solaris).

Several weeks where lost digging around after the correct way to link our CGI-script with the oSIP library using SWIG. We tried different methods like static linking (which should only be used when dynamic linking is not available though), and dynamic linking with shared objects. We discovered that there were undefined references to functions when linking the objects together (static linking), and that there were undefined references at runtime when using dynamic linking. We tried to find all the necessary libraries for static linking but we only ended up needing more and more libraries (you add one, and then

you find that that you need other libraries and so on). The dynamic linking seemed good at link-time. But at run-time we realised that we lacked just as many dependencies as in static linking.

After asking a few people directly, and through mail, we finally solved the problem through a mailing list. The ones, who helped, told us that we should use the flags -W1 and -G. -G is used to create shared objects, and -W1 is used for error handling. The -G option is used in SOLARIS and -shared is used in LINUX. We thought we could use -shared for SOLARIS too but we were wrong. After using the -G option we could finally link.

8.2 What we could have done Differently

If we were given the possibility to do this project all over again, there are some things we would do differently. One thing is that we would start to build our own User Agent for oSIP right away, instead of take the simple way, and use the example already included in oSIP. This example was obviously too big for SWIG. We would also try to ask around a lot more for help with the linking problems we had with SWIG (and other problems). We did ask a lot of people, but the help came from a Ericsson AB employee when we finally solved the linking problem, although it was through a mailing list.

9 Possible Future Developments

In this chapter we will suggest different kinds of developments that are possible for our application. For some of our suggestions we discuss possible solutions, and for some we do not. We will start with functions that could be added to the web site. Then we will discuss how the whole system can be developed, and for some of our suggestions, we will also discuss different solutions.

9.1 Development of the Web Site

Since our goal on this bachelor's project was to examine if it was possible to set up a connection between the two protocols HTTP and SIP, our web site has far from all the functions it could have. Below there is a list of different functions that could be added. We will not suggest any solutions though.

- A better login system where new users can register.
- Make it possible for a user to add new SIP contacts to its address book.
- A button which the user can click on to answer incoming calls at the **Make Call**-window.(see also Section 8.2).
- Information at the **Make Call**-window about missed calls, answered calls etc.
- More information about the user in the user profile, like phone number, birth date etc.
- A SIP address for every user like usernamewebsip.com.
- A local search engine where a user can search for other users.
- A square at the **Your Profile**-window which the user can mark if he wants his profile to be visible for other users.
- Some kind of support like a FAQ.

9.2 Development of the Whole System

There are still several problems to consider. According to Figure 1.2, information should be sent to the Websip server about the desired changes to the session. The user can merely select his options on the web site. For example if he wants to add or remove another user, or add or change the service (e.g. add a video display). Possible help programs

(for running video conferences etc) that could be downloaded from Websip would inform Websip about how the session is to proceed (through built in functionalities). If the user closes his help application a message could be sent to Websip, or perhaps the application could send frequent messages to the Websip server informing Websip about its presence (a "keep-alive" connection). The problem arises if the user has an application independent of Websip using that to transfer session-data to and from the other SIP user. If the Websip user closes his application the session ends without Websip being notified (resulting in an incorrect SIP session). Of course the other user will notice and his SIP User Agent will take the correct action (in the case where the SIP user is not a Websip user).

Possible ideas about solutions to this problem:

- Perhaps the user programs could be started inside of another window like an ActiveX window. When this window closes it would inform Websip about the change.

- Description files (like real media files) could be dynamically sent to the user from Websip when a data-transfer (media, ftp, etc) session commence. The user would click on those files, and another window would pop up with his own program running inside of it. When this window closes it would in turn inform Websip about the change.

9.2.1 Improving the Interaction between the Web Site and the User

At current state a constant open connection between the web site and the user through CGI-html server-side push technique is used. This technique results in a lot of open connections and the hogging of recourses if Websip where to be used by many people.

Suggestions to improve server to client communication follows:

- A more sophisticated and elegant technique (which unfortunately we could not test due to lack of recourses) can be used pushing the information to the user from the server without having to have an open connection ,although this demands the use

of a light weight applet. This technique called Pushlets [15] demands that the client can run DHTML (Dynamic HTML), and JavaScript (which is a built in function in many browsers today). Java (which is commonly installed on most computers [13]) is also required.

- Server-side callbacks. The server can call back a Java-applet client using either CORBA (Common Object Request Broker Architecture) [19] or RMI [18] (Remote Broker Architecture). This solution can be read about further in [18]

- Messaging following the JMS. Could be used within applets to create a dialog between the client and the Websip. Again the client would have to have Java installed on his computer.

9.2.2 Considerations when Expanding Websip

At present state, our User Agent is just a test prototype, and needs to be extended to suit future web site services.

Problems that needs to be solved:

- As seen in Figure 9.1, the oSIP stack needs to have a SAP (Service Access Point) where users can retrieve their own User Agent service. At the moment the send CGI script initiates the stack and can access its services. But what if the stack is already running? Indeed, it should be since it should act as a server. How does the CGI script / script get access to the SIP server (oSIP server) if it does not initiate its own copy?

- Management of users in the system needs to be built in. oSIP is reentrant, and supports multiple users, which is good.

- One port should be used for any incoming calls, so that users have constant port numbers in their addresses. The User Agent should, when receiving a call, check

weather the user is logged on or not and take the appropriate action.

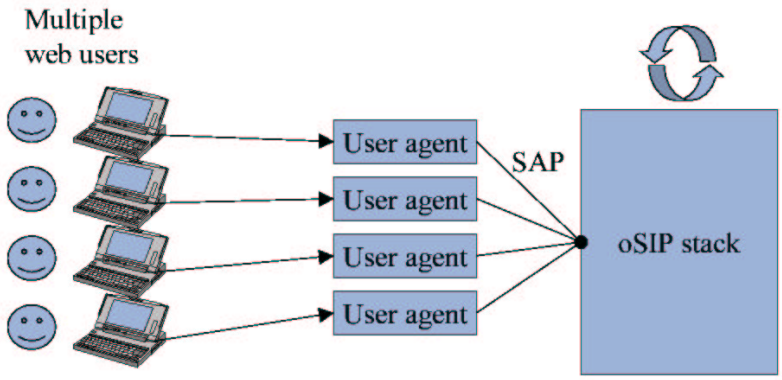

Figure 9.1: Multiple users accessing a SAP in the oSIP stack

9.2.3 Other Alternatives to Our System

Ericsson AB wanted to examine the possibilities to communicate between SIP and HTTP. That is, a client should be independent of SIP. One obvious option to this system is to put the SIP functionalities on the client instead by letting the web user download and run a SIP application from the web site. This would relieve the server from a lot of traffic and developers would have fewer problems when developing the system. The application could be an applet that is loaded when the user logs on to the web site. The web site would perhaps still offer services, like answering machines for each user, mail services, etc. A web site that uses this idea, although it does not include SIP is [16]. On this site an

applet is loaded onto the users computer, which includes all the needed functionalities. One advantage of our system could be that the clients could use existing software on their computer to manage the data-transfer. It is really a question on how to divide the functionalities and workloads between the web service and the web user, and which techniques that are available to do so. Java and .Net are both interesting when investigating this.

10 Conclusions

The work conducted doing this project has resulted in new ideas about extension possibilities developing a future web site intended for multimedia conferencing with SIP. Also it has been shown that it actually is possible to develop such a web site. Or at least that using SIP from HTTP is possible. Our conclusions can be seen in the list below.

- We can, with the completion of this project conclude that create a link from HTTP to SIP is possible in at least one way for outgoing calls. Although we did not complete all the signaling, we managed to transfer messages between the user of Websip and a SIP server.

- We have also discussed several possible solutions for a web user to receive an incoming call from a SIP user and concluded that this should also be possible.

- Selecting an appropriate scripting language or usual language for the web site including Perl, Mzscheme, Python, TCL, Guile, Ruby, and Java is made possible through SWIG. This is very useful since the developer can use the User Agent from such a variety of languages. It gives the developer a lot of flexibility when developing Websip or any similar systems that makes use of oSIP or a SIP stack written in C/C++ further.

- The MUST part of our requirements was not totally fulfilled, since we have not sent all of the messages for a SIP connection. Also, we did not manage to transmit a confirmation message from SIP user to web user when the connection was established. The reason why we did not manage to do this is because we ran out of time because of problems that occurred. These problems are described in Chapter 8 and in appendix C the workflow is described. A web user is though, able to make an outgoing call to a SIP user from the web page, and there is a connection established between them.

References

[1] Aymeric Moizard *The GNU oSIP library*
http://osip.atosc.org Read: 020124 Updated: 020517

[2] Mark Overmeer *Perl/Tk saves your favourite tools*
http://perl.overmeer.net/yapc2001/index.html Read: 020510

[3] Philippe Chiasson *Pathologically Polluting Perl with C, Java, and other Rubbish using Inline*
http://www.extropia.com/press/2001/philippe_lugs.html Read: 020510

[4] Dave Kristula *HTML - An Interactive Tutorial For Beginners*
http://www.davesite.com/webstation/html Read: 020228 Updated: 020605

[5] Dave Raggett *Getting started with HTML*
http://www.w3.org/MarkUp/Guide Read: 020229

[6] Dean Roehrich *perlxs - XS language reference manual*
http://www.cs.technion.ac.il/Manuals/perl/perlxs.html#NAME Read: 020321

[7] Mark Overmeer *Perl/Tk saves your favourite tools*
http://mark.overmeer.net/yapc2001/48-0/index.html Read: 020321

[8] David Beazley *SWIG official homepage*
http://www.swig.org Read: 020315

[9] David Beazley *SWIG mailing list*
http://www.swig.org/mail.html Read: 020315

[10] Mattias Ranbro *CGI/Perl*
http://www.it-mattias.com/cgi_perl.html Read: 020308

[11] IETF *IETF - The Internet Engineering Task Force*
http://www.ietf.org Read: 020124

[12] AAdvantage Systems *Questions and answers about Windows Terminal Server*
http://www.aadvantage.net/citrix/faq.htm Read: 020503

[13] Computer Solutions Kauai *How to Enable and Update JavaScript in Netscape and Internet Explorer*
http://www.computersteve.com/javasetup.html Read: 020506

[14] Java Sun *A Simple Network Client Applet*
http://java.sun.com/docs/books/tutorial/applet/practical/clientExample.html
Read: 020507

[15] Just van den Broecke *Pushlets: Send events from servlets to DHTML client browsers*
http://www.javaworld.com/javaworld/jw-03-2000/jw-03-pushlet_p.html
Read: 020510

[16] ICQ inc. *Web based ICQ*
http://lite.icq.com
Read: 020510

[17] Java Sun *JAVA MESSAGE SERVICE API*
http://java.sun.com/products/jms/
Read: 020510

[18] Java Q&A Experts *An in-depth look at RMI callbacks*
http://www.javaworld.com/javaworld/javaqa/1999-04/05-rmicallback.html Read : 020510

[19] Object Management Group, Inc. *CORBA BASICS*
http://www.omg.org/gettingstarted/corbafaq.htm Read: 020513

[20] Fred Halsall, 2001 *Multimedia communications (Applications, Networks, Protocols and Standards* Addison Wesley Longman, Inc.

[21] James F. Kurose Keith W. Ross, 2001 *Computer Networking (A top-down approach featuring the Internet)* Addison Wesley Longman, Inc.

[22] Rafe Colburn, 1998 *Lär dig CGI-programmering på 1 vecka (Swedish edition, translated by Björn Mattsson)* Pagina Förlags AB

[23] Niklas Frykholm, 1997 *CGI-programmering* Pagina Förlags AB

[24] Eleftherios Gkioulekas *Introducing the GNU tools*
Web page: ftp://ftp.ugcs.caltech.edu/pub/elef/autotools/node64.html
read 020510

A List of Terms

callbacks A callback function is a function in your program, the address of which you pass to another code context. This context calls back your function when it wants to.

HTTP HyperText Transfer Protocol, an application protocol for requesting web pages.

ICT Invite Client Transaction, an INVITE message sent from a SIP client.

IP Internet Protocol, a protocol that provides for transmitting blocks of data called datagrams from sources to destinations.

IST Invite Server Transaction, a response from a SIP server for an INVITE message.

JOSUA Jack's Open Source User Agent, a User Agent for oSIP implemented by Aymeric Moizard. See more at [**?**, oSIP]

NICT Non-Invite Client Transaction. A Non-INVITE message sent by a SIP client.

NIST Non-Invite Server Transaction, a response from a SIP server for a non-INVITE message.

oSIP open source SIP, a help library for making implementations of SIP applications easier.

RTCP Real-Time Control Protocol, an application protocol for multimedia transactions.

RTP Real-Time Transport Protocol, an application protocol for multimedia transactions.

SDP Session Description Protocol, an application protocol that transmits information about the media streams in a SIP session.

SIP Session Initiation Protocol - An application layer protocol which initiates a multimedia session between two or more parties.

SIP address The address where a certain SIP user is located. It is formed like an email address e.g. mewebsip.com

SIP connection A connection between two users made using the SIP protocol.

SIP session A call from the beginning to the end involving two or more SIP users.

session time The time for a SIP session.

signaling Establish, supervise and end a session between two specific users.

SS7 Signal System 7, a signaling system.

SWIG Simplified Wrapper and Interface Generator - A software which can link certain script languages with C and C++.

TCP Transmission Control Protocol, a transport protocol for transporting packages.

UDP User Datagram Protocol, a transport protocol for transporting packages.

UML Unified Modeling Language, is used to design software application before coding.

User Agent A program that performs what the user of the program tells it to do.

Windows Terminal Server Windows Terminal Server is a product that adds UNIX-like multi-user capabilities and support for thin-client Windows-based Terminals, to Windows NT Server 4.0 operating system. See [12]

B GNU Build System

This appendix introduces a tool that helped us during the implementation of our User Agent Websip.

B.1 Introduction

The GNU build system is a comprehensive toolkit that manages the configuration, makefile generation, structuring of source code, version handling, testing, installation, and packaging of software systems. Although the GNU build system is intended for the distribution of free

software, we have chosen to use it since oSIP was made with it and thereby it is also easier to include oSIP into our software packages if needed. Another reason is that it seemed to be the best tool around for managing makefiles and configuration scripts on a high, easy to understand and maintain level.

B.1.1 Usage

Instead of writing for example Makefiles at a normal level the user of the GNU build system writes templates that then is processed by automake (a tool in the GNU build system).The user simply defines which files to include in the project, which to make libraries of, what files that are going to make up the executables in the software etc. There are also several configuration tools that analyses the system and helps to customise the makefiles of the project even further. The configuration files can also be used to check weather the software is compatible to be installed on the system (e.g. if all necessary libraries that are not included in the package itself exists). The GNU build system refers to four different packages:

1. Autoconf produces a configuration shell script named configure from a template named configure.in . This script probes the current platform for portability information used when creating makefiles configuration header files, and other files related to the application. When this is done it continues to generate these files with help from templates written by the user and the information collected during the probe.

2. Automake produces makefile templates Makefile.in used by autoconf from higher templates Makefile.am written by the user. One advantage of generating makefiles is that it keeps the same structure, making the makefile more readable to everyone familiar with the automake tool.

3. Libtool is used to build shared libraries and compile position independent code. It can be used by automake, automating the process of building libraries even further.

4. Autotools Generates boilerplate files which can be used developing software.

This build system helped us a lot, and saved a lot of time since we did not have to write huge makefiles and since configuration and installation was more automated. It saved us the repetitive task of writing Makefiles, and gave us a nice system to handle our software.

If you want to dig deeper into the GNU build system (recommended), please refer to [24].

C Workflow and Time Schedules

This appendix describes our workflow in this project, and shows our first and our last time schedule. We think this could be interesting for future developers to read and see, so maybe they will not make the same mistakes as we did.

C.1 Workflow

In the beginning of this project the work went pretty smoothly. One little problem we did have though, was that the kick-off meeting was not held until the fourth week, for several reasons. After the meeting, we could work more efficient though, since it was more clear exactly what we were supposed to accomplish.

The first four weeks we were making clear what we were supposed to do and based on that we worked out a requirements specification and a time schedule. We also took this time to read and learn about SIP and oSIP. We downloaded oSIP, compiled it and tested the example that were included. In working week five we started to design and implement our web site. The graphical part did not give us any problem, but in the implementation of the CGI-scripts we had some problems. These problems are documented in Section 8.2. Even though, we managed to finish the web site implementation in time according to our time schedule and requirements specification. Still there were though some problems left to solve, and these we worked on now and then, a couple of weeks longer.

About the same time we started on our web site, we also started to examine which different possibilities we had to link C and Perl. We found SWIG, tested it and manage to make a simple example work. Big problems occurred though, when we tried to use SWIG with the included example in oSIP (see which problems in Subsection 8.1.3). We struggled with this for several weeks, and we could no longer follow our time schedule. For a while we also tried to write the Send-script on the web site in C, so that SWIG would not be needed, but in the end it just gave us the same problems as SWIG did. Finally, after a meeting with our advisor, we decided to skip the included example in oSIP, which is very big and includes more than we needed, and write our own User Agent. This User Agent should only be very simple, with creating and sending an INVITE, and creating and sending a response to the INVITE, since we as for now did not have time for more.

The building of the User Agent went pretty smoothly, since we had the help from the recently developed JOSUA, and as soon as we had a contact between two computers, we started the linking with SWIG again, and there were all the problems again, but after some work we finally made it work. The next thing we had to do was connecting our three system parts, that is send the INVITE from our web site, receive it on another computer and then sending a response from a server to that INVITE. Then we could see the connection between our web site and a SIP user on another computer. Our time had now run out, so we did not manage to fulfil all of the MUST requirements in our requirement specification, but at least we manage to connect a web user to a SIP user, and in the end we also found the time to examine different possibilities for the web user to receive an incoming call from a SIP user.

C.2 Time Schedules

This section shows our first and last time schedule. We thought it would be interesting to see the differences between what we thought we would have time for in the beginning, and what we really did had time for in the end.

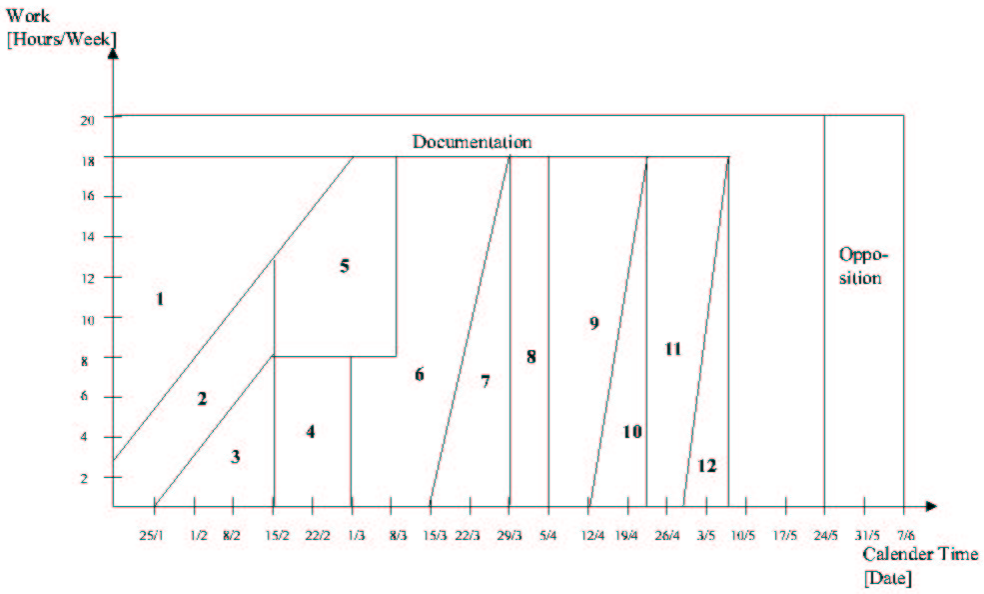

Figure C.1: Our time schedule in the beginning

Figure C.1 shows our first time schedule. As can be seen we planned in our whole requirement specification.

Figure C.2, which is our last time schedule, shows that we fulfilled all necessary requirements as specified in the MUST part of the requirements specification. However, all the requirements included in number 6 where not fulfilled. Refer to the numbered list below when looking at Figure C.1 and C.2.

1. Read about SIP and oSIP.

2. Finish requirements specification and delimitations.

3. Design web site.

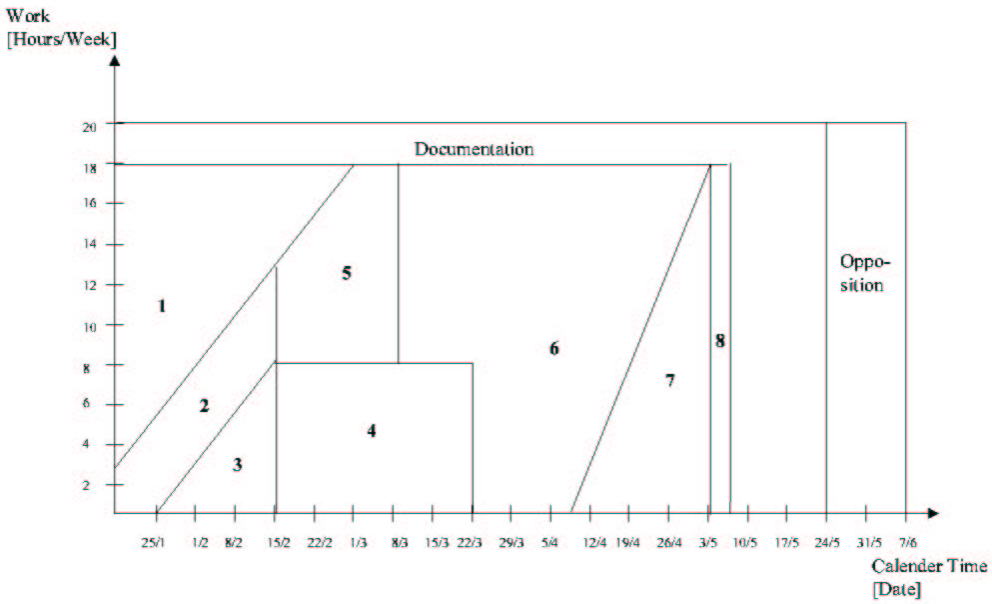

Figure C.2: Our time schedule in the end

4. Design link between Perl and C.

5. Implement web site.

6. Build User Agent and connect web user to SIP user.

7. Testing

8. Consider solutions for a SIP user to connect to a web user via the web site.

9. Implement one of these solutions.

10. Testing

11. Implement a media transfer.

12. Testing

From these time schedules, we have learnt, not to be as optimistic when scheduling a project. The next time we will try to specify a more realistic time schedule.

D Other Options to SWIG

This appendix describes the options that could be used instead of SWIG.

XS The following quote is taken from [6].

> *XS is a language used to create an extension interface between Perl and some C library that one wishes to use with Perl. The XS interface is combined with the library to create a new library, which can be linked to Perl. An XSUB is a function in the XS language and is the core component of the Perl application interface. The XS compiler is called xsubpp. This compiler will embed the constructs necessary to let an XSUB, which is really a C function in disguise, manipulate Perl values and creates the glue necessary to let Perl access the XSUB. The compiler uses typemaps to determine how to map C function parameters and variables to Perl values. The default typemap handles many common C types. A supplement typemap must be created to handle special structures and types for the library being linked.*

Advantages:

- Platform independent.
- C code is separate from the script code that is good for large projects.

Disadvantages:

- This toolkit demands a whole lot of job to learn since it is a whole big language of its own.
- The program is difficult to understand (We think so).
- It is a low-level language compared to inline and SWIG (see below).

Inline C The following quote is taken from [7].

> *Inline is a recent module, which simplifies the creation of C-libraries in conjunction with your Perl-program. The difference with XS is that where h2xs starts with setting-up a compilation environment which you then modify to your needs, Inline only creates that setup on the moment that you know what you have.*

and

> *Usually just write the function as you would do in C, and leave-out the #includes and it works! Inline compiles the code when it is seen first. Then it is slow, but the next time the same code is detected, it is not recompiled, but a cached version is used.*

Note: h2xs is a toolkit that generates XS templates used for wrapping your software.

So the C-code is actually written straight into the Perl code. Inline seems easy to learn but limited in its functionality.

Advantages:

- Works on both UNIX and Windows.
- Easy to learn.
- Little work needed to get going.

- Support for more than Perl.

Disadvantages:

- The code is compiled when first seen and then cached, which makes it slower, compared to XS and SWIG that runs compiled code straight away.
- It is Inline. The code is mixed up with your target language, for example Perl, which can make less clear.

[2] includes a comparison of Perl and TCL, a description of how to connect a C program with inline, and a comparison of XS and SWIG.

[3] Puts Inline in focus. It also has a comparison of SWIG and XS.

Refer to the above mentioned to get a more in depth information.