XML Based Service Provisioning in Converged Voice and Data Networks

Sertac Cetiner

Preface

This master thesis has been written at the University of Ulm, Germany. The thesis consisted of a project work at Siemens AG in Munich, Germany. Three groups were involved in this project. Each group had different locations, and different tasks to work on. The first group was responsible for the design and the development of the SIP Server. The complete SIP server was developed and integrated into the CPL environment by that group. The second group was responsible for the design and the development of the CPL User Editor, whereas the third group was responsible for the design and the development of the CPL Engine and the coordination of the project.

Two persons formed the third group: The author and Giovanni Benini (Siemens AG, Munich), the supervisor of the thesis. Mr. Benini coordinated the project, while the author was responsible for the design and the development of the CPL Engine.

I would like to express my respect to the University of Ulm, especially to the Communications Technology Program for providing me the opportunity to do my MSc. at the Univeristy of Ulm.

I would like to thank Giovanni Benini for his invaluable support during all the project work. His friendly personality helped a lot completing the project successfully.

Thanks to Prof. Bossert for his interest and guidance.

And, thanks to Hermann Granzer, Achim Fahrner, Zonya Dengi and Nashwa Abdel-Baki for reading and commenting on the thesis.

Finally, thanks a lot to my family for their unconditional support.

List of Abbreviations

ACE	Adaptive Communication Environment
API	Application Programming Interface
CORBA	Common Object Request Broker Architecture
CPL	Call Processing Language
DCE	Distributed Computing Environment
DOM	Document Object Model
DTD	Document Type Definition
HTML	Hyper Text Markup Language
IN	Intelligent Network
IOR	Inter Object Reference
IOR	Inter Object Reference
IPX	Internetwork Packet Exchange
LDAP	Lightweight Directory Access Protocol
ORB	Object Request Broker
RFC	Request for Comments
RPC	Remote Procedure Call
SAX	Simple API for XML
SGML	Standard Generalized Markup Language
SIP	Session Initiation Protocol
TAO	The ACE ORB
TCP/IP	Transmission Control Protocol/Internet Protocol
UDP	User Datagram Protocol
URL	Uniform Resource Locators
W3C	World Wide Web Consortium
XML	Extensible Markup Language

Table of Contents

1 Introduction

In today's world, there are mainly two types of communication networks: circuit-switched networks and packet-switched networks. The current telephone networks are mostly based on the circuit-switched networks, whereas the Internet is mainly based on the packet-switched networks, which are also called IP networks. However, there is a strong tendency to combine both of these networks, which points to the direction that the IP networks are going to replace services provided by current telephone networks. This would eventually mean that IP networks might replace the telephone networks, in the future.

Following are some reasons why IP networks seem to replace the circuit-switched networks:

- First of all, the IP networks provide cheaper communication. Considering that the Internet access is nearly free, the cost advantage of IP networks gets clearer [25].
- Secondly, IP networks provide the ability of integrating the data and voice applications, and even some other applications, like video-conferencing, integrated voice mail, e-mail, and the like [26].
- Another important reason is that IP networks allow open implementation of end systems. With a reasonable programming knowledge everybody could implement an end system for IP networks. In the classical telephony end users cannot implement any end system, but have to use whatever provided by the service providers. [27]

Beside these attractive advantages, IP networks face an important problem compared to the circuit-switched networks [25]. The circuit-switched networks provide a guaranteed level of voice quality, while IP networks cannot guarantee certain level of voice quality, since the data transfer is based on random access in the IP networks. However, with ongoing research in this field, the voice quality requirements are going to be met in coming years.

Internet Telephony is defined as the provision of telephone-like services over the Internet. Some consider it as the next stage of the telephone network and the first incarnation of the long-held goal of an "integrated services" network [20]. The integrated services here means, for example, the integration of data delivery, voice delivery, video conferencing, and other possible services provided by the Internet and the telephone networks.

Although the Internet Telephony offers very low call costs, the current pricing schemes of the classical telephony are getting lower every day, and there may not be any significant difference in the future [25]. Because of that, the additional services offered by the Internet Telephony achieve a bigger importance to be able to beat the classical telephony. The more advanced features Internet Telephony offers, the more customers switch from the traditional telephone networks to the Internet Telephony.

The purpose of this thesis was to design, implement, and evaluate the creation of additional services using Extensible Markup Language (XML [3]) as the data definition format. The Call Processing Language (CPL [6]), developed by the Internet Engineering Task Force (IETF), is the XML based service creation language for the Internet Telephony. As a result, the CPL is going to be the main focus of this thesis to create the additional (intelligent) services for the Internet Telephony.

Without using any additional services, simple Internet Telephony provides the ability to connect two or more call requests through the IP networks. However, the CPL offers

additional services, such as *forward on busy, redirection, decisions based on the origin of call, generating log of calls, e-mail notification, rejection, call waiting, multiple line signaling, and so on.* These services are expected to make the Internet Telephony more attractive and functional for the end users.

The Call Processing Language (CPL) is evaluated in this thesis for creating the additional services, as mentioned above. Session Initiation Protocol (SIP) [4], which is again developed by the Internet Engineering Task Force (IETF), is used as the signaling protocol for the Internet Telephony. However, basically, these services could also be applied to the H.323 [21] protocol, which is developed by the International Telecommunications Union.

Organization of the thesis is as following: Chapter 2 gives the fundamentals underlying the project work. Chapter 3 explains the details of the Call Processing Language (CPL). Chapter 4 presents the architecture of the project, and details each component in the architecture. Chapter 5 gives some information about the tools used to implement the project. Chapter 6 goes into the details of the implementation. Results are discussed in Chapter 0, while chapter 7 discusses the conclusions of the project. Some suggestions for further studies can be found in chapter 0, as well.

2 Fundamentals

The project is based on two main standards: Extensible Markup Language (XML) and Session Initiation Protocol (SIP). Another standard, the Intelligent Networks (IN), is the current implementation of the intelligent services for traditional telephone networks, e.g. the 0800 free phone call services.

Considering the goals of this project, it is important to understand what these standards define, and how they work in the telephone networks. In the following sub-chapters these three standards are going to be explained briefly.

The services and the architecture implemented in this project are very similar to the services and architecture of the IN. However, Internet Telephony allows new services to be provided comparing to the traditional telephone networks allow.

2.1 Extensible Markup Language (XML)

The Extensible Markup Language (XML) has been developed by World Wide Web Consortium (W3C) [34]. The main purpose of the XML is to provide a standard markup language for the Internet world (World Wide Web). It was released as a recommendation of the W3C in February 1998. Since then, many supporting standards have been released by W3C and some other bodies. For example, the Namespaces recommendation was released in January 1999, and the XML Schema recommendation was released just a few days ago (2 May 2001). So, XML is getting more stable every day.

In the following subchapters first an XML overview will be given, and then the XML specifications will be discussed in detail. After that a brief discussion of how XML works is given in 2.1.3. SGML as the super set of the XML, and HTML as the subset of XML are explained in the coming subchapters. At the end, a comparison of HTML and XML takes place to clarify their differences.

2.1.1 XML Overview

The Extensible Markup Language is a subset of Standard Generalized Markup Language (SGML, see chapter 2.1.4). XML is the universal format for structured documents and data on the Web [3].

The design goals of the XML are:

a. XML shall be straightforwardly usable over the Internet.

b. XML shall support a wide variety of applications.

c. XML shall be compatible with SGML.

d. It shall be easy to write programs, which process XML documents.

e. The number of optional features in XML is to be kept to the absolute minimum, ideally zero.

f. XML documents should be human-legible and reasonably clear. The XML design should be prepared quickly.

g. The design of XML shall be formal and concise.

h. XML documents shall be easy to create.

i. Terseness in XML markup is of minimal importance.

As can be seen, the first and the most important goal is that XML should be usable over the Internet. Combining this with its simple use and wide range of available applications XML is a candidate to be the standard document language in the Internet.

Human readability makes maintenance of XML documents easy. An XML expert may fix errors in an XML document simply by reading it. XML also provides compatibility among the programs using a document, since it is in the text format. For example an EMACS text editor cannot work with an MS Word document, however any text editor can work with an XML document.

Another important aspect of XML comes from its name: extensibility. Any XML document can be in the future extended or converted to another document. For example, a CPL (see chapter 3) document can be converted to another type of document some time in the future, if it is necessary, or there can be some additions to the current CPL. Additions could be some new tags, or new properties, or new text fields.

2.1.2 XML Specifications

2.1.2.1 Document Type Definition (DTD)

A Document Type Definition (DTD) defines the structure of XML document. All the tags, elements, attributes, and properties are defined in the DTD. An XML document can be either with or without a DTD. However, a document without a DTD does not have a structure definition, so it cannot be used as a structured document. As an example, if there were not the HTML definition, there would be no possibility to browse the Internet pages, since the meaning of each tag could be different for each browser.

2.1.2.2 Well-Formed XML Documents

For a document to be an XML document, it has to be well-formed. Well-formed document conforms to the following criteria (from XML specifications [3]):

a. It must start with an XML declaration, which includes the XML version:

   <?xml version="1.0"?>

b. If it has got a DTD, then it should be referred in the document:

   <?xml version="1.0"?>

   <!DOCTYPE cpl PUBLIC "-//IETF//DTD RFCxxxx CPL 1.0//EN" "cpl.dtd">

c. If it does not have a DTD, it should start with a Standalone Document Declaration (SDD)

   <?xml version="1.0" standalone="yes"?>

d. All tags must be balanced, i.e., each tag should have a start and an end. The only exception is the empty tags, where the closing slash is included in the start tag:

   <html></html>

e. All attribute values must be in quotes (the single-quote character [the apostrophe] may be used if the value contains a double-quote character, and *vice versa,* or ' and " can be used).

f. There must not be any isolated markup-start characters (< or & or >) in the text data (*i.e.* they must be given as <, & and &gt).

g. Elements must be properly nested in each other. No overlapping is allowed.

```
<html>
    <body>Something</body>
</html>
```

2.1.2.3 Valid XML documents

A valid XML document should have a DTD, and it should obey the rules defined in the DTD. For instance, only the tags defined in the corresponding DTD can be used in the XML document, otherwise the document becomes invalid.

As an example for a well-formed and valid XML document, a simple call forwarding example in chapter 3.2.1 can be seen. The DTD for that example is the CPL DTD in appendix B.

2.1.3 How XML works

As explained above, for an XML script to be a valid script it should conform to the corresponding DTD. To be able to use an XML script in an application, an XML parser is necessary. There can be many XML parsers found in the market nowadays, and most of them are free of charge.

Fig. 2-1 explains how an XML script is used in an application. First of all, an XML script and corresponding DTD should be available. The XML parser parses the script according to the DTD, and checks whether the script is valid. If it is valid, then the XML parser offers an Application Programming Interface (API) to the application in order to allow easy read/write operations on the XML document. Using this API, the application accesses the tags and the text in the document and decides what to do with them.

Depending on the specifications in the DTD the meaning of the script may change. In other words, each DTD defines a new language.

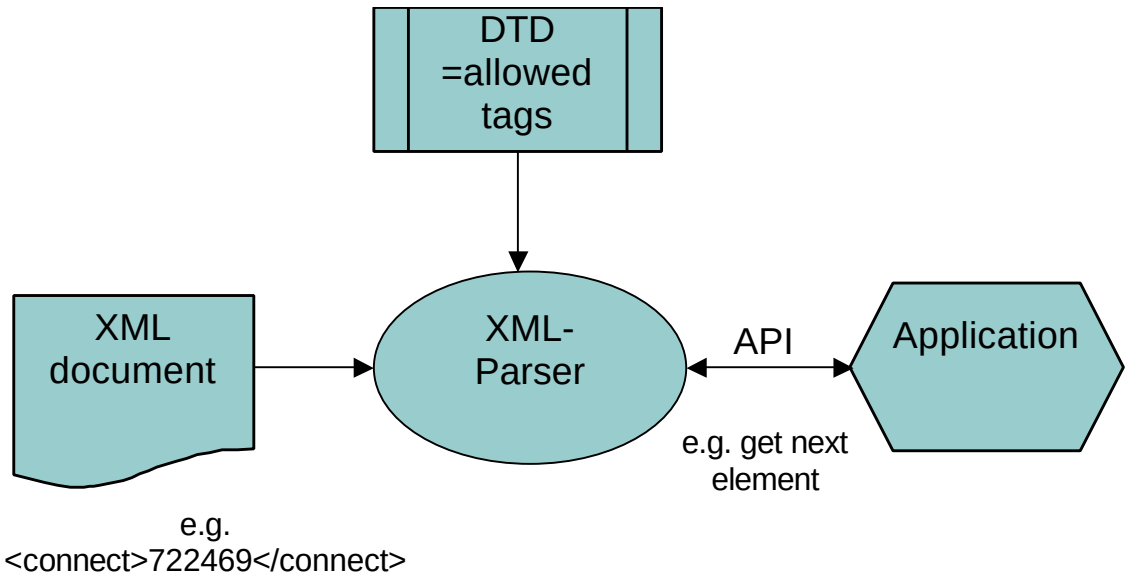

Fig. 2-1 Simple XML usage.

2.1.4 SGML

SGML is the abbreviation of Standard Generalized Markup Language (ISO8879) [22]. It defines the international standards for descriptions of the structure and content of different types of electronic documents.

SGML specifies a standard method for describing the structure of a document. At the same time it prescribes a standard format for embedding descriptive markup within a document.

2.1.5 HTML

HTML is the abbreviation of Hyper Text Markup Language and it is a subset of SGML [24]. Any valid HTML document is at the same time an SGML document. It is very widely used in the Internet world today. However, because it does not cover all the needs of developers it has already been extended several times by different browser vendors. Since the original HTML definition is not powerful enough, some incompatibilities appeared among different manufacturers. For example, Internet Explorer and Netscape do not support the same tags and properties.

Any starting tag should be closed by a closing tag in HTML documents. But, HTML browsers generally do not validate any HTML document. Instead, they ignore any unknown tag in the document, and any unclosed tag is closed by the browsers by predicting the end of the starting tag. For example, an HTML document has to start with “<html>” and end with “</html>” tag. However, if there exists no “<html>” tag at the beginning of an HTML document, browsers add it automatically. Similarly, if an HTML document does not have the ending tag “</html>”, it is automatically added by the browser at the end of the document.

2.1.6 XML vs. HTML

Although HTML is like an application of XML, there are some major differences. For instance, HTML documents do not have to be well formed. So, XML browsers (parser) can, also, read and validate an HTML document, if they have the HTML DTD.

The most important difference between XML and HTML is that with XML the developer can define his/her own tags according to his/her needs, whereas with HTML the developer has to use already defined tags of HTML. HTML tags are defined in the HTML DTD.

Fig. 2-2 presents the difference between the HTML and the XML. While HTML is used for graphical representation of data in the Internet, XML describes the data content. With the help of style sheets XML can also be used to represent data in the Internet, where style sheets define how tags should appear in a browser.

Shortly repeating, an HTML document can only be used for describing graphical formatting, while an XML document can also be used as data repository, e.g. an application for the library systems.

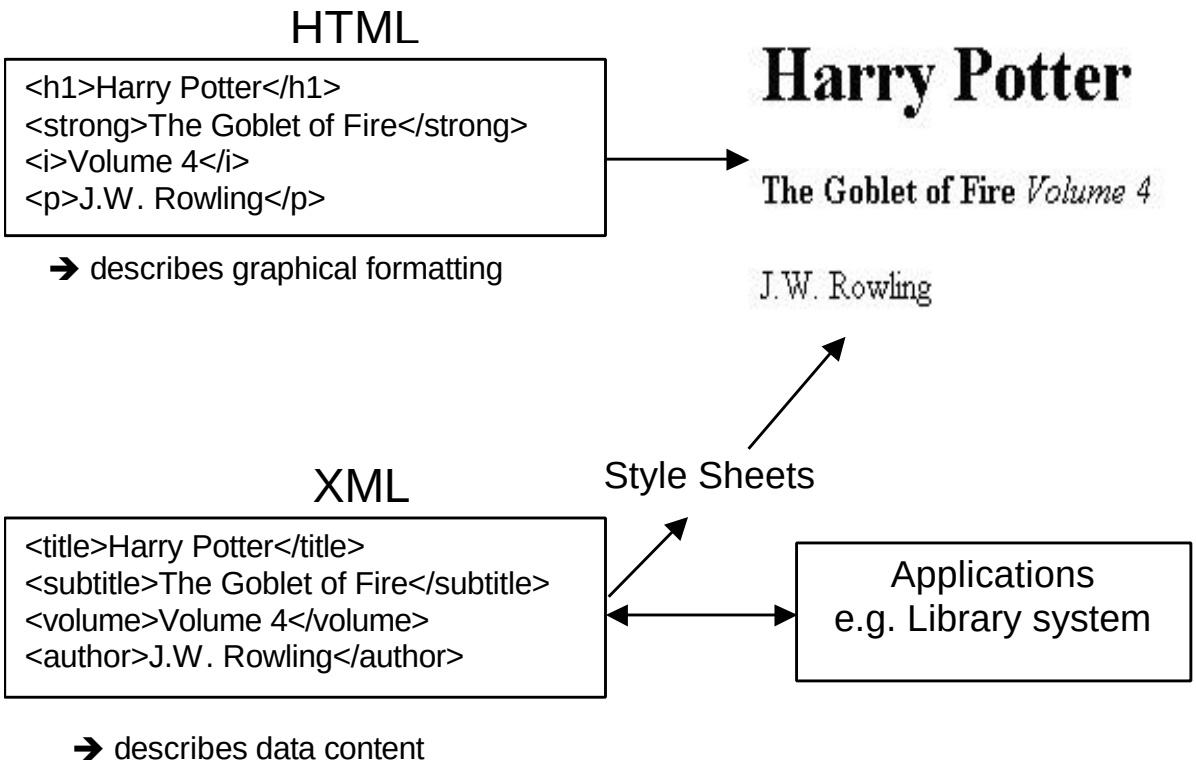

Fig. 2-2 The difference between XML and HTML.

2.2 Session Initiation Protocol (SIP)

In this chapter, the Session Initiation Protocol (SIP) and its necessity for this project is going to be explained briefly.

SIP is an application-layer control (signaling) protocol for creating, modifying and terminating sessions with one or more participants. These sessions include Internet multimedia conferences, Internet telephone calls, and multimedia distribution. Members in a session can communicate via multicast, via a mesh of unicast relations, or a combination of these [4]. Applications of SIP could be Internet conferencing, telephony, presence, event notification, and instant messaging.

SIP is the signaling protocol used in this project. H.323 protocol could be an alternative for SIP. H.323 was developed by the International Telecommunications Union (ITU) and originally designed for multimedia conferencing on a LAN, but has been extended to cover the Internet Telephony [21].

SIP offers many advantages as a platform for programming telephony services [27]:

- Its clean request-response model is amenable to simple programming.
- Its textual formatting and simple header structure make it easy to use text processing languages, such as Perl, and textual interfaces, such as CGI, for developing services.
- Its ability to work in a fully distributed fashion, avoiding routing loops and maintaining consistent behavior across servers, helps avoid future interactions when programming services.

Because of the listed advantages, SIP has been used as the signaling protocol for this project. As a result, a SIP server has been used to implement and test the services provided by the Call Processing Language (CPL).

2.2.1 SIP Characteristics

SIP has got the following characteristics:

a. It supports various addressing types expressed as URLs, such as SIP, H.323, or telephone (E. 164) addresses.

b. It is text-based, so it allows easy implementation and debugging.

c. It is independent of the packet layer protocols and requires only an unreliable datagram service, since it provides its own reliability mechanism. It could be run over TCP, UDP, IPX, or even other network layer protocols.

d. It can be used for signaling the Internet real-time services.

e. SIP provides the necessary protocol properties to support, e.g., the following services:

- Call forwarding for different conditions.
- Number delivery for both participants in different naming schemes.
- Personal mobility, i.e. the same address (number) can be used for many locations and it can be changed any time.
- Terminal-type negotiation, so that the caller can select how to reach the destination, with a software phone, Internet Telephony, or a mobile phone, and so on…
- Authentication for both parties.
- Invitations for multicast conferences.
- Terminal capability negotiation.
- Blind and supervised call transfer.
- It can be extended with some other services. To get more information how SIP can be extended, the reader may refer to [29].

2.2.2 SIP Basics

SIP has got two main components: A SIP User Agent and a SIP Network Server. The SIP User Agent is an end system to be used by the SIP users. It can function as a client, or a server depending on the job it performs: When a user initiates a call request it functions as a client and whenever the user receives a call it functions as a server.

SIP Network Server is necessary to handle the call requests of the user agents. There can be three kinds of SIP Network Servers: SIP register server, SIP proxy server, and SIP redirect server. SIP register server updates the location database when a register request comes from a SIP user agent. SIP proxy server forwards requests to the next server after deciding which server should be the next. The next server can be any kind of SIP server, and the proxy server does not need to know anything about it. SIP redirect server does not forward requests to the next server, but it sends the redirect response back to the client, and the client makes the contact with the next server itself.

2.2.3 SIP Addresses

SIP addresses are in the form of “sip:user@host“. The user part can be a user name or a telephone number, while the host part is either a domain name or a numeric network address. A user’s SIP address can be obtained out-of-band, learned via existing media agents, included in some mailers’ message headers, or can be recorded during previous invitation interactions. In many cases it can be guessed from the user’s e-mail address [4].

2.2.4 SIP Messages

All the communication between a SIP user agent and a SIP network server is done through the SIP messages. A SIP message can be either a request from a client or a response from a server.

There can be six possible methods used in a SIP request message: *INVITE, ACK, BYE, CANCEL, OPTION and REGISTER*:

- *INVITE*: This method is used whenever a user or a service is invited to participate in a session.
- *ACK*: This method is used to confirm that the client has received the *INVITE* message. It can only be used following an *INVITE* message.
- *OPTIONS*: This method is used to query a server’s capabilities.
- *BYE*: This method is used by a SIP user agent client to indicate to the SIP server that it wishes to release the call. It can be issued by either a caller or a callee.
- *CANCEL*: This method is used to cancel a pending request.
- *REGISTER*: Clients use this method to register their addresses with a SIP server.

Each of the above request messages can be answered by a response message. The response messages are numbered from 100 to 699 and each message is associated with a text field. Following are the meaning of the response messages (x means any number):

- 1xx: Informational message
- 2xx: Successful
- 3xx: Redirection
- 4xx: Request failure
- 5xx: Server Failure
- 6xx: Global Failures

For example, a positive response message is “200 OK”, which means that the request was performed successfully. A more detailed list of the SIP response messages can be found in Appendix D.

2.2.5 SIP Operation

There can be three kinds of SIP operation depending on the type of the SIP Server. In Fig. 2-3 operation with a SIP Register Server, in Fig. 2-4 operation with a SIP Server and in Fig. 2-5 operation with a SIP Redirection server is illustrated.

With a register server:

1. Caller sends a REGISTER message.
2. SIP Register Server adds location of the caller into the location database.
3. SIP Register Server sends an OK message back to the caller.
4. Callee sends a REGISTER message.
5. SIP Register Server adds location of the callee into the location database.
6. SIP Register Server sends an OK message back to the callee. Now, Both caller and the callee are registered to the SIP Register Server.

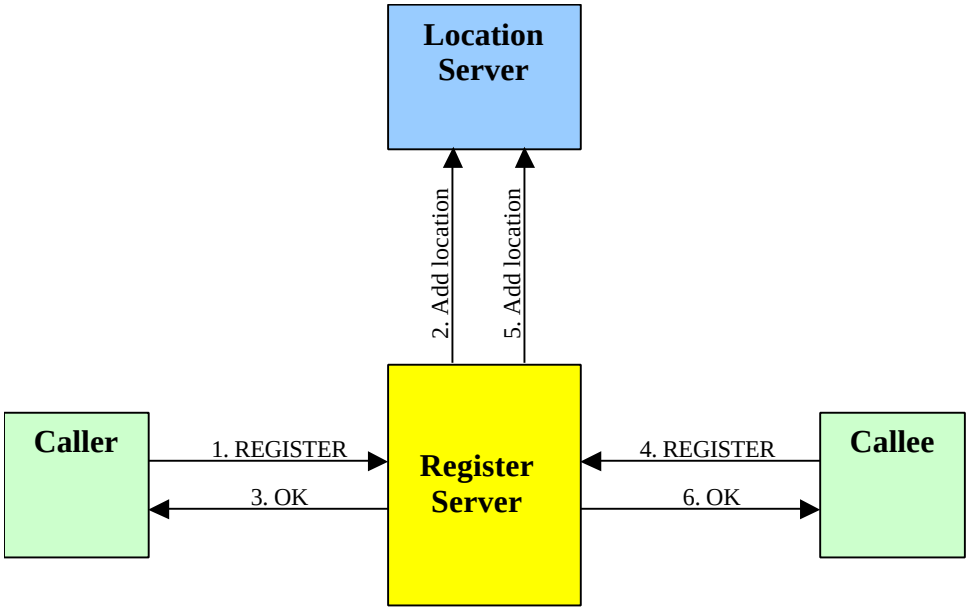

Fig. 2-3 SIP Operation with a Register Server.

With a proxy server:

1. Caller sends an INVITE.
2. Proxy server contacts to the location server to get the address resolved.
3. Location server resolves the address and sends it to the proxy server.
4. Proxy server sends an INVITE to the callee.
5. Callee sends an OK back to the proxy server.
6. Proxy server sends OK to the caller.
7. Caller ACKnowledges the proxy server that it has received the OK message.
8. Proxy server sends an ACKnowledge message to the callee that the caller has received the OK message. Then, session is established.

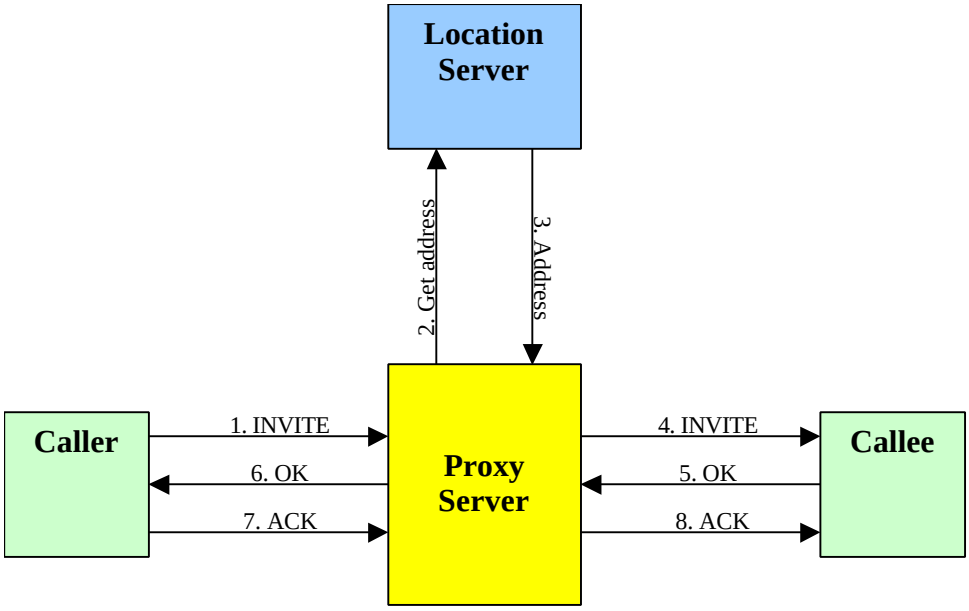

Fig. 2-4 SIP operation through a proxy server.

With a redirect server:

1. Caller sends an INVITE.
2. Redirect server contacts to the location server to get the address resolved.
3. Location server resolves the address and sends it to the redirection server.
4. Redirect server sends the callee's address back to the caller.
5. Caller ACKnowledges the redirect server that it has received the address.
6. Caller sends INVITE to the callee.
7. Callee sends an OK to the caller.
8. Caller ACKnowledges the callee that he has received the OK message. Then session is established.

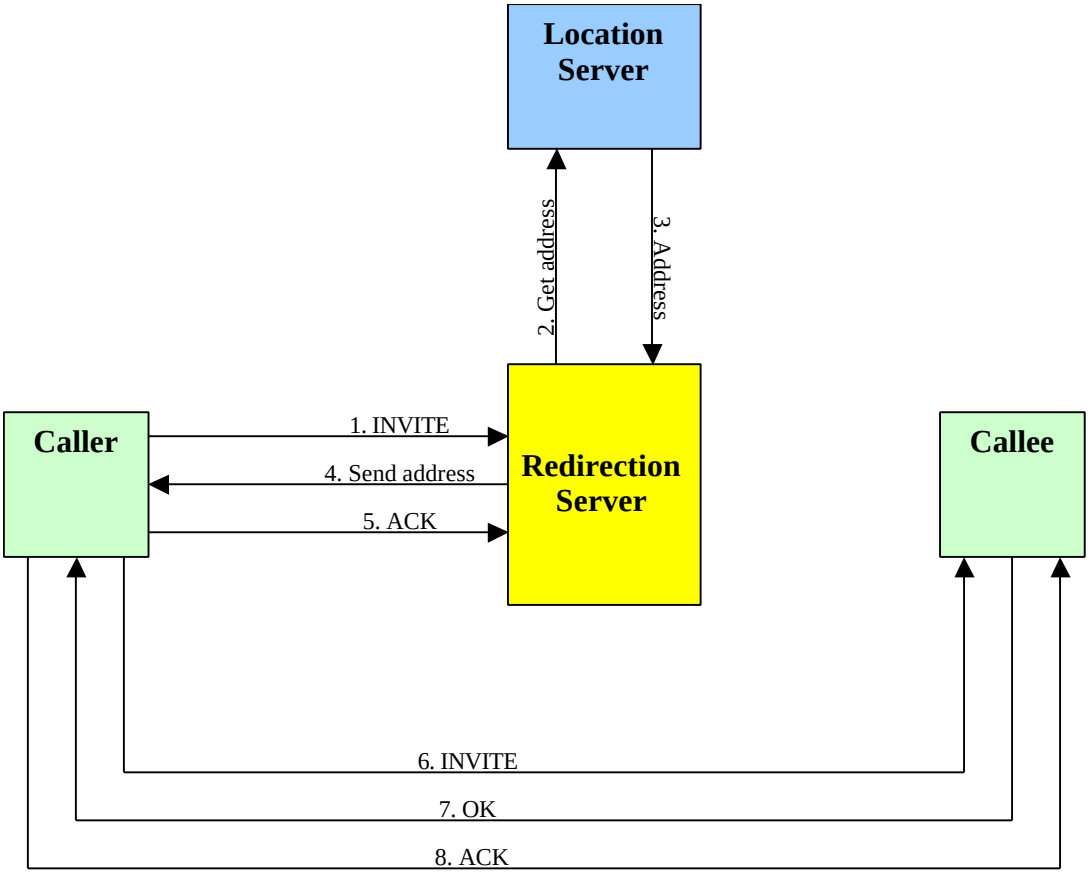

Fig. 2-5 SIP operation with a redirect server.

2.3 Intelligent Networks (IN)

An Intelligent Network (IN) is a service-independent telecommunications network where intelligence is taken out of the switch and placed in computer nodes that are distributed throughout the network [12]. In this way, services can be developed and controlled easily and efficiently. Additionally, new services and capabilities can rapidly be introduced into the network.

Intelligent networks are used to provide additional services to the telephone networks. A call can be setup without IN, however, as an example, billing for 0800 numbers cannot be handled by normal telephone networks. There can be many other services provided by IN. Some of these services are given in chapter 2.3.3.

The IN architecture given in the following subchapter is very similar to the architecture designed in this project for the CPL services. The services provided by IN are also very similar to those provided by the CPL, but the CPL provides new services, which current IN does not offer. For example, e-mail integration is a new service, which does not exist in the IN services.

2.3.1 IN Architecture

Fig. 2-6 shows Advanced Intelligent Network (AIN) Release 1 architecture. In the old plain telephone networks the Service Control Point (SCP) given in Fig. 2-6 does not exist, and

the Service Switching Point (SSP) is replaced by a switching system. However, the old plain telephone network could offer only the call connection between two parties, no other services.

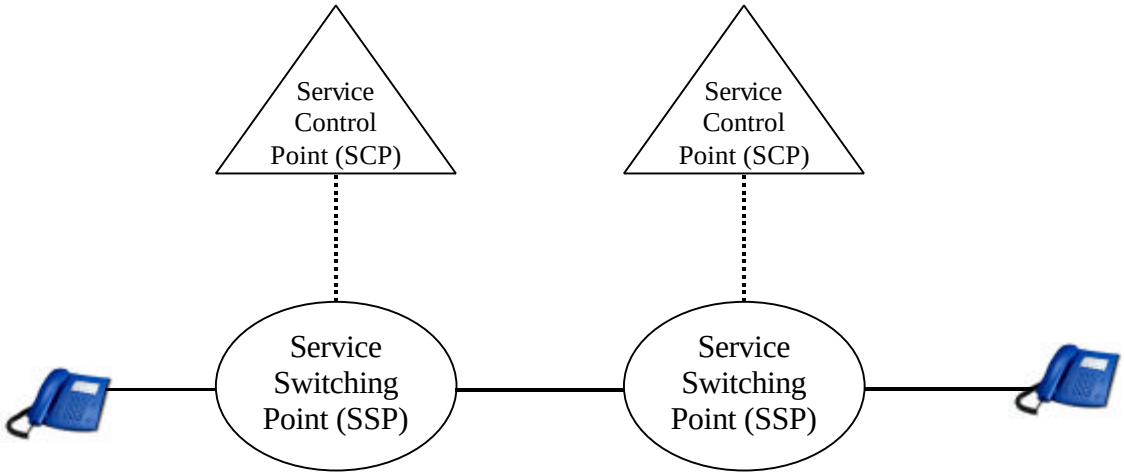

Fig. 2-6 AIN Release 1 Architecture

The service switching point (SSP) in Fig. 2-6 is an AIN capable switching system, which connects the end users to the telephone network, and also provides the access to the set of AIN capabilities. The SSP detects any requests for AIN-based services and establishes the communication with the service control point (SCP) that includes the AIN service logic.

The service control point (SCP) controls the service switching point (SSP) according to the services offered by the provider. In other words, for an incoming or an outgoing call the Service Switching Point communicates with the Service Control Point, and the Service Control Point decides which action to be taken. For example, if the number being dialed is an 0800 (or Free-phone) number, then the Service Control Point realizes that the billing should be done for the called party instead of the calling party.

2.3.2 Benefits of Intelligent Networks

IN offers quite important benefits over the old plain telephone networks. In today's world no operator would survive without providing the services enabled by IN. Its main benefit is the ability to improve existing services and develop new sources of revenue.

With very low rates of phone calls, it is also very important to decrease the costs and offer new services to the customers. To meet these objectives, providers should accomplish following points [12]:

- Introduce new services rapidly
- Provide service customization
- Establish vendor independence
- Create open interfaces

IN is used in the classical telephone networks to achieve the objectives given above. It has been very successful, since its integration into the telephone networks. Similarly, the services offered by the CPL can be used in the Internet Telephony to achieve these objectives.

2.3.3 IN Services

Intelligent networks offer additional services for the current telephone network. These services are very similar to the services planned to be provided by this project in the IP telephony world. Below are some examples of the services offered by an IN network:

- *Basic routing*: With this functionality calls are routed to a single destination.
- *Single number service*: Single number service allows the calls to be treated based on the originating geographical area and the calling party identification.
- *Routing by day of week*: The calls are routed based on the day of the week. For example, a customer may want to be connected to his home location from his office location on Saturdays, and Sundays.
- *Routing by time of day*: Similar like previous entry, this option allows the calls to be routed by time of the day.
- *Selective routing*: When a call to a selective routing customer is forwarded, the SCP determines where to route the forwarded call based on the caller's number.
- *Alternate destination on busy*: With this service, customer is allowed to specify another destination for the call to be forwarded, if the original line is busy.
- *Personal access*: Personal access is a type of follow me service. A virtual telephone number is assigned to the subscriber and whenever this number is dialed the software decides the route to be followed.
- *Enhanced 800 service (Freephone)*: A call to an 800-service number can be routed to different destinations depending on the caller's geographical location, time of the day, day of the week, and caller responses to the prompts. This numbers are generally free numbers, which means that the called party pays for the call, so a different billing mechanism works with this service.
- *Automatic route selection/least cost routing*: This service allows the subscriber to design a priority route for every telephone number dialed. Call is either directed or blocked according to the privileges that the user is restricted.
- *Call counter*: This service lets the subscriber count calls made to a specific phone number that the subscriber owns. This is generally used by TV channels for televoting services.
- *Inmate service*: This service is used to route prisoners' calls. It tracks the call information and offers call control features such as prompts for personal identification numbers, blocking certain called numbers and time of day restrictions.

There can be many other services provided by IN architecture depending on the user needs and the provider specifications. However, nearly all telecommunication companies all over the world provide the services explained above.

3 Call Processing Language (CPL)

The Call Processing Language (CPL) is a language that can be used to describe and control Internet Telephony services. The CPL is based on XML and developed by the Internet Telephony Workgroup of the International Engineering Task Force (IETF). Currently, it has been submitted for Recommendation for Comments (RFC), but it has not yet been approved to be an RFC. The language is defined in the draft "draft-ietf-iptel-cpl-**.txt", where "*" is used to replace the version number. The latest version is "draft-ietf-iptel-cpl-04.txt" currently. Below is a brief history of the CPL:

- 26.02.1999: draft-ietf-iptel-cpl-00.txt (The first CPL draft)
- 10.03.2000: draft-ietf-iptel-cpl-01.txt
- 14.07.2000: draft-ietf-iptel-cpl-02.txt
- 25.10.2000: draft-ietf-iptel-cpl-03.txt
- 14.11.2000: draft-ietf-iptel-cpl-04.txt
- 17.11.2000: submission for RFC

As can be seen from the history of the CPL, nearly two years passed between the first draft and the submission for RFC. In this time significant changes occured compared to the first draft. However, after submission for RFC, very significant changes are not expected on a draft. So, it can be said that the version 4 is going to be more or less standardized (approval of RFC means a standardized draft).

CPL is designed to be implemented on either network servers or user agent servers. It is meant to be simple, extensible, easily edited by graphical clients, and independent of operating system or signaling protocol. It is suitable for running on a server where users may not be allowed to execute arbitrary programs, as it has no variables, loops, or ability to run external programs. [6]

Since CPL is based on XML, a CPL document starts with the XML definition line. Then, comes the "<cpl>" tag, which is used to identify a CPL document. And, naturally, it ends with the closing "</cpl>" tag. The example below illustrates this:

```
<?xml version="1.0" ?>
<cpl>
        .
        .
        .
</cpl>
```

In the following chapters generally the term "CPL script" is used instead of the term "CPL document". This is because a CPL script is processed by a server and some actions are carried as a result of the specifications in the document. Yet, these two terms can be used identically in this thesis.

The CPL offers IN like services for the IP telephony. Even some new services can be implemented using the CPL. For example, integrating e-mail notification into the system is an additional service to the ones already offered by IN.

In this chapter, the specifications of the CPL are discussed in detail; furthermore two examples are introduced to help the reader understand CPL scripts better. At the end of the chapter, extensibility of the language is discussed, and a possible extension is introduced for trusted users, such as service providers.

3.1 Language Specifications

The CPL has been designed to allow end users of the Internet Telephony describe their own service scripts and customize these scripts to their own needs. However, at the same time, it has got a limited power that end users cannot harm the system. For example, a CPL script cannot call any external program, or it cannot have any loops internally.

The CPL is based on XML that allows simple creation and edition of the scripts by graphical tools. Since it is text based, the editor can also understand the script by simply looking at the script.

In the following subchapters, the tags defined in the CPL are explained. The tags are classified in 6 classes: top level actions, switches, location modifiers, signaling actions, non-signaling actions and subactions.

3.1.1 Top Level Actions

Top Level actions are actions that are triggered by signaling events that arrive at the server. Two top-level actions are defined: incoming and outgoing. **Incoming** action is triggered when a call arrives whose destination is the owner of the script. **Outgoing** action is triggered when a call arrives whose originator is the owner of the script.

Any tag can be included in the top-level actions, except for the subactions (see chapter 3.1.6) and the other top-level action. And, at most one incoming and one outgoing tag can be used in the script.

3.1.2 Switches

Switches are used to make choices in a CPL script. There can be four kinds of switches in a CPL script: address switches, string switches, time switches and priority-switches. All switches are list-like conditions to be matched to a variable. Otherwise the tag is used to make decisions in case of no matching condition.

Address switches are used when the user wants to make decisions based on the addresses present in the original call request. These addresses could be the original destination address, originating address, or any other address in the call request.

For example, if a user does not want to get any calls from a specific person, then he/she can define an address-switch that matches the specified person’s address, and reject the call request. The example below illustrates this:

```
<address-switch field=”origin”>
  <address is=”somebody@test.com”>
    <reject />
  </address>
</address-switch>
```

String switches allow a CPL script to make decisions based on free-form strings present in a call request. String switches are based on the signaling protocol being used by the server. There are five fields to be matched, which are defined by the CPL, but the developer can add new fields. Already defined fields are:

Subject: The subject of the call.
Organization: The organization of the originator of the call.
User-agent: The name of the program device with which the call request was made.
Language: The languages in which the originator of the call wishes to receive responses

Display: Free-form text associated with the call, intended to be displayed to the recipient, with no other semantics defined by the signaling protocol.

Time switches allow a CPL script to make decisions based on the time and/or date the script is being executed. For example, a user may not want to be disturbed after midnight. Then he/she can specify this in his/her script that if the time of the incoming call is after midnight, then the call is forwarded to the voice mailbox. There can be a very wide variety of time conditions, and matching algorithms. These are all defined in [6].

Priority switches allow a CPL script to make decisions based on the priority specified for the original call. Priority switches mainly depend on the signaling protocol.

3.1.3 Location Modifiers

While processing a CPL script a list of locations is created to be signaled according to the signaling action specified in the script. More than one location may be signaled for an incoming call at the same time.

CPL defines three tags to modify the location list for a call: location, remove-location and lookup. **Location** tag is used to add an address in the location list, while **remove-location** tag is used for removing a location from the list. **Lookup** tag is used to check the validity of a location.

In the example below, two locations are first inserted into the location list: “sip:home@sertac.test.com” and “tel:+491707665432”. Then the signaling action comes, proxy: The server connects the call to both of the locations, and both locations ring at the same time. After one of the locations holds the phone, the call request to the other location is canceled.

```
<location url=”sip:home@sertac.test.com”>
  <location url=”tel:+491707665432”>
    <proxy />
  </location>
</location>
```

3.1.4 Signaling Actions

Signaling actions are sent to the SIP server for an appropriate response to the corresponding user. This user could be the originating user or other users depending on the signaling action. Three signaling actions can be sent to a SIP server by a CPL script: proxy, redirect and reject.

Proxy causes the triggering call to be forwarded on to the currently specified set of locations (see Fig. 2-4). A proxy signal can have some output nodes:

- busy: The called party is already busy
- no-answer: The called party does not hold the phone in the specified time (timeout).
- redirection: The called party has redirected the call to another location.
- failure: Failed to connect to the destination.
- default: Any other response from the destination.

After making a proxy signal the server attempts to make the connection to the appropriate location. If the attempt is successful, then the CPL execution is terminated and the server proceeds to its default behavior. However, if the call attempt is not successful, e.g. busy, the script is further processed with the corresponding output node.

Proxy signaling action is clarified in the example below. There, the server tries to connect the call to the specified set of locations. If one of the locations is busy, then the server proceeds with the busy output in the script, and this forwards the call to the voice mailbox. Or, if one of the locations does not answer in five seconds, which is specified in the proxy tag, the server proceeds with the no-answer output, and sends an e-mail to the address specified in the mail tag and terminates the call.

```
<proxy timeout=”5”>
  <busy>
    <sub ref=”voicemail” />
  </busy>
  <no-answer>
    <mail url=”sertac@mail.test.com” />
  </no-answer>
</proxy>
```

Redirect causes the server to direct the calling party to attempt to place its call to the currently specified set of locations (see Fig. 2-5). The difference between the redirect and the proxy is that redirect terminates immediately after the signaling, but proxy makes the server wait for the result of the call attempt(s).

Reject causes the server to reject the call attempt. Like redirect, execution of the script is terminated immediately after the rejection.

3.1.5 Non-signaling actions

Non-signaling actions are used to make the server perform some operations without resulting with a signaling action. These operations are independent of the signaling protocol, and do not affect it. Currently, two non-signaling actions are defined for the CPL: mail and log.

Mail operation causes the server to notify a user of the status of the CPL script through electronic mail. It takes the e-mail address as an argument. In the mail message, server and developer specific information exists. The time of the call, the originating address and the original destination address (may be different from the destination address in case of a redirection) could be the information given in the mail body.

Log operation causes the server to keep a log of the current call in the system. The user should have some means to access his/her log files through FTP or some other file access protocols. Similar to the mail action, the information given in a log message is specified by the server and the developer. Similar fields, like time of the call, the originating address and the original destination address could be written in a log file.

3.1.6 Subactions

Subactions are defined by the CPL to allow multi usage of one branch in the script. However, a subaction cannot be called from within a previous subaction in the script, which prevents any loops in the script.

In the example below, voicemail location is specified as a subaction, which allows the voicemail location to be refered more than once in the script. The reference to a subaction is done with the sub tag. The “ref” attribute gives the id of the subaction to be referred in the script.

```
<subaction id="voicemail">
  <location url="sip:sertac@voicemail.test.com">
    <redirect />
  </location>
</subaction>
            .
            .
            .
<proxy timeout="20">
  <default>
    <sub ref="voicemail" />
  </default>
</proxy>
            .
            .
            .
<proxy timeout="8">
  <busy>
    <sub ref="voicemail" />
  </busy>
</proxy>
```

3.2 Examples

To be more specific two examples are included in this chapter. The first example is a very simple one, while the second one gives a deeper understanding of the CPL. In the examples below; the CPL DTD reference line is not included for a better readability.

3.2.1 Example 1: Simple call forwarding

In Fig. 3-1 a simple call forwarding example is given. This 8-line script forwards each incoming call to the “sip:sertac@test.com” location. This URL is a SIP address, which has already been explained in chapter 2.2.3. However, this could be a phone number, or an H.323 address, as well.

```
<?xml version="1.0" ?>
<cpl>
  <incoming>
    <location url=”sip:sertac@test.com”>
      <proxy />
    </location>
  </incoming>
</cpl>
```

Fig. 3-1 A simple CPL Script

In this example the script describes only the location for the call to be forwarded. The first line defines the XML document and its version. In the second line the CPL script starts with the “<cpl>” tag. “<incoming>” tag is used for any incoming call, while “<outgoing>” tag is used for any outgoing call. As it can be seen in the example, each starting tag is closed with a closing tag. However, there is an exception for this rule: If the tag is an empty tag which means it does not include any text or any other tags in it, then it does not need a closing tag,

but an empty tag closure. This is illustrated in the example with the “<proxy />” tag, which is an empty tag.

3.2.2 Example 2: A more complicated example

Compared to the previous example, this is a more complicated, but at the same time a more functional script. In this script some important concepts of the CPL is illustrated, such as redirection, address-switch and proxy-response behavior.

```
<?xml version="1.0" ?>
<cpl>
 <subaction id="voicemail">
  <location url="sip:sertac@voicemail.test.com" clear="yes">
   <redirect />
  </location>
 </subaction>
 <incoming>
  <mail url="sertac@mail.test.com">
   <address-switch field="origin">
    <address is="boss@test.com">
     <location url="tel:+491704567348">
      <proxy timeout="8">
       <busy>
        <sub ref="voicemail" />
       </busy>
       <noanswer>
        <location url="sip:home@test.com">
         <proxy />
        </location>
       </noanswer>
      </proxy>
     </location>
    </address>
    <otherwise>
     <location url="sip:sertac@test.com">
      <proxy timeout="20">
       <default>
        <sub ref="voicemail" />
       </default>
      </proxy>
     </location>
    </otherwise>
   </address-switch>
  </mail>
 </incoming>
</cpl>
```

Fig. 3-2 A complicated CPL script

In this script, an e-mail is sent to “sertac@mail.test.com“ for each incoming call. This e-mail gives some information about the incoming call, which is server and implementation

specific. However, the time of the call, calling party address, and the original destination address would most probably be included in the mail body.

Each incoming call is checked for its originating address, and if it is from “boss@test.com”, which represents the boss of the user, then call is forwarded to the mobile number. If the mobile number is busy, the call is forwarded to the voicemail location, and if there is no answer from the mobile location in 8 seconds then the call is forwarded to the “home@test.com” location, which represents the home location of the user.

Voicemail location is represented by a subaction in the script, so that, it can be referred more than once in the script. For example, in Fig. 3-3 two different branches refer to the same voicemail location.

If the call is from somebody other than the boss, the call is forwarded to “sertac@test.com" location, and if it is not successfully connected in 20 seconds, the call is forwarded to the voicemail location. Fig. 3-3 shows graphically how the script is interpreted.

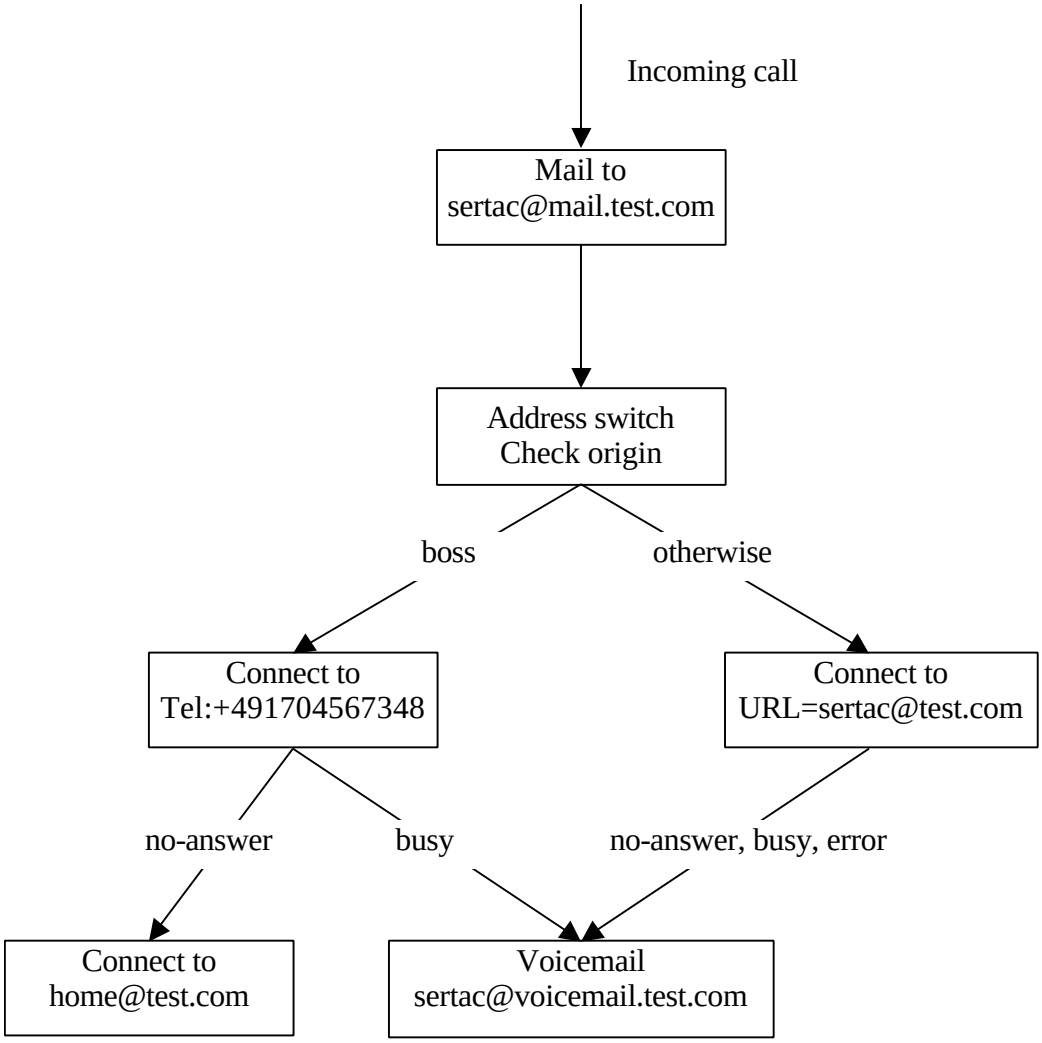

Fig. 3-3 Graphical representation of Fig. 3-2.

3.3 Extensibility

As a general rule for the XML documents, CPL is extendable with any other tag, which is not defined in the CPL DTD. Extensibility of the language can be used for the administration of the CPL Scripts, which could be very useful for the service providers. With some additional tags user administration, billing, and some other operator specific services can be included in the system.

Considering the trust relations in the call services, an end user is un-trusted, while a service operator is a trusted user. CPL has been designed for un-trusted users, but with some extensions trusted users can also benefit from it. To provide services for trusted users a new DTD should be defined. Fig. 3-4 illustrates how CPL can be extended for trusted user services.

Extensibility of the CPL could be a field for further study on this project. With the further study, Fig. 3-4 could be implemented and evaluated, completely. In this thesis (project), only the CPL and services for un-trusted end users have been implemented and evaluated.

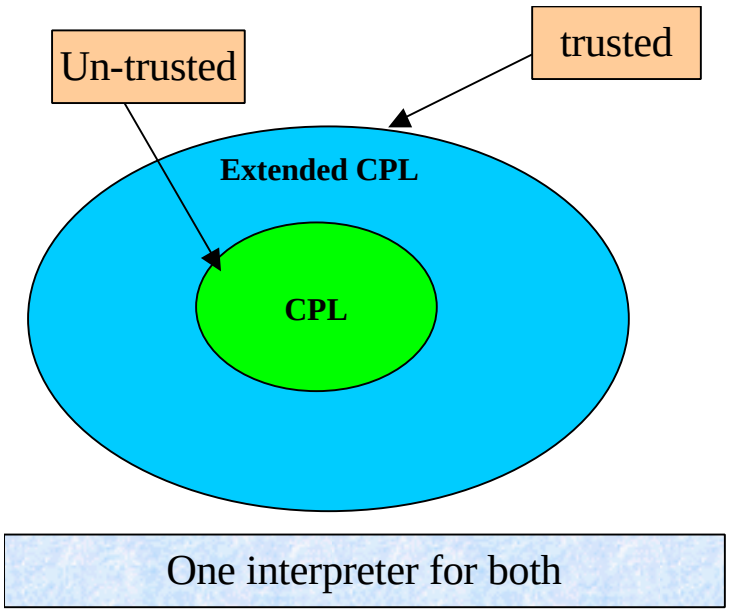

Fig. 3-4 CPL Extension.

4 Architecture

In Fig. 4-1, all the components necessary for the implementation of the CPL can be seen. This picture could be drawn in different ways depending on the architecture of implementation. In chapter 4.7 possible different architectures are going to be discussed. As can be seen clearly in Fig. 4-1 six main components exist in the architecture of the project: The SIP user agent, the SIP Server, the Location Database, the CPL Engine, the CPL Repository, and the CPL User Editor.

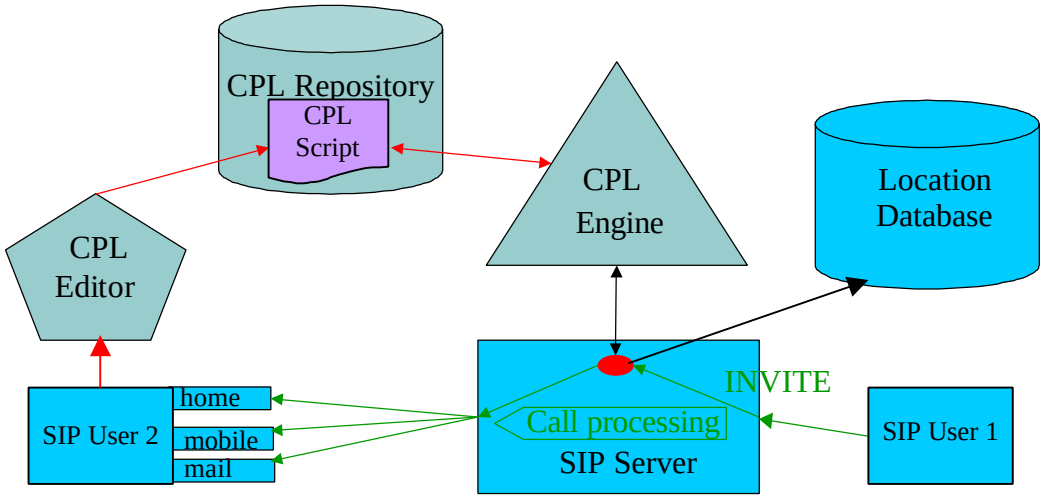

Fig. 4-1 Architecture of the project.

Fig. 4-2 illustrates how a call is established between two SIP users, e.g. SIP User 1 and SIP User 2. All steps are explained below:

1. SIP User 1 registers to the SIP Server.
2. SIP Server adds the location of the SIP User 1 into the Location Database.
3. SIP User 2 registers to the SIP Server.
4. SIP Server adds the location of the SIP User 2 into the Location Database.
5. SIP User 2 loads his/her CPL script into the CPL Repository.
6. SIP User 1 makes a call to request for the SIP User 1 (INVITE).
7. The SIP Server gets the call request of the SIP User 1, and checks the Location Database if the SIP User 2 is already registered, and if the CPL flag of SIP User 2 has already been set. (Positive for both cases)
8. The SIP Server contacts the CPL Engine to ask for the appropriate signaling action for the SIP User 2.
9. The CPL Engine gets the CPL script for the SIP User 2 from the CPL Repository. The CPL Engine parses the CPL script and interprets it. Resulting signaling action is sent back to the SIP Server. (Lets assume that the signaling action is proxy to the SIP User 2)
10. The SIP Server forwards the call request to the SIP User 2 (INVITE).
11. SIP User 2 holds the phone, and the connection is established.

Following subchapters explain each component shown in Fig. 4-1. The SIP Server, The Location Database and the SIP User are explained briefly. However, the CPL Engine and the CPL Repository are explained in detail, since they are the main focus of the thesis.

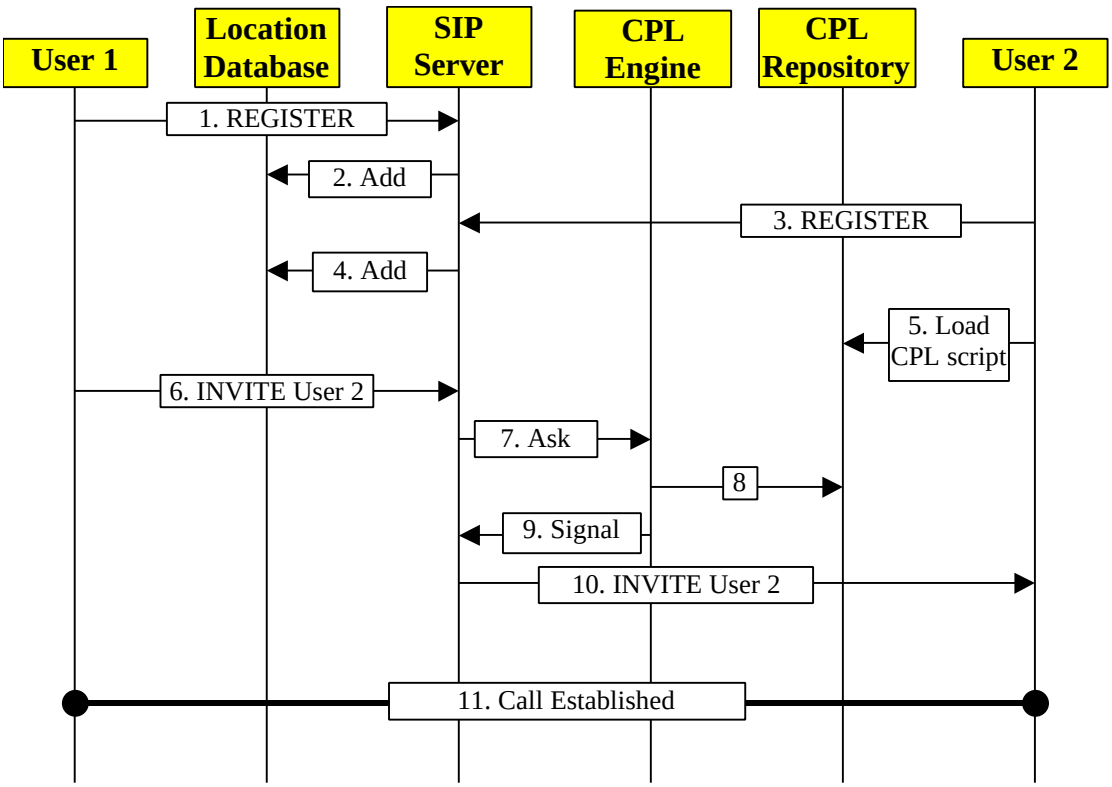

Fig. 4-2 Call establishment between two parties.

4.1 SIP User

The SIP User is basically an end user who wants to make a phone call through the IP network. He/She could have a SIP phone, a software phone, or a mobile phone. Here, a SIP phone is a real phone, not for the telephone networks, but for the IP networks. A software phone is not a real phone, but just a software implemented to get phone calls, and make phone calls. Using the headsets and a software phone, a user can make phone calls through his/her computer. An example of software phone for SIP can be downloaded from [30]. Of course, the SIP user has to be configured to an appropriate SIP Server. The SIP User registers itself to the SIP Server, and the SIP Server registers the user's IP to its location database. The SIP User periodically registers itself to the server to provide the synchronization.

4.2 SIP Server

The SIP Server listens to the network for any register and call requests (INVITE). A register request causes the SIP Server register the location of the SIP User into the Location Database. After getting the call request (INVITE) message, the SIP Server asks the location database if the destination user is registered at the server. If it is not registered then the SIP Server cancels the request. If it is already registered, then the SIP Server checks if the user has got a CPL Script. If it has, then the CPL Engine is queried for the appropriate signaling action. Otherwise, the connection is done with the destination address in the location database.

In Fig. 4-3 a block view of the SIP Server is seen. The details of the SIP Server are not explained here since the SIP Server is out of the scope of this thesis. However, the CPL API at the top of the figure should be emphasized that it was added to the SIP Server implementation to allow the SIP Server communicate with the CPL Engine. There, another important thing is the connection between the CPL API and the Location Database API. With the help of this

connection, the CPL Engine can communicate with the Location Database through the SIP Server.

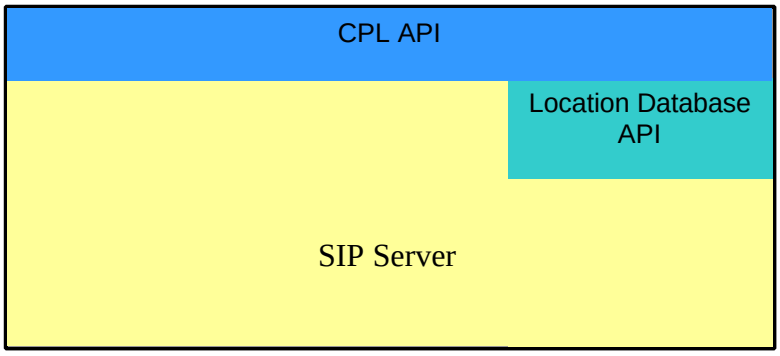

Fig. 4-3 Block Representation of SIP Proxy server

4.3 Location Database

The Location Database keeps records of the users registered to the SIP Proxy. Location addresses and the CPL flags of the users are recorded in the database. There could be more than one location for one user. In this case multi locations are stored and all these locations are signaled for an incoming call.

The CPL flag is a location URL to specify the location of the CPL script. CPL flags are used to decide if the user has got an active CPL script that is already loaded into the CPL Repository. If the CPL flag for a user exists in the Location Database, then the user has got an active CPL script in the CPL Repository. Otherwise the user does not have any active CPL script in the CPL Repository.

4.4 CPL Engine

The CPL Engine is the core part of the project. After getting a call request (INVITE) from a SIP user, if the destination user has got an active CPL script, the SIP Server communicates with the CPL Engine. Then, the CPL Engine proceeds with the request and gets the CPL script from the CPL repository, parses it, interprets it, and finally returns back to the SIP Server with the resulting signaling action.

4.4.1 Design

Fig. 4-4 presents a block design of the CPL Engine. There, it is seen that the CPL Engine communicates with the CPL Repository and with the SIP Server. Both communications are two-way communications, which allows both sides to make requests on each other.

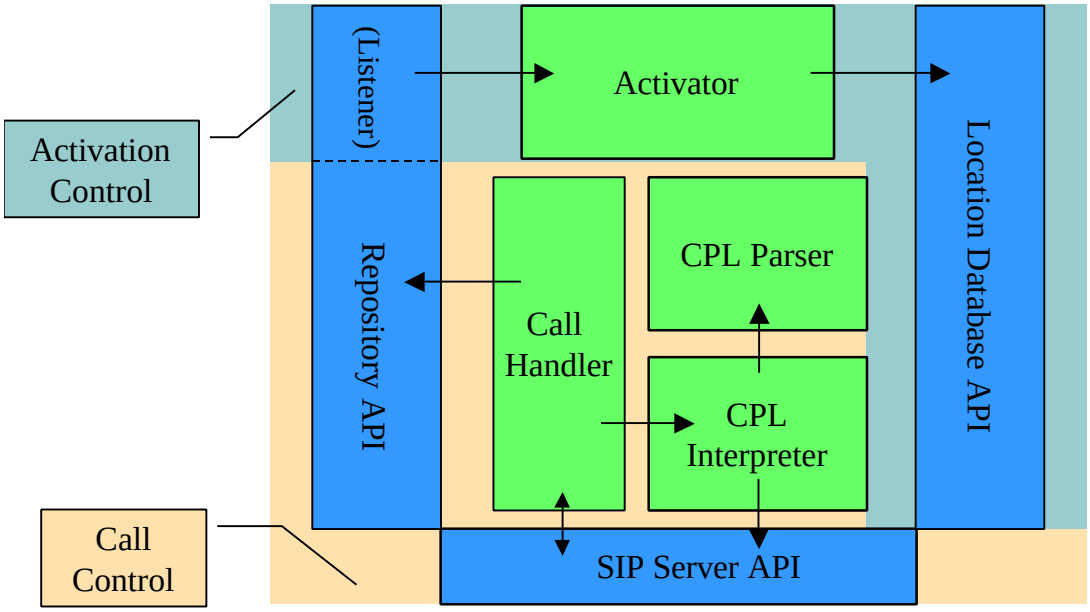

Fig. 4-4 Main blocks in the CPL Engine.

The CPL Engine performs two main tasks: Call Control and Activation Control. Call Control is performed when there is a request from the SIP Server, and Activation Control is performed when there is a request from the CPL Repository (see chapter 4.5).

The CPL Engine consists of 4 main blocks: the Call Handler, the CPL Parser, the CPL Interpreter and the Location Database Activator. As it can be seen in Fig. 4-4, the Call Handler makes all the communication between the CPL Engine and the CPL Repository. Additionally, the Call Handler handles most of the communication between the SIP Server API and the CPL Engine. The SIP Proxy Server makes all the requests to the CPL Engine through the Call Handler, while the CPL Engine can make the requests to the SIP Server API through either the Call Handler or the CPL Interpreter. The CPL interpreter calls the SIP Server API as a result of signaling action found in the CPL script.

For every request coming from the SIP Server API, the Call Handler starts a transaction. The states of the transaction exist as long as the script processing has not reached its end (this might include several different call setup attempts). After that, following actions occur in the CPL Engine:

- The Call Handler gets the CPL script of the corresponding user from the CPL Repository.
- The Call Handler passes the CPL script to the CPL Parser.
- The CPL Parser parses the CPL script.
- The Call Handler passes the parsed CPL script to the CPL Interpreter.
- The CPL Interpreter interprets the CPL script. If any signaling action is found in the script, the CPL Interpreter calls back the SIP Server API with the appropriate signaling action. If no signaling action is found in the scrip, then the SIP Server API is informed to continue its default action, as there was no CPL script for the user.
- In case of any errors, the Call Handler informs the SIP Server API to continue its default action, as there was no CPL script for the user.

For every request coming from the CPL Repository, the Location Database Activator starts, and activates or deactivates the location database for the corresponding user, which is done by putting or removing the CPL Flag of the user in the Location Database.

4.4.2 Flow diagram

Fig. 4-5 illustrates all activities performed by the CPL Engine for a request coming from the SIP server. There can be three types of requests from the SIP Server. StartCall, ContinueCall or DestroyCall.

StartCall request comes if the SIP Server receives a new INVITE message from a SIP User and the callee has got a CPL flag in the Location Database. After getting the StartCall request from the SIP Server, the CPL Engine starts a transaction for this call. Then the CPL Repository is asked for the CPL Script for the corresponding user. If the user does not have an active CPL Script in the repository, the CPL Engine calls back the SIP Server with the "Continue SIP Default" signaling action. Otherwise, the CPL Script is parsed by the parser and passed to the CPL Interpreter to be interpreted for the corresponding signaling action. "Continue SIP Default" action tells the SIP Server to behave as the user did not have any active CPL script.

After signaling the SIP Server with "proxy", the SIP Server tries to connect the user to the given address. If the connection is established successfully, which means a "200 OK" message from the caller has been received, then the SIP Server calls the CPL Engine with "DestroyCall". As a result of this call, the CPL Engine destroys the transaction for the corresponding user.

If connection attempt of the SIP Server is not successful (busy, redirection, error), the SIP Server calls the CPL Engine with "ContinueCall". ContinueCall means continue from the previous state, which was a proxy signal. As a result, interpretation continues from that point on, until a new signaling action is resulted, or the script ends.

Two main blocks have been defined in the flow diagram: Process Call and Prepare Call blocks. Process Call block is executed after having a script parsed by the XML parser, or after getting a Continue Call request from the SIP Server. Process Call block has to result with a signaling action. If no signaling action is achieved in the block, then Continue SIP Default action is taken. Prepare Call block is executed after getting a Continue Call request from the SIP Server. This block prepares the call for further processing. The response of the call attempt is located in the CPL script, and if the response tag is not found in the script then the "<default>" tag is located.

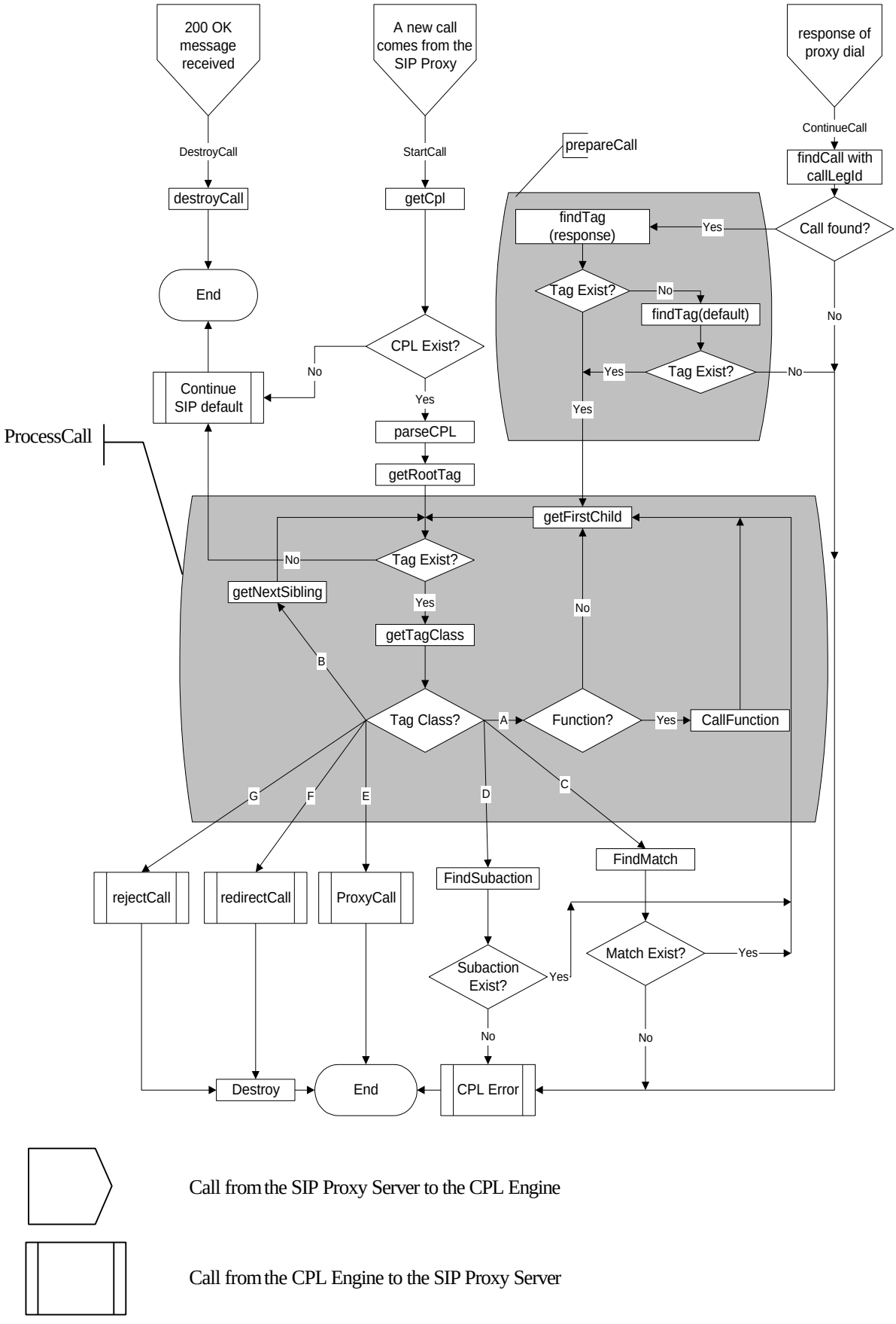

Fig. 4-5Flow diagram of the CPL Engine.

4.4.3 Tag Classes

The CPL Interpreter interprets each tag in the CPL script separately. Each tag is assigned a class according to the Table 4-1. Based on the Table 4-2, some actions are performed for each tag class. For example, if the tag class is A, the CPL script is further processed with the first child of the current node.

Classes E, F and G are the signaling classes, while other classes do not have any signaling action to be performed. If the tag class is E, timeout for the proxy is read from the script and the proxy signal is sent to the SIP Server. Similarly, if the tag class is F, the redirection signal is performed, or if the tag class is G the reject signal is sent to the SIP Server.

Some tags of class A have some additional functions to be performed. These functions are performed before continuing with the actions defined in Table 4-2. Only four tags are defined to have these optional functions: location, remove-location, mail and log.

Tag	Interpretation	Class	Function
cpl	Cpl	A	None
incoming	Incoming	A	None
location	Location	A	AddLocation
busy	Busy	A	None
noanswer	Noanswer	A	None
redirection	Redirection	A	None
failure	Failure	A	None
default	Default	A	None
remove-location	remove-location	A	RemoveLocation
mail	Mail	A	SendMail
Log	Log	A	CreateLog
subaction	Subaction	B	None
address_switch	address_switch	C	None
string_switch	string_switch	C	None
time_switch	time_switch	C	None
sub	Sub	D	None
proxy	Proxy	E	None
redirect	Redirect	F	None
reject	Reject	G	None

Table 4-1 Tags implemented in the prototype.

Class	Action
A	GetFirstChild
B	GetNextSibling
C	FindMatch, GetFirstChild
D	FindSubaction, GetFirstChild
E	GetTimeout, ProxyCall
F	RedirectCall
G	GetStatus, GetReason, RejectCall

Table 4-2 Functions to be performed for each tag class.

4.5 CPL Repository

The CPL Repository holds all scripts of the users. The CPL Repository can be read or written by the CPL Editor, while the CPL Engine can only read it. The CPL Repository is also responsible for informing the CPL Engine about activation or deactivation of a script, so that the CPL Engine could set the CPL flag in the Location Database.

When the CPL Repository gets a query from the CPL Engine or from the CPL User Editor, it must be able to give unique active and inactive scripts for the specified user. Although there can be more than one inactive script, there can be at most one active script for one user in the CPL Repository. Considering the address definitions in the IP world, <user name>@<domain name>, domain names, user names and the scripts should be stored in the CPL Repository to be able to define unique scripts for the users.

Three points play an important role in the architecture of the CPL Repository:

First, it could be a part of the Location Database, and with some additions to the Location Database the implementation would be complete, or a separate location would be assigned to the CPL Repository to have more flexibility. Flexibility in this context brings the following advantages:

a. The CPL Repository and the Location Database do not depend on each other, can be modified, improved or changed without considering any effects on each other.

b. One CPL Repository or one Location Database could be used to store information from more than one CPL Engines, and SIP Servers.

c. Performance of the SIP Server would be better, since there would be less information in the Location Database.

The second discussion in the implementation of the CPL Repository is using the flat files for storage, as it is the case for the Location Database, or using a database.

The third discussion is having an active or a passive CPL Repository. These two points are discussed in the following two subchapters.

4.5.1 Flat File vs. Database

4.5.1.1 Flat File

Using flat files for the CPL Repository would be an inexpensive solution, but it does not provide a scalable implementation, since any changes to the CPL Repository would require too much effort to change the file access implementation. In this project flat files were used at the early phases for testing, but then a database implementation replaced the flat files.

The directory structure for the flat files used in this project can be seen in Fig. 4-6 below. There, the root directory starts with “users”, which means all the user scripts are stored under this directory. Under the “users” directory a directory is created for each domain, such as “test.com”. Then, under the domain directory for each user a directory is created, such as “sertac”. Each user has got two directories, one for active scripts, and one for inactive scripts. There can be normally only one active script for each user. This has to be controlled by the implementation.

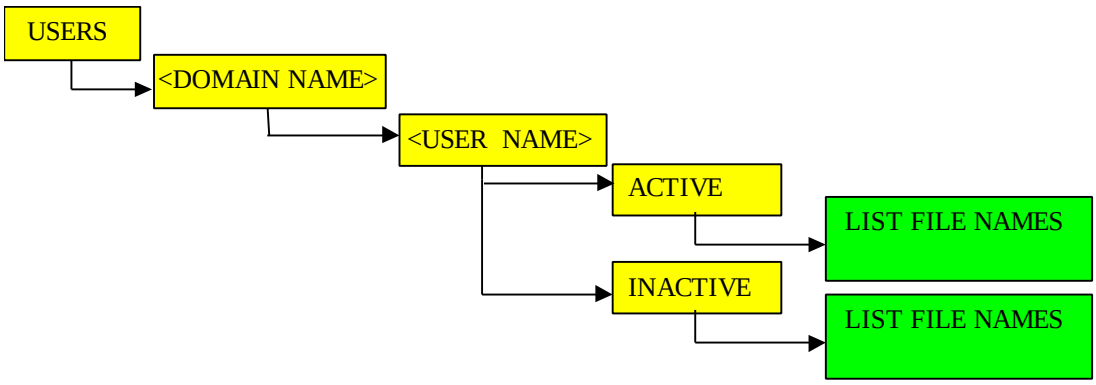

Fig. 4-6 Directory structure for the repository.

4.5.1.2 Database

The database solution could be more expensive compared to the previous solution for the repository architecture, but more scalable, and it would have a better performance for large number of entries. The database solution could be more expensive, because a real database license, such as Oracle or MS SQL Server, costs quite a lot. However, some freely available databases could be used instead of these expensive ones. Of course, this would have pay-offs, like some leaks in the security mechanisms.

Database usage would simplify the directory structure to just some tables. In the above case only one table would be enough. Additionally, database could give better opportunities for extending the system for new services. And, it would be possibly just an addition of a table. Beside its simplicity, authorization provided by the databases would be very useful for the user authentication through the CPL User Editor.

In Table 4-3, a simple table structure is presented for a database usage. And in Table 4-4, some example entries from the database are presented. Each entry in the database consists of five fields: The domain name, the user name, the script name, the script itself and the active flag. The domain name, the user name and the script name must be unique, i.e. there can be only one entry with the same domain, user and the script name. Same as the flat files, there can be only one active script for one user in the database. Active scripts are marked with a “1” in the “active” field. So, each user can have at most one script marked with “1” in the “active” field.

Column Name	Type
Domain	Text
User	Text
ScriptName	Text
Script	Text
Active	Number (1)

Table 4-3 A simple table structure for the database.

Domain	User	Script	ScriptName	Active
test.com	sertac	<cpl>...</cpl>	my_favorite_script.cpl	1
test.com	sertac	<cpl>...</cpl>	my_alternate_script.cpl	0
test.com	sertac	<cpl>...</cpl>	my_alternate_script_2.cpl	0

Table 4-4 Example entries in the database.

4.5.2 Active vs. Passive Repository

When a user loads his/her script to the database, and activates it, or deactivates it, the CPL flag in the Location Database has to be set accordingly. The flag could be set directly by the user interface or by the CPL Repository.

If the user interface would set the flag in the Location Database, then it would require information about the location (address) of the Location Database. However, in the practical call flow, the user interface does not need to know the location of the SIP Server or the Location Database. Additionally, a direct communication between the user interface and the Location Database would complicate the user interface, and additional configuration would be necessary for this purpose. This solution is illustrated in Fig. 4-7.

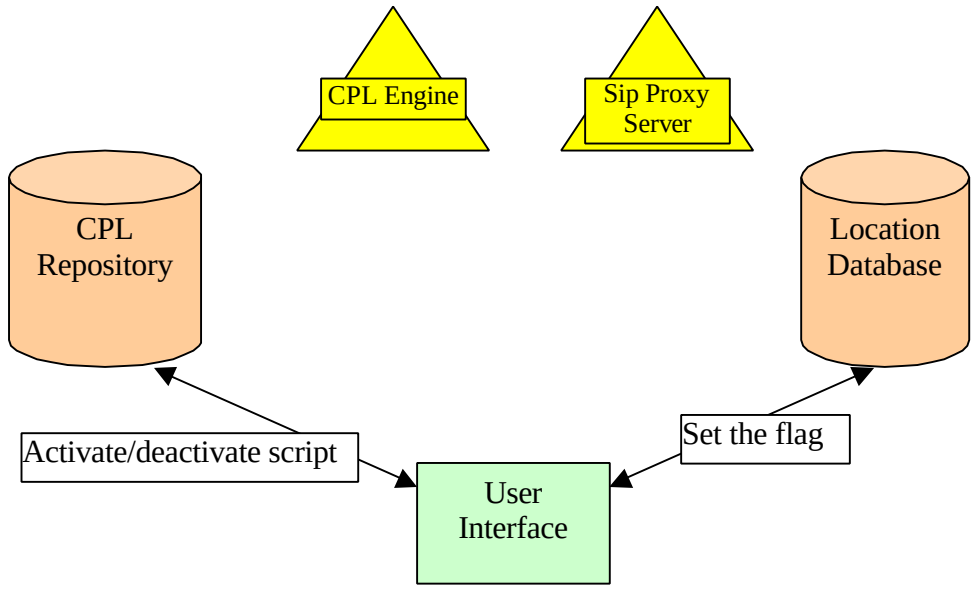

Fig. 4-7 User interface sets the CPL Flag in the Location Database

A better solution can be achieved using an active CPL Repository. In this case, whenever a new script comes in or whenever a script is deactivated, the CPL Repository informs the CPL Engine and the CPL Engine informs the SIP Server and the SIP Server sets the flag in the Location Database. This simplifies the user interface and moves some additional configuration from the user side to the repository side, since the user interface does not need the address of the Location Database. Fig. 4-8 illustrates the flow of the communication from the user interface to the Location Database.

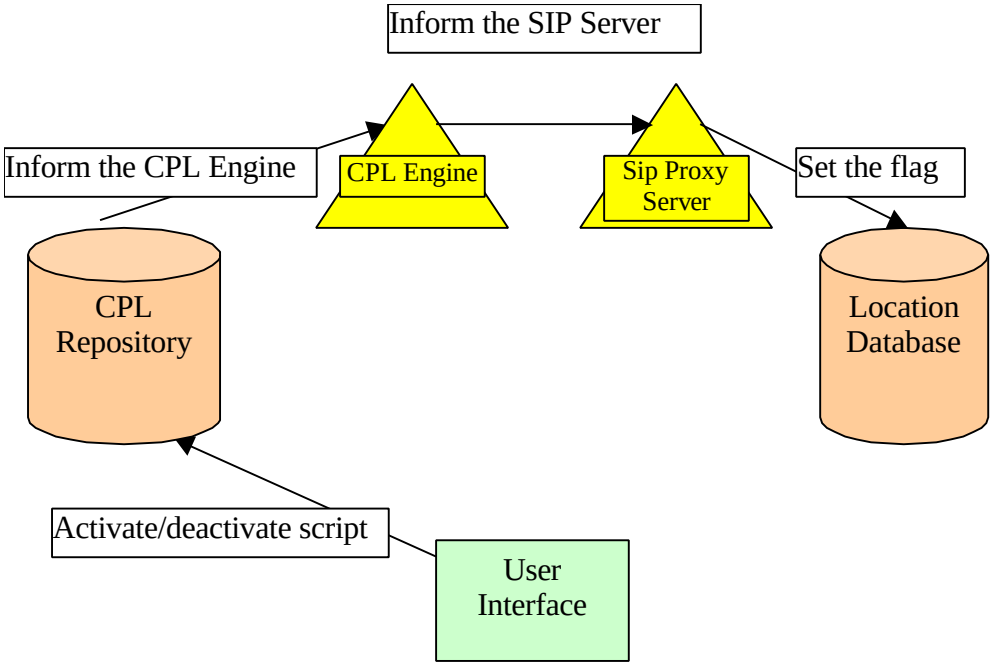

Fig. 4-8 Flow of script activation/deactivation from the user interface to the Location Database

This solution complicates the repository, and the flow of the information would have a longer path, as it is seen in the figure above; but this is preferable to having a more complicated user interface.

4.6 CPL User Editor

CPL User Editor is the only part used by the end-user. This was out of the scope of this thesis, however it forms an important part of the complete project. Originally, the CPL Engine can work without the CPL User Editor for evaluation purposes, but adding and updating the users' CPL scripts in the CPL Repository would be more difficult without the CPL User Editor.

CPL User Editor is a graphical user interface (GUI) for the end users to create the CPL Script and load it to the CPL Repository. The user does not need to know anything about the CPL, or the overall mechanism, which handles the calls, but he/she only defines his/her requests for an incoming/outgoing call by using a drag & drop GUI. The CPL User Editor is responsible for creating the CPL script from the user input and validating it through the XML parser by using the CPL DTD. To get more information, the web site of the developers of the CPL User Editor may be visited [28].

4.7 The relations of the components

In the previous chapters only distributed architecture and one to one relations have been discussed. However, there could be some other possibilities like n to one relations, and non-distributed architectures, as well.

As the first possibility, more than one SIP Server may use the same Location Database. In this case, the Location Database gets more complicated, but at the same time more effective, since the most effective resource usage would be achieved. Nevertheless, the

performance would be the trade off of this effectiveness. Fig. 4-9 represents one to three relation between the Location Database and the SIP Server.

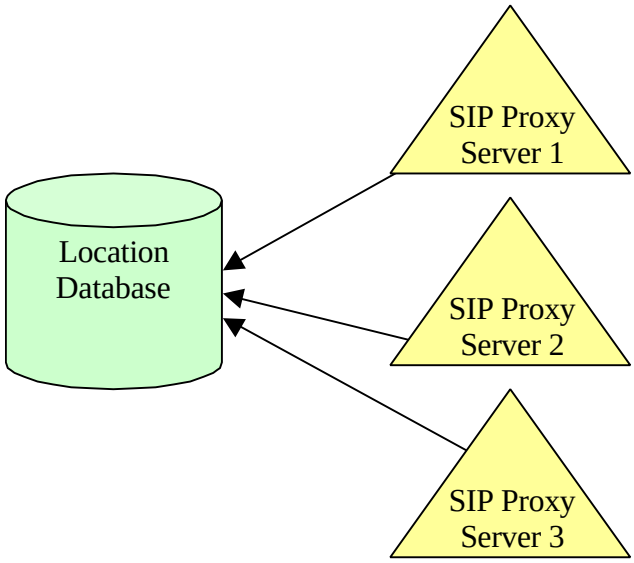

Fig. 4-9 One to 3 relation of the Location Database.

Similarly the CPL Repository could have one to n relation with the CPL Engine. In other words, more than one CPL Engine could use the same CPL Repository to get the users' CPL scripts. A similar picture to the previous one can be seen in Fig. 4-10 for one to three relation between the CPL Repository and the CPL Engine.

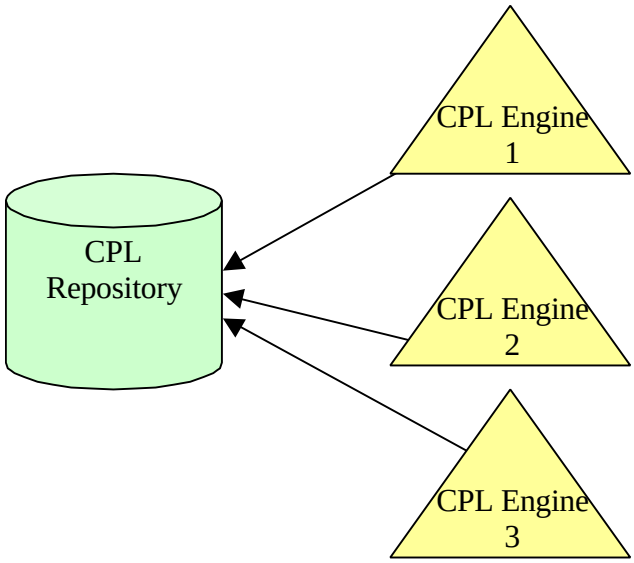

Fig. 4-10 One to 3 relation of the CPL Repository.

Another approach is to implement the Location Database and the CPL Repository in the same database. Fig. 4-11 illustrates two different architectures having the CPL Repository and the Location Database combined in one database.

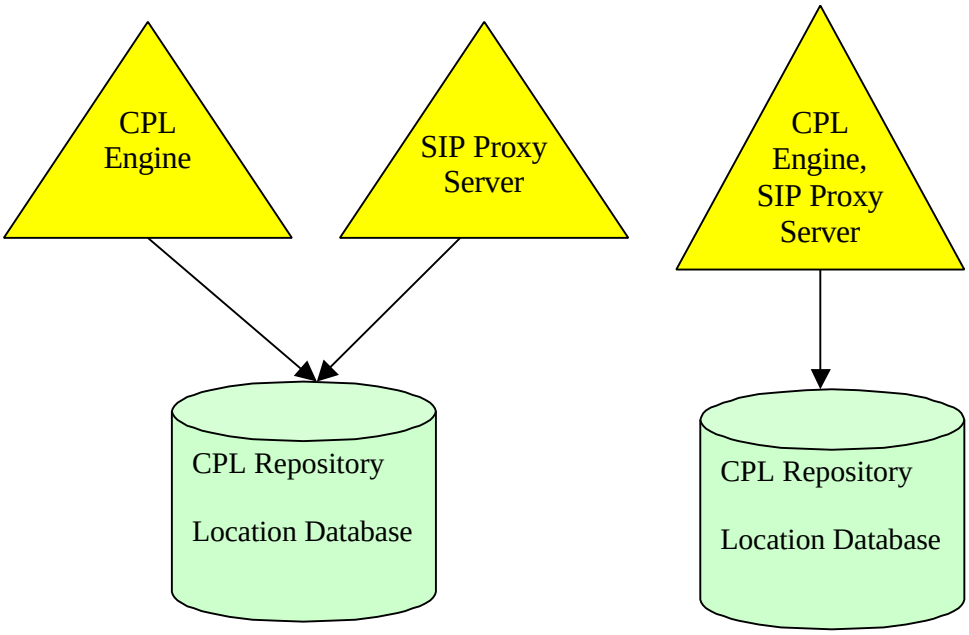

Fig. 4-11 Two alternative architectures.

At the left part of the Fig. 4-11 the CPL Repository and the Location Database are combined in one database. The CPL Engine and the SIP Server accesses to the same database, but asking for different information: The former one asks for the CPL script of a user, while the later one asks for the location of a user, and checks if the user has got a CPL flag in the database.

At the right part of the Fig. 4-11 the CPL Repository and the Location Database are combined in one database, as it is the case for the previous picture. Beside that, the CPL Engine and the SIP Server are combined in the same server. In this architecture, the SIP User could put his/her CPL script directly with the SIP REGISTER message. All the process would be controlled in one server. However, this would remove the flexibility of creating services by different providers, and would increase the complexity of the server. Additionally, new services could not be introduced independent from the SIP Server.

5 Tools used in the project

In this chapter, tools used for the implementation of the project are going to be briefly discussed. Although, the word “tools” may mislead the reader, there is some fundamental information about Common Object Request Broker Architecture (CORBA) and Light Weight Directory Access Protocol (LDAP), as well.

The first subchapter deals with the XML Parser, which is necessary to parse the XML documents, the CPL scripts in this project. Then, in 5.2 a detailed discussion of CORBA is given, and in 5.3 LDAP is briefly explained. The reader is suggested to read the appropriate references for a better understanding of both CORBA and LDAP.

5.1 XML Parser

Since the CPL is an application of XML, parsing the XML tags and values would be necessary. Because of very promising features provided by The Apache Xerces C++ project, the Apache Xerces C++ parser has been used to parse CPL scripts in this project. Xerces-C is a validating XML parser written in a portable subset of C++ [1]. Its main features are:

a. Conforms to XML Spec 1.0
b. Tracking of latest Document Object Model (DOM) Level 1.0, DOM Level 2.0, SAX/SAX2 specifications.
c. Source code, samples, and documentation are provided.
d. Programmatic generation and validation of XML
e. Pluggable catalogs, validators and encodings
f. High performance
g. Customizable error handling

Additionally, this parser supports many platforms, which could be very useful for porting the project to another platform. This project has been developed on Windows NT platform, keeping in mind that the system would be ported to Sun Solaris, sometime in the future.

Another important concept for parsing an XML document is using the DOM (Document Object Model), or the SAX (The Simple API for XML) approach. Following three chapters (5.1.1, 5.1.2 and 5.1.3) discuss the two models. Both approaches have got some advantages and disadvantages. However, DOM approach would be the best approach for the CPL script processing, since the states of the CPL Script should be kept until the end of the call processing.

5.1.1 Document Object Model (DOM)

As it is explained in [16], DOM is a platform- and language-neutral interface that allows programs and scripts to dynamically access and update the content and structure of documents. DOM has got two versions up to now: DOM Level 1 and DOM Level 2. They both have been standardized by the World Wide Web Consortium (W3C). The XML Parser used in this project conforms to both DOM Level 1 and DOM Level 2.

DOM is simply an application-programming interface for valid HTML and well-formed XML documents. It defines the logical structure of documents and the way a document is accessed and manipulated. In the DOM specification, the term "document" is

used in the broad sense - increasingly, XML is being used as a way of representing many different kinds of information that may be stored in diverse systems, and much of this would traditionally be seen as data rather than as documents. Nevertheless, XML presents this data as documents, and the DOM may be used to manage this data [16].

DOM is mainly a tree based API, which means after parsing a document, a tree of the document is established. This tree holds all the information in the document. It is possible to walk the tree from roots to the leaves, and vice versa.

5.1.2 Simple API for XML (SAX)

SAX, the Simple API for XML, is a standard interface for event-based XML parsing. It has been developed collaboratively by the members of the XML-DEV mailing list [17]. Similar to DOM, it has got two versions up to now: SAX1 and SAX2. Again the Xerces-C Parser conforms to these two versions.

SAX is mainly an event based API, which means while parsing a document, the application is reported about the parsing events. Then, the application takes the necessary steps according to the event reported.

5.1.3 DOM vs. SAX

DOM is comparably easy to implement, and use. Once the script is parsed, we get a tree of the XML document and then we can access all leaves (children) of the tree.

SAX does not give access to the already parsed tags, so they have to be stored explicitly, which is very close to the DOM approach. Otherwise, we may need to parse the same script many times.

Speed of SAX is faster for only one element of the script, but if we consider the above reasons then DOM would be expected to be faster, and more effective.

In Table 5-1, the reader can see the performance test results for SAX, SAX2 and DOM approach. SAX2 is the second version of SAX. Here three approaches were tested for parsing and printing the same XML document. Two different computers were used for testing: C1, C2. The reason for using two different computers was, first of all, to make sure that the results are computer independent. Another reason was that C1 was a very widely used computer by the department, and because of that it could mislead the measurement results. However, C2 was a standalone computer, but a slower one.

Specifications of the computers are given below:

C1:

Type:	SUN Enterprise 250
Processor:	250 MHz UltraSparc
RAM:	256 MB

C2:

Type:	SUN Ultra 1
Processor:	166 MHz UltraSparc
RAM:	128 MByte

Measurements were made to calculate the (average) time required for printing the first element in the document to the screen, and the (average) time required for printing the whole document to the screen.

As it is seen clearly, SAX and SAX2 is almost always faster than the DOM approach. However, the speed difference is not very much. SAX takes nearly 75% time of DOM approach to complete the printing of the document. Nevertheless, this speed advantage of SAX does not make it suitable for this project, since the status of the document should be kept till the end of the call, and probability of going back to the already parsed elements is quite high. If it gets necessary to go back to an already parsed element in SAX approach, then all the document must be re-parsed, since SAX approach does not offer the ability to go back to the already parsed elements. However, DOM approach offers this ability that makes the DOM approach more suitable for this project.

	SAX Print		SAX2 Print		DOM Print	
Machine	First (ms)	All (ms)	First (ms)	All (ms)	First (ms)	All (ms)
C1	25	50	35	62	47	65
C2	25	55	37	65	50	70

Table 5-1 Performance test results of DOM and SAX approaches.

5.2 Common Object Request Broker Architecture (CORBA)

CORBA is a method of providing communication among the distributed systems. Distributed systems can be understood as the computer networks, as well. In the following subchapter a brief history of the distributed systems is explained.

The Object Management Group has developed CORBA, and it is an “open, vendor-independent architecture and infrastructure that computer applications use to work together over networks.“ [31] The main idea is to let a computer application interoperate with a CORBA-based program independent from the platform, network, programming language, and the vendor of the program. CORBA uses the General Inter ORB Protocol (GIOP) and the Internet Inter ORB Protocol (IIOP), where ORB stands for Object Request Broker. “GIOP specifies transfer syntax and a standard set of message formats to allow independently developed ORBs to communicate over any connection-oriented transport”, while “the IIOP specifies how GIOP is implemented over Transmission Control Protocol/Internet Protocol (TCP/IP).” [2]

In 5.2.2 alternatives of CORBA are discussed, while in 5.2.3 advantages of CORBA are listed. Later in 5.2.4 the CORBA implementation used in this project (TAO CORBA) is explained. The last two subchapters give some information about two CORBA services, the Naming Service and the Event Service.

5.2.1 History of Distributed Systems

Although the name ”Distributed Systems” is comparably a new expression, the concept of distributed systems was in use for a long time. It started with the monolithic systems and the mainframes, and continued with the client/server architecture and the multi-tier client/servers. The next generation was the distributed systems.

Beginning with the main frames it was possible to serve large number of users and to manage the users centrally. Software systems were often monolithic, i.e. the user interface, and the data access functionality were all contained in one large application. The mainframe architecture can be seen in Fig. 5-1.

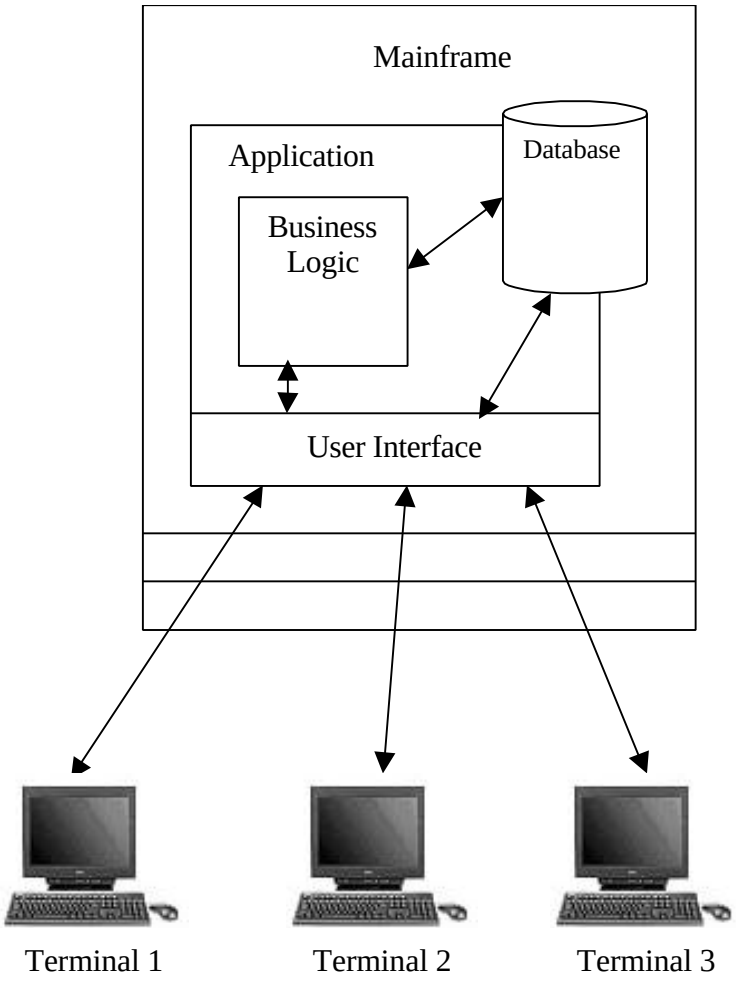

Fig. 5-1 Mainframe architecture

The next step was the client/server architecture, which took some responsibilities of the mainframes to the client PCs. And of course client/server architecture was much more cost effective than the mainframes. The traditional client/server architecture is at the same time called as two-tier client/server architecture. In this approach, database access functionality and the business logic are contained in the client component, and the database itself and the application are contained in the server component. This architecture is seen in the Fig. 5-2. The client/server is generally dependent on any changes in the business logic, or in the database access. That is the reason why any changes in any of these components would generally break the client component.

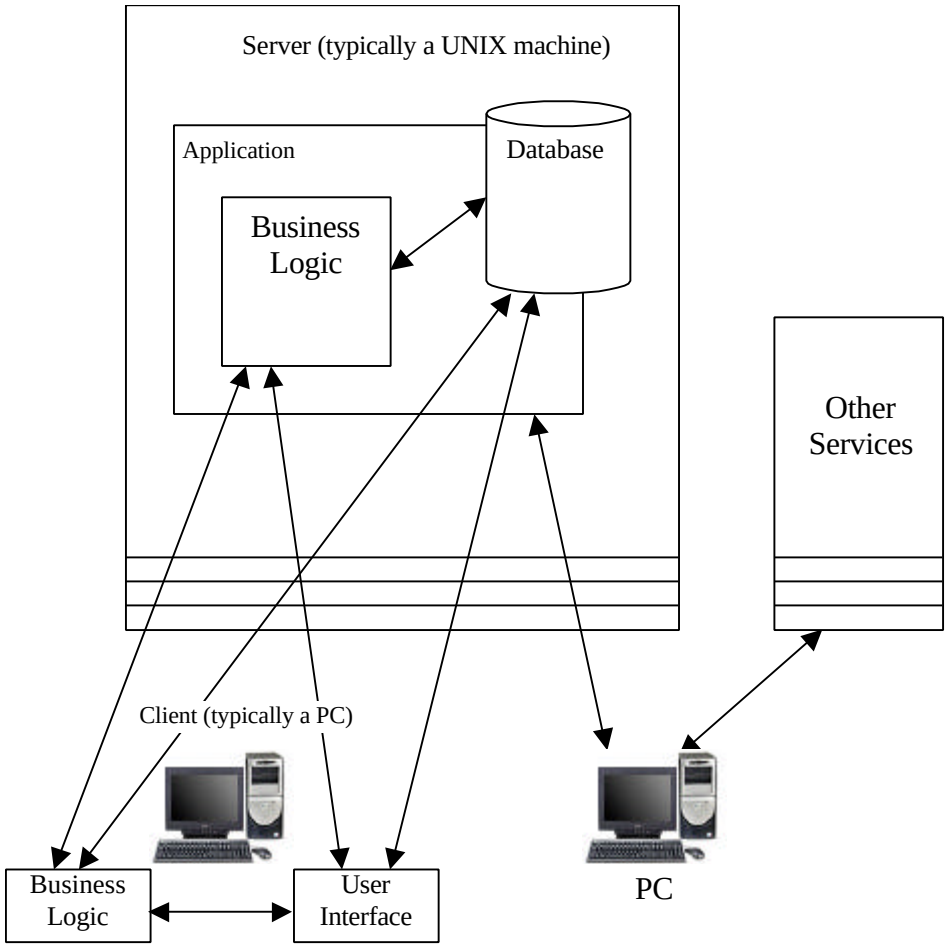

Fig. 5-2 A two tier client/server architecture

The multi-tier client/server architecture distributed the system into more tiers. As an example, in three-tier client/server architecture, illustrated in Fig. 5-3, the system is partitioned into three tiers: the user interface layer, the database access layer and the business rules layer. This architecture makes the application less fragile, since the client is more insulated from the changes in the rest of the application.

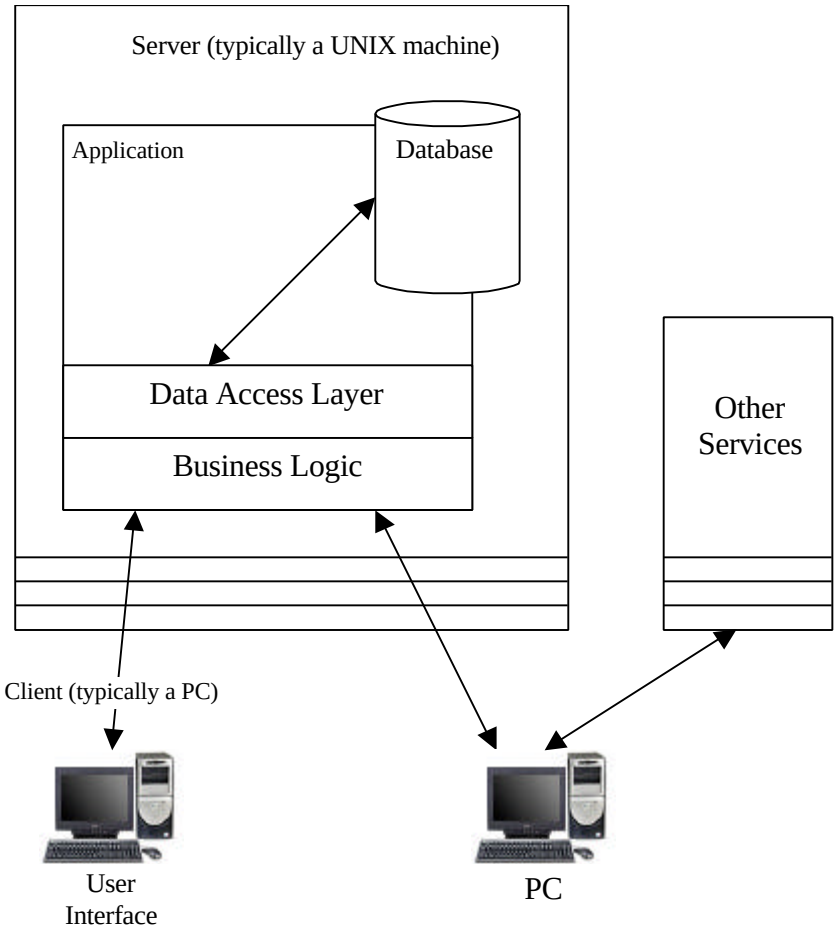

Fig. 5-3 A three tier client/server architecture

The next generation was the distributed systems. In the distributed systems, all functionality of the application is exposed as objects. And, each object can use the services provided by other objects. This architecture also changes the typical roles of the client and the server, since generally each client behaves as a server at the same time.

5.2.2 Alternatives of CORBA

Of course, CORBA is not the only choice for providing the communication between the CPL Engine and the SIP Server. There are some alternatives of CORBA, which could be considered for the design purposes.

5.2.2.1 Socket Programming

First alternative could be the socket programming. A socket can be seen as a channel through which applications can communicate with each other. If the developer decides to use socket programming, then he/she has to write and read data directly to and from the sockets.

This is the most straightforward way of communications between two applications, but the Application Programming Interface (API) for socket programming is very low-level. So, it

has got many drawbacks, like complexity of handling data types, complexity of handling different types of machines and operating systems, complexity of dealing with different programming languages at different applications, and so on.

Although socket programming could result in with very efficient applications, it is not suitable for developing complex applications, like CPL Engine. This is because, all the levels of the communication has to be handled by the developer in socket programming. For example, data from the sockets must be read manually, data to the sockets must be put manually.

5.2.2.2 Remote Procedure Call (RPC)

Remote Procedure Call is one higher level than the raw socket programming. Using RPC, developer defines a function, and generates the code, which makes the function look like a normal function to a caller. However, in the hidden part, the function uses sockets to communicate the application, but this time it is easier than basic socket programming, since RPC provides a function oriented interface. RPC is quiet powerful for client/server architecture.

5.2.2.3 OSF Distributed Computing Environment (DCE)

The Distributed Computing Environment (DCE) is a set of standards prepared by the Open Software Foundation (OSF), and it includes a standard for RPC, as well. Although these standards have been available for long time, they have never gained wide acceptance, and they are used very rarely.

5.2.2.4 Microsoft Distributed Component Object Model (DCOM)

DCOM is Microsoft's entry into the distributed computing environment. It offers similar capabilities like CORBA. However, it is mainly established for Microsoft operating systems (Windows 95 & NT), and it causes some problems with other operating systems. But, Microsoft is working on this subject to make DCOM available for all operating systems.

5.2.2.5 Java Remote Method Invocation (RMI)

Java Remote Method Invocation (RMI) is the last alternative of CORBA for distributed systems. RMI is very similar to CORBA, too. It has got an advantage against CORBA that it supports passing objects by value. However, at the same time, it has got the disadvantage of being a Java only solution.

5.2.3 Advantages of CORBA

In this project CORBA has been used to establish the communication between the CPL Engine and the SIP Server, because of the following advantages:

a. The CPL Engine uses the services of SIP Server, as they are its own functions, and classes. For example, making a proxy call is as easy as calling the corresponding function of the SIP Server.

b. The CPL Engine does not need to know anything about the location of the SIP Server (With the use of Naming Service).

c. The implementation language of SIP Server is not important for the CPL Engine.

d. Implementation of CORBA communication is easier than the alternatives.

5.2.4 The TAO CORBA

TAO stands for The ACE ORB, where ACE is the abbreviation of the Adaptive Communication Environment. ACE is a research group of the University of California and the Washington University, St. Louis. “TAO is a real-time implementation of CORBA built using the framework components and patterns provided by ACE.” [33]

TAO is an open source, vendor independent, and freely available high-quality CORBA-compliant middleware platform. It is continually improved to get a better performance, and to provide new services. Following are the main features of the TAO CORBA to be selected for this project:

- Free license
- C++ support (The programming language for this project was C++.)
- Open source
- Continuing improvements (2 new versions came out during the project work, and many bugs were fixed. Expected to get better in coming years.)
- Very good support (through an e-mail list, the TAO development group offers a very good support to TAO customers. Any bugs can be reported to and fixed by this group.)

TAO CORBA implementation has been used in this project to provide the CORBA communication between the SIP Server and the CPL Engine. Additionally, the naming service and the event service, provided by TAO, were integrated into the project for a better implementation. These two services are discussed in following two subchapters.

5.2.5 The Naming Service

There are two ways of finding a CORBA object over the net: One is using the Inter Object Reference (IOR) files, and the other one is using a naming service, which keeps the records of the references to the objects.

By using an IOR file, the server writes its reference to a file, and this file is transferred to the client by simply copying it to the client machine. Then the client is started by using this IOR file, which is used communicate the server.

The above process seems to be very complicated for a real life application, and actually it is. The alternative of this approach is the naming service. By using the naming service, server registers itself to the naming service and the client queries the naming service for the server. After getting the reference of the server client communicates to the server directly. Naming Service works very similar to the DNS servers in the internet world: DNS Servers resolves the names to the IP addresses, while the Naming Service resolves the names to the object references.

In this project, both the CPL Engine and the SIP Server are clients and servers at the same time. So, they both register themselves to the naming service and query the naming service for each other.

5.2.6 The Event Service

There are different ways of invoking requests in the distributed systems. Synchronous requests and asynchronous requests are the basic methods.

With synchronous requests, the client and the server are synchronized to each other and the client makes a request on the server and waits for the result of the request. This is more like a normal function call in a functional programming language. However, with this method client and the server have to hold the object references to the target objects, which means they should be aware of the destinations of the requests.

Using asynchronous request method gives the ability of non-blocking function calls. In this case the client makes the request, but it does not wait for the result of the request, instead it checks for the result of the request at a later time in the program execution.

The Event Service is an intermediate solution for decoupled requests in the distributed systems. Using the event service allows the clients and the servers to communicate without being aware of each other. In other words, clients make a request to a virtual server, and this virtual server passes the request to the real servers, if they exist. In case of no servers the client continues with its default action.

With the event service two new terms are used instead of client and server: supplier and consumer. Suppliers produce events, and consumers receive them. An event channel is used to pass the events from the suppliers to the consumers. The event channel plays the central role in the Event Service. It is responsible for supplier and consumer registration, timely and reliable event delivery to all registered consumers, and handling of errors associated with unresponsive consumers. [2]

The event service provides two mechanisms for handling the events: the push model and the pull model. With the push model, suppliers push events to the event channel, and the event channel pushes events to consumers. Fig. 5-4 illustrates the push model of the event delivery.

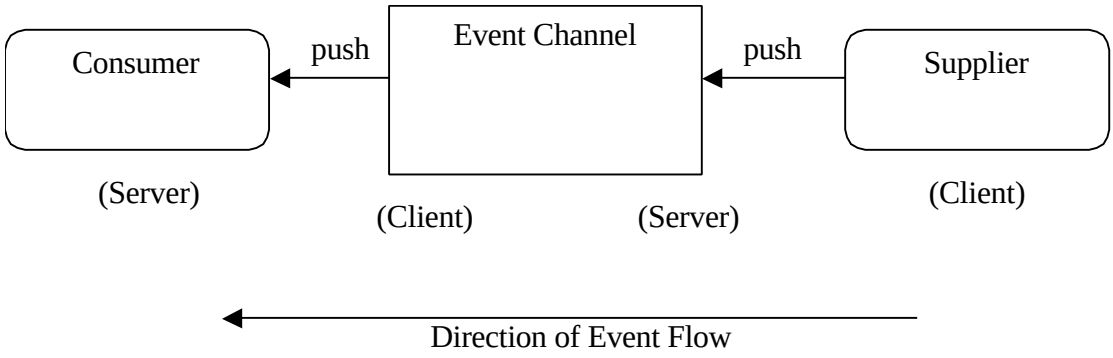

Fig. 5-4 Push model for Event delivery.

With the pull model, the actions take place in the opposite direction: Consumers pull events from the event channel, and the event channel pulls events from the suppliers. This model is illustrated in Fig. 5-5.

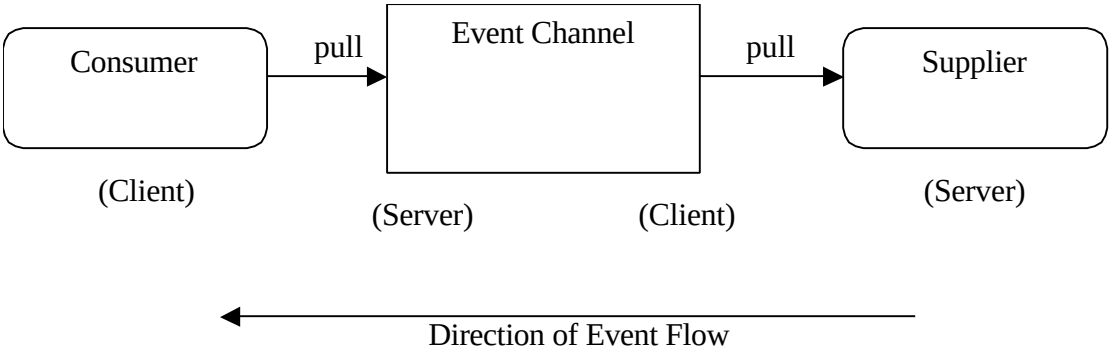

Fig. 5-5 Pull model for the Event delivery.

The push model is more like a real function call in a program, and it fits the natural event flow, as well. In case of no events in the supplier, consumers and the event channel would try to pull an event, although there would not be an event produce by the suppliers.

Although the Event Service has been evaluated during the project work, it has not been included in the overall implementation because of the time restrictions. In other words, it was evaluated separately from the real implementation.

5.3 Lightweight Directory Access Protocol (LDAP)

Lightweight Directory Access Protocol (LDAP) is designed to provide access to directories supporting the X.500 models, while not incurring the resource requirements of the X.500 Directory Access Protocol (DAP). [7]

LDAP is designed to run over TCP, which makes it ideal for Internet and intranet applications.

LDAP arranges the directories as a tree. It starts with a root and goes till the leaves. In Fig. 5-6, an example LDAP tree is illustrated. There, the directory structure starts with the root directory, and then comes the domains, and users and scripts follow them. This is at the same time the directory structure proposed for the CPL Repository.

LDAP provides authentication mechanism, as well. Using this authentication the CPL Editor can be authenticated for downloading and uploading the CPL scripts.

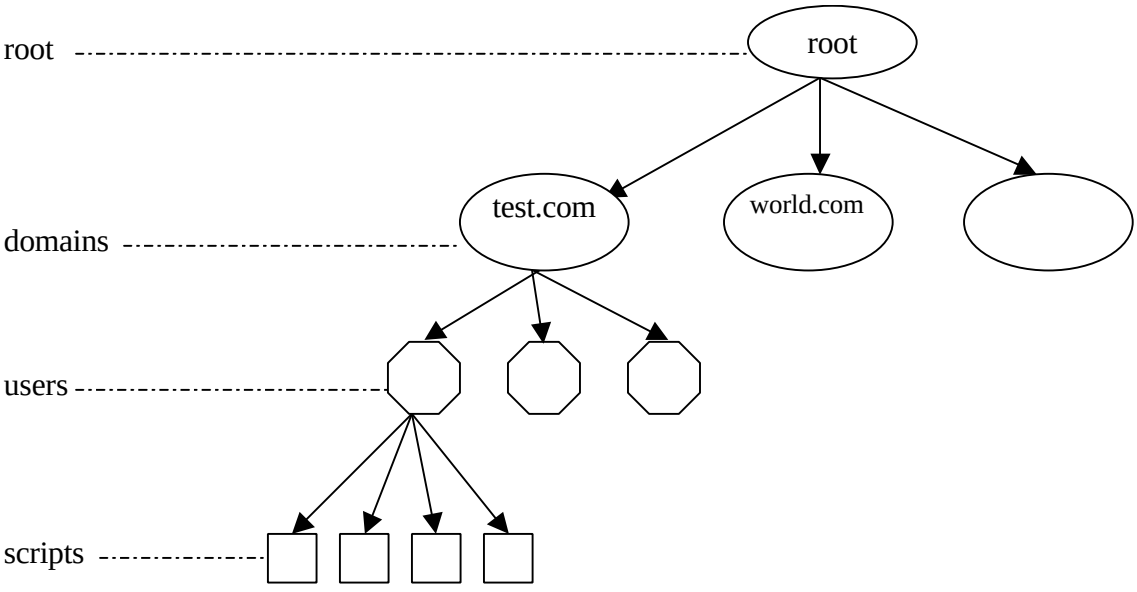

Fig. 5-6 An LDAP Tree.

6 Implementation

One of the most important purposes of this thesis was to evaluate the CPL in the IP telephony environment. Because of that an evaluation environment has been established. For this environment, all the components given in Fig. 4-1 had to be implemented. The SIP Server was already implemented by another group at Siemens AG in Austria, but it had to be modified for the needs of the CPL Engine. The Location Database was also implemented as a part of the SIP Server. However, the CPL Engine, the CPL Repository, and the CPL User Editor had to be implemented completely from scratch.

Coming subchapters give some information about the implementation of the components. And chapter 6.2 goes into details of the CPL Engine implementation.

6.1 SIP Server

At the beginning of the project SIP Server was already implemented and making telephone calls over IP networks was possible. The implementation language for the server was C. However, since an Object Oriented (OO) approach was used for other components, SIP Server had to be modified to reflect the changes necessary for the communication between the CPL engine and the SIP Server.

The SIP Server is a proxy server as specified in the chapter 2.2, however, it actually does the job of all three types of the servers specified in SIP. So, the SIP Server is a register server, a proxy server and a redirect server at the same time. It depends on the request arriving at the server. If the request is a REGISTER request, then it behaves as a SIP register server. If the request is an INVITE request, then it behaves as a SIP proxy server. If the signaling action sent from the CPL Engine to the SIP Server is a redirection, then it behaves as a SIP redirection server. For more information about these three types of SIP servers, the reader might refer to the chapter 2.2.

The SIP Server works as a state machine (see Fig. 6-1). It starts with an idle state. Whenever a SIP INVITE message arrives in the SIP Server a transaction is started for this call. After starting the transaction, the state is changed to “locating destination”, and the location database is queried for the corresponding user. If the user exists in the location database and he/she does not have any active CPL Script, then the call is forwarded to the appropriate location and the state goes back to “idle”.

If the user exists in the location database and the user has got an active CPL Script, then the state is changed to “CPL” and the CPL Engine is contacted for the appropriate signaling action. For a proxy signal, the state continues to be in the “CPL” state. In this state the user location is proxied. If the connection is established successfully, the CPL Engine is contacted again to destroy the call, and the state is changes back to the “idle” state. If the connection is not established successfully, then the CPL Engine is contacted again for the appropriate signaling action.

If the signaling action from the CPL Engine is “redirection”, then the call is redirected to the corresponding location and the SIP Server goes back to the “idle” state. Similarly, if the response from the CPL Engine is “reject”, the call request is rejected and the state is set to “idle”.

The “CPL” state seen in Fig. 6-1 has been defined for the services provided by the CPL Engine.

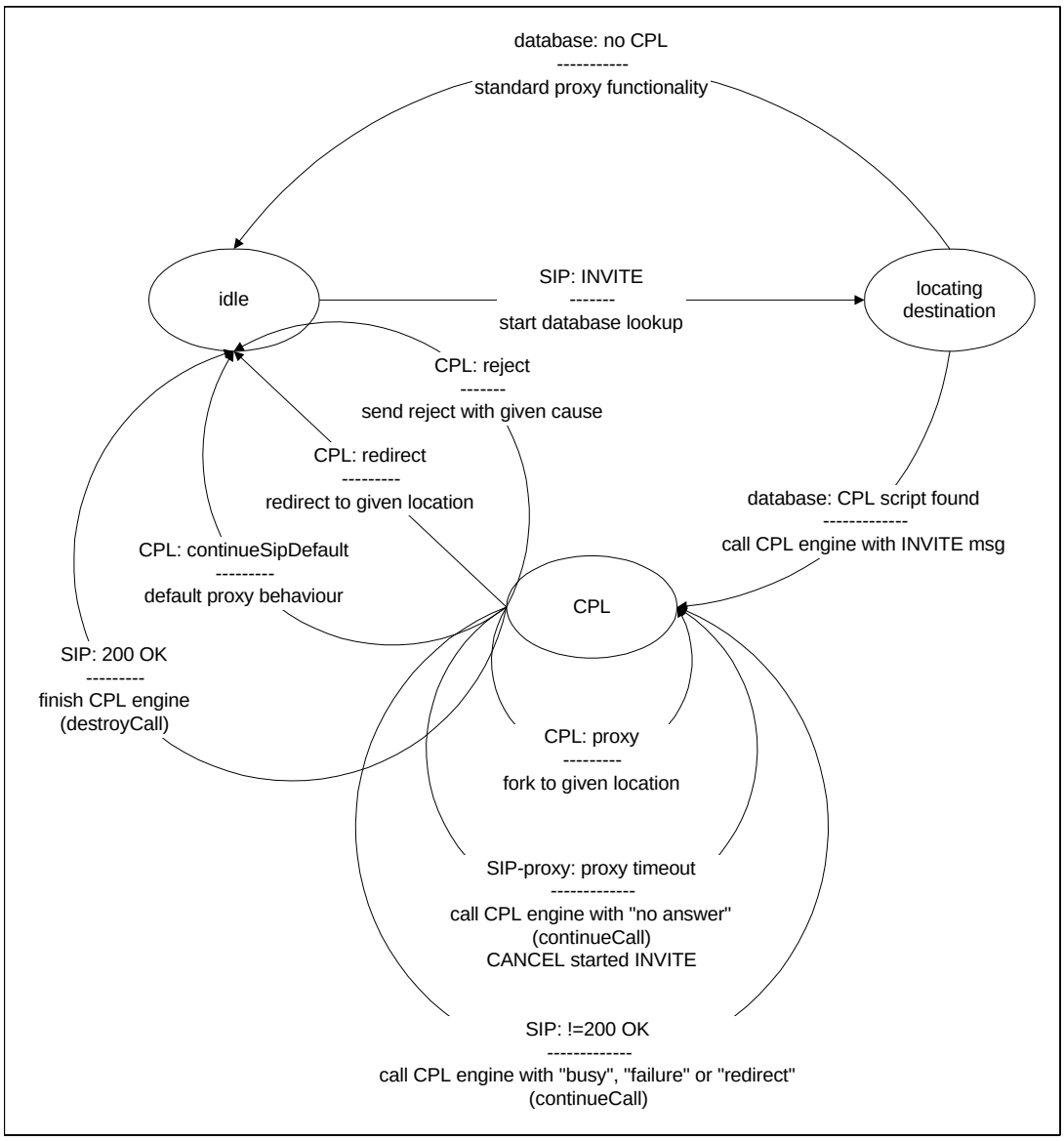

Fig. 6-1 State diagram of the SIP Server.

6.2 CPL Engine

The CPL Engine has been implemented as a server. CORBA has been used as the communication interface between the SIP Server and the CPL Engine, while the communication between the CPL Engine and the CPL Repository uses an LDAP interface.

In the following subchapters, first a general overview of the CPL Engine implementation is explained, and then the details of the CORBA interface and the classes created for implementing the CPL Engine are discussed.

6.2.1 General overview

The programming language for the CPL Engine was C++, and an object oriented (OO) approach was used. The alternative of C++ as an OO programming language was using JAVA, but because of the performance considerations JAVA would not be a good selection.

The CPL Engine assumes that CPL scripts provided by the CPL Repository are valid and well formed. To speed up the script parsing process, the CPL Engine does not check the

validity of CPL scripts. Fortunately, the CPL User Editor validates CPL scripts before loading them into the CPL Repository.

Most important CPL functionalities have been implemented and evaluated in this project. However, the following functionalities of the CPL have not been implemented in this project:

- **Outgoing calls**: The CPL Engine interprets only incoming calls, and the SIP Server does not contact to the CPL Engine for outgoing calls.
- **Time switch**.
- **Priority switch**.
- **Location lookup**.

6.2.2 CORBA Interface

Considering the CORBA interface between the CPL Engine and the SIP Server, asynchronous function call would be a perfect solution for this project since the SIP Server would not be blocked during a request to the CPL Engine. However, because of the time restriction of the project, evaluation has been done with the synchronous function calls. Event Service has also been implemented and tested, but not included in the overall evaluation.

The ACE-TAO CORBA implementation has been used as the CORBA interface in the project. The Interface Definition Language (IDL) is used to define the CORBA interface. IDL is a special language for defining the interfaces, and according to the IDL compiler used, the code can be created in different languages. In this project the IDL compiler from ACE-TAO group created C++ code. The IDL definition for the CORBA interface between the SIP Server and the CPL Engine is given in the Appendix C.

Two interfaces (classes in C++) are defined in the CPL IDL: CallServer and SipProxyServer. CallServer is the interface used by the SIP Server to call the functions in the CPL Engine, and similarly SipProxyServer is the interface used by the CPL Engine to call the functions in the SIP Server.

CallServer interface defines three methods to be called by the SIP Server: startCall, continueCall, and destroyCall. Similarly, SipProxyServer defines four methods to be called by the CPL Engine: proxy, redirect, reject, and continueSipDefault.

In the following subchapter, some more details can be found about the classes defined for the CPL Engine.

6.2.3 Classes

Five classes have been defined to implement the CPL Engine: CPLServer_i, Configuration, CallServer_i, Call_i and TagTable_i. The CPLServer_i class is used to start the server. It first creates a configuration object and then calls the necessary methods of the configuration object to make the initial configuration. Then a call server object is created and started for listening to the SIP Server. The CPLServer_i class handles the manual inputs from the keyboard, as well. If the input is quit, then the server is stopped.

The Configuration class makes all the initial configurations necessary for the start of the server, for example the path information for temporary files, the CPL Repository Server address, and so on.

The CallServer_i class corresponds to the call handler block in Fig. 4-4. It inherits from the class CallServer, which is defined by the CPL IDL and created by the ACE-TAO IDL compiler. All three methods of the CallServer class are re-implemented in CallServer_i that corresponds to the actions to be taken when a request comes from the SIP Server. Fig. 6-2 shows all the methods and the parameters of the CallServer_i class.

The CallServer_i class gets the object reference to SIP Server when one of the methods of the CallServer_i class is called by the SIP Server. This reference is given by the parameter “caller”. When a new call comes with the startCall method, a call object is created for this call and registered to the call vector, which keeps the references to the call objects. The caller reference is passed to the call object, too.

continueCall method is used by the SIP Server to respond to the CPL Engine with the result of a proxy action. If the result of the proxy action is successful, then destroyCall method is used to end transaction in the CPL Engine already started for the corresponding call.

Three private methods are provided by the CallServer_i class to handle the registry of the calls. A vector of Call_i class is used to hold information about the active calls. All these three methods use this vector to access and modify the call registry.

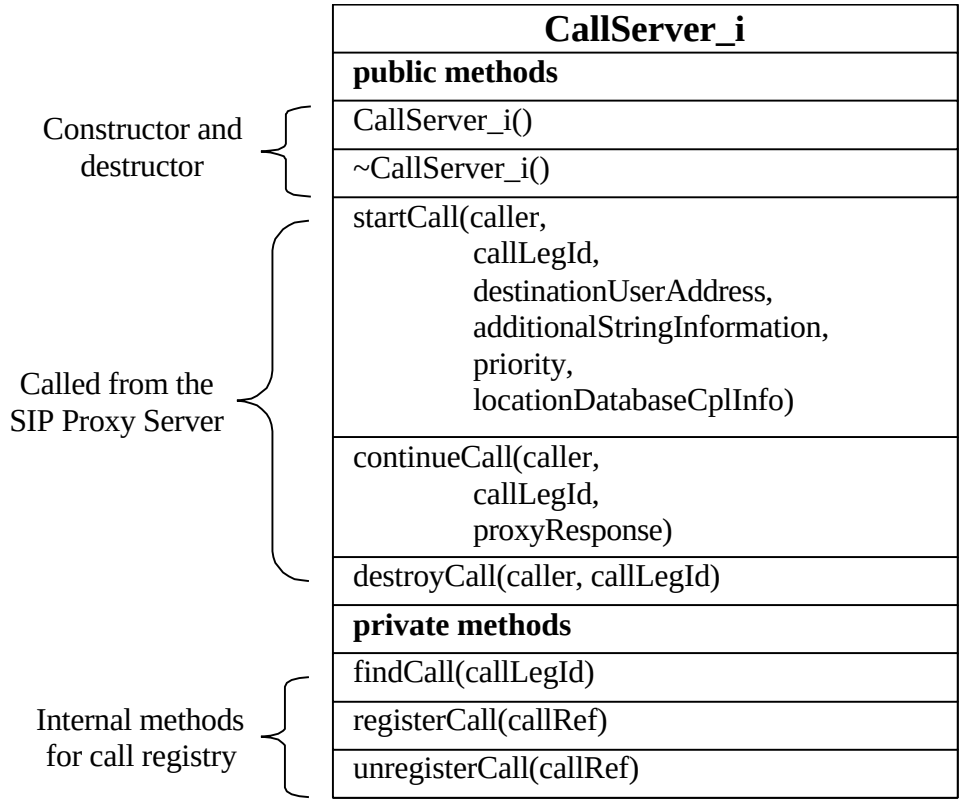

Fig. 6-2 Object model of the CallServer_i Class.

The Call_i class corresponds to the interpreter in Fig. 4-4. It provides the necessary methods to get the CPL script, to make it parsed, to interpret it and to send the signaling actions to the SIP Server. Four signaling methods are implemented in the Call_i class: proxy, reject, redirect, continueSipDefault. The Call_i class, also, provides methods for non-signaling actions, such as sendMail and logFile. The public and private methods of the Call_i class are illustrated in Fig. 6-3.

Proxy, reject, redirect and continueSipDefault methods use the caller reference passed by the CallServer_I object to call the corresponding method in the SIP Server. The caller

reference is considered as a local object in the CPL Engine, although it refers to a remote object that is the SipProxyServer object in the CORBA interface.

addLocation method is used to add a URL to the location list to be sent to the SIP Server, while removeLocation method removes the specified URL from the location list. clearLocation method is provided to remove all the entries from the location list, and reset it.

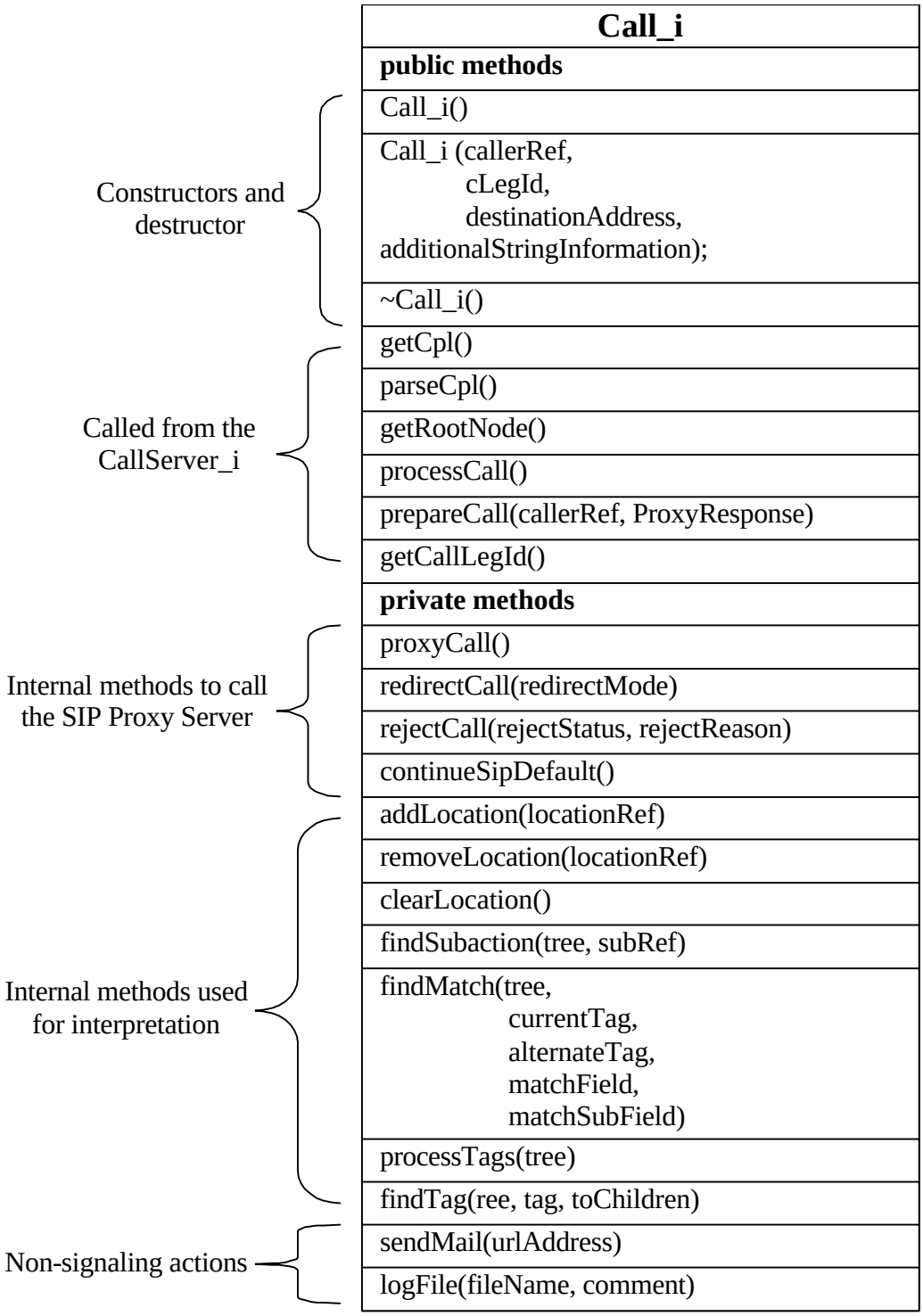

Fig. 6-3 Object model of the CallServer_I Class.

The TagTable_i class is used to classify the tags provided by the XML parser. It provides the getTagClass method to investigate the class of the tag, so that the necessary action would be taken.

6.3 CPL Repository

The CPL Repository has been implemented as a separate database from the Location Database. Both the flat file implementation, and the database implementation have been used to test the system. The LDAP Server has been used as the database for the CPL Repository.

The LDAP Server already offers object classes for saving the domain and user information. LDAP can offer means of restricting the reading and writing rights for single users, too.

For the CPL script a new object class has been defined in the LDAP Server. Below, the fields created for this object class can be seen. The “sipuser” field represents the user name of the owner of the script. The “domain” field defines the domain name that the user belongs to. The “cpl” field stores the CPL script of the user. The “cplplus” field was defined for the CPL User Editor since the CPL User Editor stores some extra information to identify how the script looks in the graphical user interface. The “active field is used to specify the active scripts in the repository. The input type “Case Ignore String” ignores the case information of the inputs, i.e. capital or lower case letters does not make any difference. However, “Case Exact String” type stores the case information, as well.

objectclass = cplScript	
sipuser	Case Ignore String
domain	Case Ignore String
cpl	Case Exact String
cplplus	Case Exact String
active	Integer

6.4 CPL User Editor

Another group at Humbold University in Berlin has implemented the CPL User Editor. To provide platform independence JAVA has been used as the programming language for the CPL User Editor. To get some more information about the CPL User Editor Implementation, the reader may visit the CPL User Editor web site at the following URL: http://www.informatik.hu-berlin.de/~xing/CPLEditor/.

In Fig. 6-4, a screenshot from the CPL User Editor is presented. With the help of this graphical user interface (GUI), users can create and modify their CPL scripts. Fig. 6-5 represents the CPL script created by the CPL Editor for the graphical view in the Fig. 6-4.

As it can be seen in the Fig. 6-5 a drag & drop mechanism has been used to implement the CPL User Editor. This allows the users simple creation and modification of CPL Scripts. Most of the tags are represented by icons, and the attributes of these tags can be accessed by right mouse click.

For example, an address-switch is represented by an icon, and an arrow originating from the address-switch icon represents an address tag. This simplifies the creation and the understanding of an address based decision.

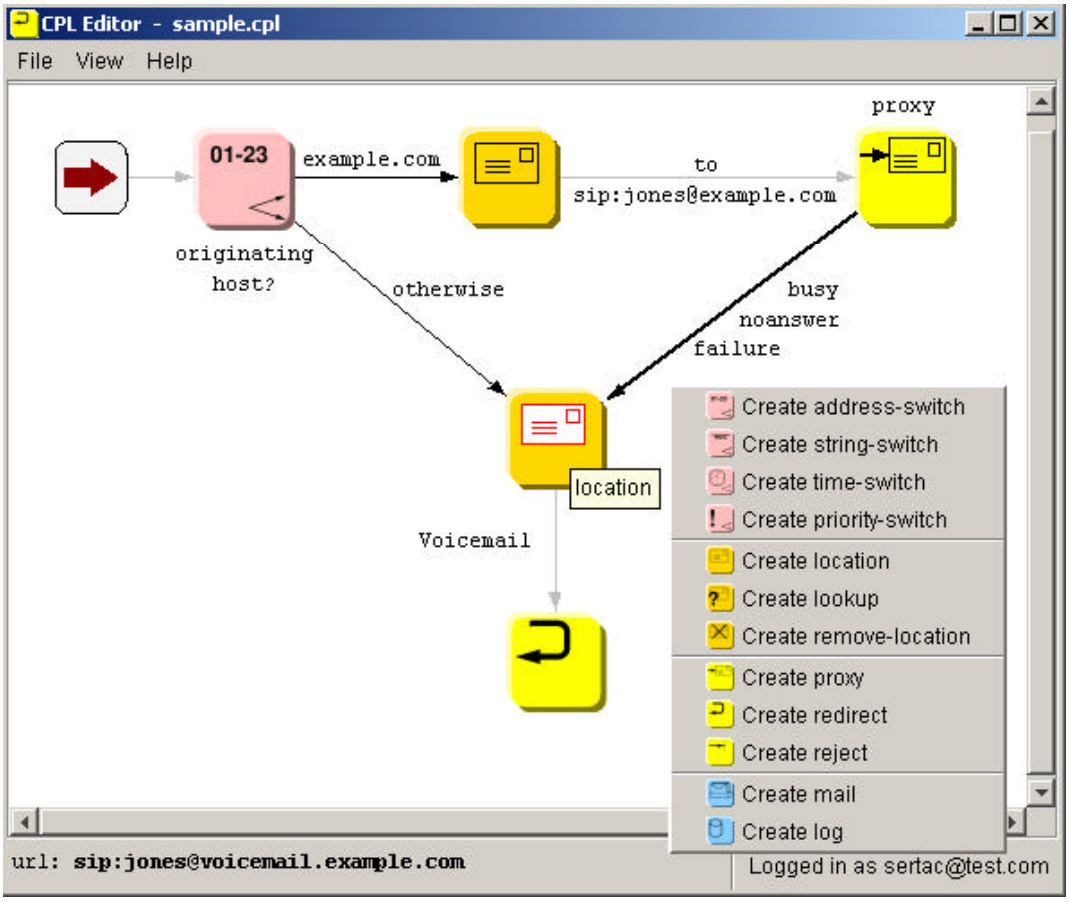

Fig. 6-4 A screenshot from the CPL User Editor.

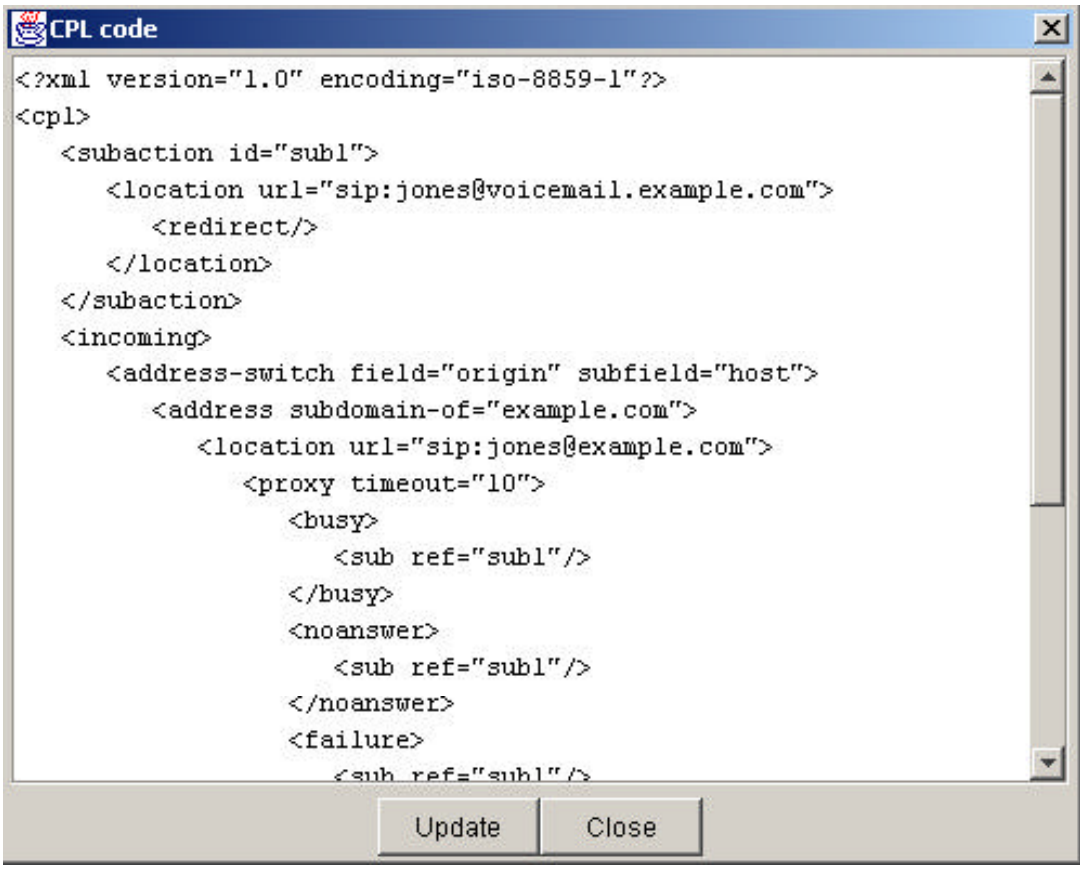

Fig. 6-5 The CPL Script created by the CPL User Editor.

Results

In this chapter, the reader is going to get some information about the evaluation results, such as the end-user feedback about the project. Additionally, the performance measurements of the current implementation are going to be discussed in the chapter 6.6. And, of course, some suggestions about further studies are going to be given in this chapter.

Shortly summarizing, the evaluation results are very promising, and the performance results are not very bad. However, both of these can be improved by further work on the weak points of the project.

6.5 Evaluation Results

The project has been successfully completed, and presented at an International technology fair in Hanover in Germany (CEBIT 2001). This presentation was a good opportunity to get the end user and the operator feedback.

The end users (normal visitors at CEBIT) were very interested in the project. The ease of the script creation, and the services offered by the project were the most interesting points for the end users. For example, some people thought that the ability to reject or redirect a call according to the originating address (or originating user) would be very useful to avoid unwanted calls.

Another important feedback came from the telecom operators. They thought that offering these services would be an important challenge against their competitors. The first provider offering such services would get the most benefit of the IP telephony.

Beside the end user aspect, two main points have been evaluated in this project. First of all, the CPL has been evaluated for service creation for the Internet Telephony, and it has been proved to be very suitable for service creation for end users. However, since the CPL is designed for un-trusted end users, provider specific services could not be implemented with the CPL. Fortunately, the extensibility of the CPL can be used to provide provider specific services for the Internet Telephony.

Secondly, the CPL has been evaluated if it is really easy to implement. With the help of already available tools, such as the XML Parser, this goal has also been achieved. Since the CPL is an XML based language, a very wide range of freely available tools could be used in the project. Although some of the functionalities of CPL were not implemented for the evaluation in this project, the time spent for implementation was considerably short (three man months).

Another important result was the easy integration of the SIP Server, the CPL Engine and the CPL User Editor. As it is aimed by the design of Session Initiation Protocol (SIP), extending the SIP with additional services created by the CPL was a very straight forward process, and comparably easy to implement.

6.6 Performance Measurements

The performance measurements have been made with the debug release of the corresponding software (e.g. the SIP Server and the CPL Engine). Normally, the debug release is slower than a normal release since in a normal release some compiler optimizations take place. For the performance measurements following environment has been used:

- 1 PC with Windows NT operating system. 256 MB RAM, 433 MHz Processor. The SIP Server, the CPL Engine and the Naming Service all ran on this PC.

- 4 IP Phones. Phone numbers: 198, 199, 200, and 201.

Four different scenarios were chosen for the measurements. A program, watching the network actions, was used to make the measurements. This program gets all the messages flowing through a specified TCP/IP port, and writes them into a file. The measurements have been recorded by checking this file. The measurement records are given in Table 0-1. A general call flow diagram for these scenarios can be seen in Fig. 0-1.

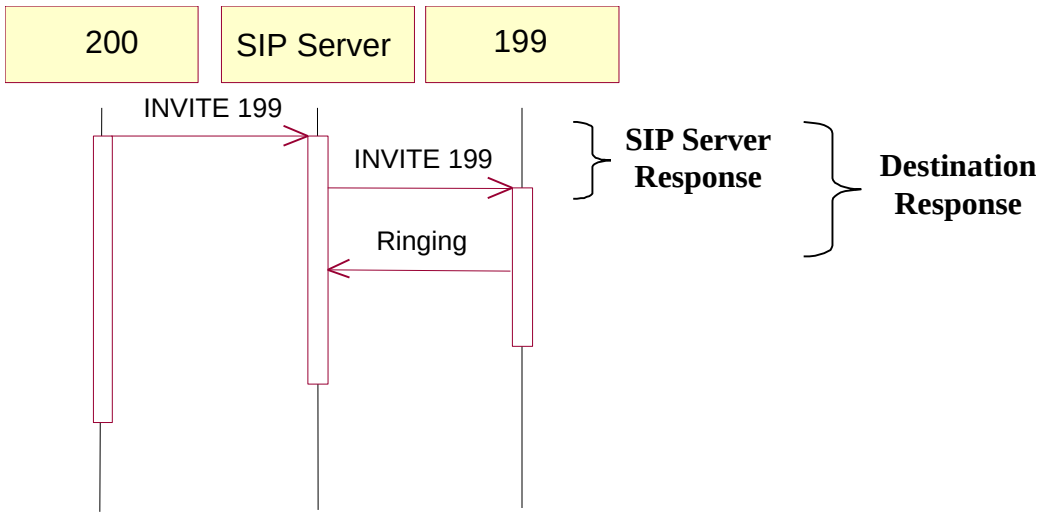

Fig. 0-1 Call flow diagram.

First scenario: Number 200 requests a call to number 199 and the SIP Server connects this call. The CPL Engine is not involved in this scenario since the user 199 does not have any active CPL Script.

Second scenario: Number 200 requests a call to number 201, and the SIP Server forwards this request to the location of 201. Again, the CPL Engine is not involved in this call establishment, since the user 201 does not have any active CPL script.

Third scenario: Number 200 requests a call to number 198. However, this time the user 198 has got an active CPL script. According to the script if the call is from the user's boss (200), then the call is connected to the office (199). If the call is from the user's partner, then it is connected to the current location (198). Since the calling party is 200, i.e. the user's boss, the call is connected to the office location (199). Fig. 0-2 shows the CPL script used during the performance measurements.

Fourth scenario: In this case, number 199 makes a call request for the number 198. Again the user has got the CPL script, but this time the origin is from the wife (199) and the call is directly connected to the current location (198).

```
<?xml version="1.0" encoding="iso-8859-1"?>
<cpl>
  <incoming>
    <address-switch field="origin" subfield="user">
      <address is="201">
        <mail url="198@surpass.com">
          <reject/>
        </mail>
      </address>
      <address is="200">
        <location url="sip:199@172.30.100.70" clear="yes">
          <proxy/>
        </location>
      </address>
      <otherwise>
        <location url="sip:198@172.30.100.70" clear="yes">
          <proxy/>
        </location>
      </otherwise>
    </address-switch>
  </incoming>
</cpl>
```

Fig. 0-2 The CPL script used during the performance tests.

In the first and second scenarios, it takes between 211 ms and 245 ms for the SIP Server to get the call request and forward it to the destination. The whole connection takes between 1119 ms and 1167 ms to be established.

In the third and fourth scenarios, where CPL Engine takes place, the time needed to respond a request is between 517 ms and 579 ms. This means that the time spent to process a CPL script is between 306 ms and 334 ms, and depending on the script length and specification it might take even longer. For example, in the fourth scenario a different location is specified for the proxy; and this different location causes the SIP Server to look for the IP address of the new location in the location database. As result, the SIP server spends some more time to respond the call request.

Shortly summarizing, the CPL Engine takes around 300 ms to process a CPL script, which is not a very good time comparing to the 211 ms SIP response time. However, considering that this is a prototyping project, and that the debug release of the software was used for performance measurements, it is an acceptable number. However, it can be improved with optimizations in the code.

As a general caution, the reader should keep in mind that the numbers given for the measurement results are average results. So, they can be larger or smaller for single trials.

Time	Origin	Destination	Action
Scenario 1 without tracing			
13:59:26:887	200	SIP Proxy	REQUEST to 199
13:59:27:132	SIP Proxy	199	REQUEST from 200
13:59:28:054	199	SIP Proxy	Ringing
Scenario 2 without tracing			
14:04:41:238	200	SIP Proxy	REQUEST to 201
14:04:41:449	SIP Proxy	201	REQUEST from 200
14:04:42:357	201	SIP Proxy	Ringing
Scenario 3 without tracing			
14:08:41:931	200	SIP Proxy	REQUEST to 198
14:08:42:510	SIP Proxy	199	REQUEST from 200
14:08:43:503	199	SIP Proxy	Ringing
Scenario 4 without tracing			
14:12:18:166	199	SIP Proxy	REQUEST to 198
14:12:18:683	SIP Proxy	198	REQUEST from 199
14:12:19:783	198	SIP Proxy	Ringing

Table 0-1 Performance measurements.

Table 0-2 summarizes the measurement results in a table. Fig. 0-3, additionally, illustrates this summary in a graphical view. It is clearly seen in this graphical view that addition of a CPL script introduces additional processing time for the SIP Server to respond a call request. This time depends on the length of the script and the tasks defined in the script. For example, more comparison tasks in the script increase the time required to process the script.

	SIP Response (ms)	Destination Response (ms)
Scenario 1	245	1167
Scenario 2	211	1119
Scenario 3	579	1572
Scenario 4	517	1617

Table 0-2 Summary of the performance measurements.

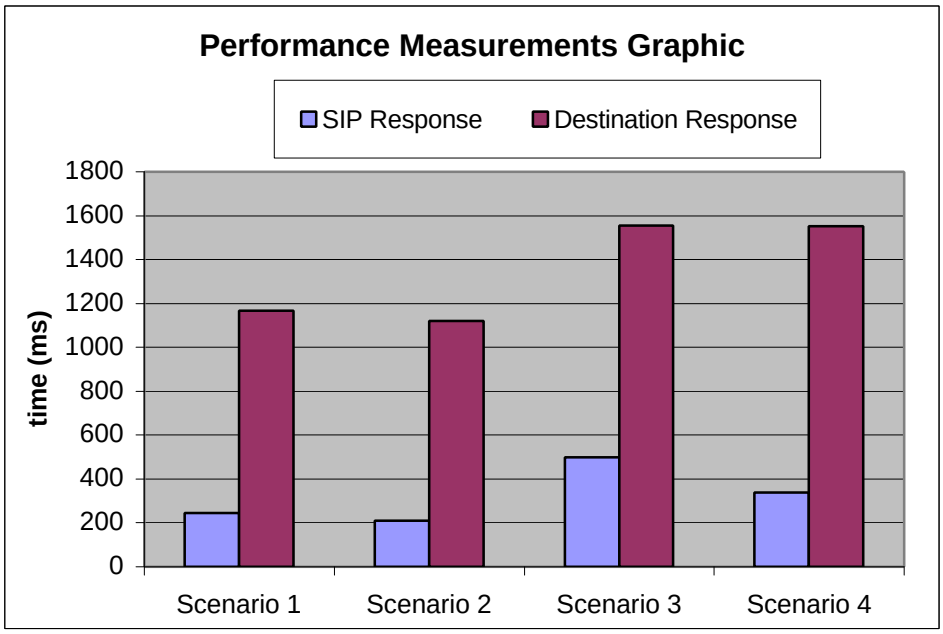

Fig. 0-3 Graphical summary of the performance measurements.

6.7 Suggestions for further studies

Four main points can be given as suggestions for the further studies: Extensibility of the CPL, not implemented parts of the CPL, asynchronous function calls through the CORBA interface and performance improvements.

Originally, CPL is designed for un-trusted end users to avoid any misusage of the server. Because of that some operator specific services cannot be implemented with the CPL. These services could be billing, user addition and removal, password changes for the users, and some other appropriate services necessary for the operators. Fortunately, as it has already been mentioned before, the CPL can be extended with some additional tags. To do that a new DTD has to be defined to cover new tags. Of course, these new tags should be merged into the CPL Engine and the CPL Editor, as well.

Another subject for further studies could be continuing with the evaluation of the CPL for not-implemented tags in this project. For example, time-switch and the location lookup were not implemented in this project because of the time restrictions.

Implementation and evaluation of the asynchronous function calls through the CORBA interface is another important topic to continue with. The function call in this project was synchronous, which created blocked functions waiting for a response from the CORBA interface. And, of course, this causes a performance drawback if multiple calls are made through the same server. Asynchronous function calls would avoid the waiting blocks, and the program would continue to process with the following call request. Because of the complications of asynchronous function calls, they have not been implemented in this project.

The last topic to continue with this project is to improve the performance. Again, because of the time restrictions code optimization was not done. Some code optimization could improve the performance significantly. Additionally, memory leaks could be checked and resolved, too.

7 Conclusions

Two kinds of conclusions came out of the project work carried on at Siemens AG in Munich, Germany: The developer aspect and the end user aspect. The developer aspect is based on the work done during the project work, while the end user aspect is based on the feedback received from the end users.

Considering the developer aspect, the CPL offers interesting new services for the Internet Telephony compared to IN for the classical telephony. Although the pricing schemes of the Internet Telephony is the most attractive advantage of the Internet Telephony for end users, the integration of the voice and data communications is another very important advantage of the Internet Telephony for future communication systems.

The ease of implementation of the CPL, which is based on XML, has also been proved by this project. The complete implementation of the CPL Engine and the CPL Repository took only three man months. Although there are some points to be improved in the CPL Engine, the most important functionality is offered by this first implementation.

Another attractive property of the CPL, being text based, has also been noticed to be very helpful for the graphical user interface development. The CPL User Editor has benefited from this property very positively.

Considering the end user aspect, the project received very positive feedback from the end users. The presentation at the International Fair in Hanover in Germany (CEBIT 2001) provided a good means of getting the end user attitude to the project. Beside normal end users, the telecom operators were very interested in the project, as well. Since the Internet Telephony is new in the market, the operators think that being the first to offer the additional services studied in this project would be a good challenge against their competitors. Due to such a good feedback, the project is going to be presented at another fair in Atlanta in USA (the Super-Com Fair).

Hopefully, this introductory work to create the additional services for the Internet Telephony using the CPL is going to attract the attention of developers and researchers to continue on improving the out comings of this project.

As a conclusion, the Call Processing Language has been proved to be very suitable for creating the additional services for the Internet Telephony.

Appendices

Appendix A: CPL Tags

Bold tags in the following list represent the tags implemented in this project.

- **cpl**
  - ancillary (not implemented)
  - **subaction**
    Parameters: id
  - outgoing (future work)
  - **incoming**
- **address-switch**
  Parameters:
  Fields: origin, destination, original-destination
  Subfields: address-type, user, host, port, tel, display
  - **address**
    Parameters: is, contains, subdomain-of
- **string-switch**
  Parameters:
  Fields: subject, organization, user-agent, language, display
  - **string**
    Parameters: is, contains
- time-switch (future work)
  Parameters: tzid, tzurl
  - time
    Parameters: dtstart, dtend, duration, freq, interval, until, byday, bymonthday, byyearday, byweekno, bymonth, wkst
- priority-switch (not implemented)
  - priority
    Parameters: less, greater, equal
- **location**
  Parameters: url, priority, clear
- lookup (not implemented)
  Parameters: source, timeout, use, ignore, clear
  - success
  - notfound
  - failure
- **remove-location**
  Parameters: location, param, value
- **proxy**
  Parameters: timeout, recurse, ordering
  - **busy**
  - **noanswer**
  - **redirection**
  - **failure**
  - **default**
- **redirect**
  Parameters: permanent
- **reject**
  Parameters: status, reason

- **mail**
  Parameters: url
- **log**
  Parameters: name, comment
- **sub**
  Parameters: ref
- extension-switch (future work)

Appendix B: CPL DTD

```
<?xml version="1.0" encoding="US-ASCII" ?>
<!--
Draft DTD for CPL, corresponding to
draft-ietf-iptel-cpl-01.
-->
<!-- Nodes. -->
<!-- Switch nodes -->
<!ENTITY % Switch 'address-switch|string-switch|time-switch|
          priority-switch' >
<!-- Location nodes -->
<!ENTITY % Location 'location|lookup|remove-location' >
<!-- Signalling action nodes -->
<!ENTITY % SignallingAction 'proxy|redirect|reject' >
<!-- Other actions -->
<!ENTITY % OtherAction 'mail|log' >
<!-- Links to subactions -->
<!ENTITY % Sub 'sub' >
<!-- Nodes are one of the above four categories, or a subaction.
This entity (macro) describes the contents of an output.
Note that a node can be empty, implying default action. -->
<!ENTITY % Node     '(%Location;|%Switch;|%SignallingAction;|
           %OtherAction;|%Sub;)?' >
<!-- Switches: choices a CPL script can make. -->
<!-- All switches can have an 'otherwise' output. -->
<!ELEMENT otherwise ( %Node; ) >
<!-- All switches can have a 'not-present' output. -->
<!ELEMENT not-present ( %Node; ) >
<!-- Address-switch makes choices based on addresses. -->
<!ELEMENT address-switch ( (address|not-present)+, otherwise? ) >
<!-- <not-present> must appear at most once -->
<!ATTLIST address-switch
field       CDATA    #REQUIRED
subfield    CDATA    #IMPLIED
>
<!ELEMENT address ( %Node; ) >
<!ATTLIST address
is          CDATA    #IMPLIED
contains    CDATA    #IMPLIED
subdomain-of  CDATA    #IMPLIED
```

```
> <!-- Exactly one of these three attributes must appear -->
<!-- String-switch makes choices based on strings. -->
<!ELEMENT string-switch ( (string|not-present)+, otherwise? ) >
<!-- <not-present> must appear at most once -->
<!ATTLIST string-switch
field        CDATA    #REQUIRED
>
<!ELEMENT string ( %Node; ) >
<!ATTLIST string
is           CDATA    #IMPLIED
contains     CDATA    #IMPLIED
>  <!-- Exactly one of these two attributes must appear -->
<!-- Time-switch makes choices based on the current time. -->
<!ELEMENT time-switch ( (time|not-present)+, otherwise? ) >
<!ATTLIST time-switch
tzid         CDATA    #IMPLIED
tzurl        CDATA    #IMPLIED
>
<!ELEMENT time ( %Node; ) >
<!-- Exactly one of the two attributes "dtend" and "duration"
must occur. -->
<!-- The value of "freq" is (daily|weekly|monthly|yearly).  It is
   case-insensitive, so it is not given as a DTD switch. -->
<!-- None of the attributes following freq are meaningful unless freq
    appears. -->
<!-- The value of "wkst" is (MO|TU|WE|TH|FR|SA|SU).  It is
   case-insensitive, so it is not given as a DTD switch. -->
<!ATTLIST time
dtstart      CDATA  #REQUIRED
dtend        CDATA  #IMPLIED
duration     CDATA  #IMPLIED
freq         CDATA  #IMPLIED
until        CDATA  #IMPLIED
interval     CDATA  "1"
byday        CDATA  #IMPLIED
bymonthday   CDATA  #IMPLIED
byyearday    CDATA  #IMPLIED
byweekno     CDATA  #IMPLIED
bymonth      CDATA  #IMPLIED
wkst         CDATA  "MO"
>
<!-- Priority-switch makes choices based on message priority. -->
<!ELEMENT priority-switch ( (priority|not-present)+, otherwise? ) >
<!-- <not-present> must appear at most once -->
<!ENTITY % PriorityVal '(emergency|urgent|normal|non-urgent)' >
<!ELEMENT priority ( %Node; ) >
<!-- Exactly one of these three attributes must appear -->
<!ATTLIST priority
less         %PriorityVal;  #IMPLIED
greater      %PriorityVal;  #IMPLIED
equal        CDATA          #IMPLIED
```

```
>
<!-- Locations: ways to specify the location a subsequent action
(proxy, redirect) will attempt to contact. -->
<!ENTITY % Clear  'clear (yes|no) "no"' >
<!ELEMENT location ( %Node; ) >
<!ATTLIST location
url          CDATA    #REQUIRED
priority     CDATA    #IMPLIED
%Clear;
>
<!ELEMENT lookup ( success,notfound?,failure? ) >
<!ATTLIST lookup
source       CDATA     #REQUIRED
timeout      CDATA     "30"
use          CDATA     #IMPLIED
ignore       CDATA     #IMPLIED
%Clear;
>
<!ELEMENT success  ( %Node; ) >
<!ELEMENT notfound ( %Node; ) >
<!ELEMENT failure ( %Node; ) >
<!ELEMENT remove-location ( %Node; ) >
<!ATTLIST remove-location
param        CDATA    #IMPLIED
value        CDATA    #IMPLIED
location     CDATA    #IMPLIED
>
<!-- Signalling Actions: call-signalling actions the script can
take. -->
<!ELEMENT proxy ( busy?,noanswer?,redirection?,failure?,default? ) >
<!-- The default value of timeout is "20" if the <noanswer> output
exists. -->
<!ATTLIST proxy
timeout      CDATA    #IMPLIED
recurse      (yes|no) "yes"
ordering     CDATA    "parallel"
>
<!ELEMENT busy ( %Node; ) >
<!ELEMENT noanswer ( %Node; ) >
<!ELEMENT redirection ( %Node; ) >
<!-- "failure" repeats from lookup, above. -->
<!ELEMENT default ( %Node; ) >
<!ELEMENT redirect EMPTY >
<!ATTLIST redirect
permanent    (yes|no) "no"
>
<!-- Statuses we can return -->
<!ELEMENT reject EMPTY >
<!-- The value of "status" is (busy|notfound|reject|error), or a SIP
4xx-6xx status. -->
<!ATTLIST reject
```

```
status        CDATA    #REQUIRED
reason        CDATA    #IMPLIED
>
<!-- Non-signalling actions: actions that don't affect the call -->
<!ELEMENT mail ( %Node; ) >
<!ATTLIST mail
url           CDATA    #REQUIRED
>
<!ELEMENT log ( %Node; ) >
<!ATTLIST log
name          CDATA    #IMPLIED
comment       CDATA    #IMPLIED
>
<!-- Calls to subactions. -->
<!ELEMENT sub EMPTY >
<!ATTLIST sub
ref           IDREF    #REQUIRED
>
<!-- Ancillary data -->
<!ENTITY % Ancillary 'ancillary?' >
<!ELEMENT ancillary EMPTY >
<!-- Subactions -->
<!ENTITY % Subactions 'subaction*' >
<!ELEMENT subaction ( %Node; )>
<!ATTLIST subaction
id            ID       #REQUIRED
>
<!-- Top-level actions -->
<!ENTITY % TopLevelActions 'outgoing?,incoming?' >
<!ELEMENT outgoing ( %Node; )>
<!ELEMENT incoming ( %Node; )>
<!-- The top-level element of the script. -->
<!ELEMENT cpl  ( %Ancillary;,%Subactions;,%TopLevelActions; ) >
```

Appendix C: The IDL definition of the CORBA interface

```
module Cpl {

  // ------------------------------------------------------------
  // modes used in both directions
  // ------------------------------------------------------------

  // SIP Call-ID
  typedef long TransactionIdType;

  // modes for address switches
  typedef string UserAddressType;

  // Call-leg Identifier used to identify call
  struct CallLegIdType
```

```
{
  TransactionIdType transactionId;                // SIP Call-ID
  UserAddressType originUserAddress; // SIP From header
  UserAddressType originalDestinationUserAddress; // SIP To header
};

// mode for CORBA communication result
enum ResultSet {
    OK,
    ERROR_OUT
};

// ---------------------------------------------------------------
// modes for SIPproxy -> CPLengine
// ---------------------------------------------------------------

// modes for string switches
struct StringParameterType
{
  string name;                // Name of string
  boolean present;            // string provided?
};

struct AdditionalStringInformationType
{
  boolean present;                // at least one value filled out?
  StringParameterType subject;      // SIP-header field subject
  StringParameterType organization; // SIP-header field organization
  StringParameterType user_agent;   // SIP-header field user-agent
  StringParameterType language;     // SIP-header field Accecpt-Language
                                    // converted to RFC 1766 conformant comma-separated list
  StringParameterType display;      // not used, present=false
};

// modes for priority switches
typedef float PriorityType;


// modes for SIPproxy -> CPLengine reply to a proxy request
enum ProxyResponseSet {
  no_answer,      // timeout
  busy,           // busy received (486 or 600)
  redirection,    // 3xx received
  failure         // all the rest not handled above: 4xx, 5xx, or 6xx
};

// ---------------------------------------------------------------
// modes for CPLengine -> SIPproxy
// ---------------------------------------------------------------

typedef long TimeoutType;
```

```
typedef string RejectReasonType;
typedef sequence<UserAddressType> LocationListType; // we need a list!
typedef string LocationDataBaseInfoType;

// mode for specifying the type of redirect mode
enum RedirectRequestSet {
  permanent,
  temporary      // default
};

// reject status
typedef short RejectStatusType;

// ------------------------------------------------------------
// interfaces
// ------------------------------------------------------------

// ************************************************************
// interface CPLengine -> SIPproxy
// ************************************************************
interface SipProxyServer {

  exception CannotProxy {
         string reason;
        };

  // SIP proxy should proxy the call
  ResultSet proxy (
         in CallLegIdType callLegId,        // SIP Call-ID + From header + To header
         in LocationListType locationList,  // where should the redirect point to?
         in TimeoutType timeOut             // specified timeout in seconds, default=20 (s)
        )
        raises (CannotProxy);

  // SIP proxy should redirect the call
  ResultSet redirect (
         in CallLegIdType callLegId,        // SIP Call-ID + From header + To header
         in LocationListType locationList,  // where should the redirect point to?
         in RedirectRequestSet redirectMode // permanent or non_permanent (default)
        );

  // SIP proxy should reject the call
  ResultSet reject (
         in CallLegIdType callLegId,        // SIP Call-ID + From header + To header
         in RejectStatusType rejectStatus,   // what's the reason for rejecting
         inout RejectReasonType rejectReason   // optional additional reason
        );

  // SIP proxy should continue with the default behaviour (as if no CPL was there)
  ResultSet continueSipDefault (
         in CallLegIdType callLegId         // SIP Call-ID + From header + To header
```

```
        );
};


// ************************************************************
// interface SIPproxy -> CPLengine
// ************************************************************
interface CallServer {

  // Start the CPL treatment for one call
  ResultSet startCall (
        in SipProxyServer sipServer,    //reference for calling back
        in CallLegIdType callLegId,     // SIP Call-ID + From header + To header

        // parameters needed for address switching
        inout UserAddressType destination_user_address,
                // Request-URI (e.g. address in the same header as INVITE)

        // parameters needed for string switching
        in AdditionalStringInformationType additionalStringInformation, //see type definition

        // parameters needed for priority switching
        in PriorityType priority,  // SIP priority header in initial INVITE
        inout LocationDataBaseInfoType locationDataBaseCplInfo
       );

  // Continue the CPL treatment for one call (after proxy)
  ResultSet continueCall (
        in SipProxyServer sipServer,          //reference for calling back
        in CallLegIdType callLegId,       // SIP Call-ID + From header + To header
        in ProxyResponseSet proxyResponse  // filled out by SIP-proxy
       );

  // 200 OK received, tell this to the CPL engine in order to stop CPL for that call
  ResultSet destroyCall (
        in CallLegIdType callLegId        // SIP Call-ID + From header + To header
       );
};

};
```

Appendix D The SIP response messages

- 1xx: Informational message
  - ⇒ 100: Trying
  - ⇒ 180: Ringing
  - ⇒ 181: Call is being forwarded
  - ⇒ 182: Queued
- 2xx: Successful
  - ⇒ 200: OK

- 3xx: Redirection
    - ⇒ 300: Multiple choices
    - ⇒ 301: Moved Permanently
    - ⇒ 302: Moved Temporarily
    - ⇒ 303: Use Proxy
    - ⇒ 380: Alternative Service
- 4xx: Request failure
    - ⇒ 400: Bad request
    - ⇒ 401: Unauthorized
    - ⇒ 402: Payment Required
    - ⇒ 403: Forbidden
    - ⇒ 404: Not Found
    - ⇒ 405: Method Not Allowed
    - ⇒ 406: Not Acceptable
    - ⇒ 407: Proxy Authentication Required
    - ⇒ 408: Request Timeout
    - ⇒ 409: Conflict
    - ⇒ 410: Gone
    - ⇒ 411: Length Required
    - ⇒ 413: Request Entity Too Large
    - ⇒ 414: Request-URI Too Long
    - ⇒ 415: Unsupported Media Type
    - ⇒ 420: Bad Extension
    - ⇒ 480: Temporarily Unavailable
    - ⇒ 481: Call Leg/Transaction Does Not Exist
    - ⇒ 482: Loop Detected
    - ⇒ 483: Too Many Hops
    - ⇒ 484: Address Incomplete
    - ⇒ 485: Ambiguous
    - ⇒ 486: Busy Here
- 5xx: Server Failure
    - ⇒ 500: Server Internal Error
    - ⇒ 501: Not Implemented
    - ⇒ 502: Bad Gateway
    - ⇒ 503: Service Unavailable
    - ⇒ 504: Gateway Time-out
    - ⇒ 505: Version Not Supported
- 6xx: Global Failures
    - ⇒ 600: Busy Everywhere
    - ⇒ 603: Decline
    - ⇒ 604: Does Not Exist Anywhere
    - ⇒ 606: Not Acceptable

List of Figures

List of Tables

References

[1] Apache Xerces C++ Parser, http://xml.apache.org/xerces-c/index.html

[2] Henning Mitchi, Vinoski Steve, Advanced CORBA Programming with C++, Addision Wesley, 1999

[3] XML, http://www.w3.org/XML/

[4] SIP: Session Initiation Protocol. M. Handley, H. Schulzrinne, E. Schooler, J. Rosenberg. March 1999. http://www.ietf.org/rfc/rfc2543.txt

[5] Call Processing Language Framework and Requirements. J. Lennox, H. Schulzrinne. May 2000. http://www.ietf.org/rfc/rfc2824.txt

[6] CPL: A Language for User Control of Internet Telephony Services, J. Lennox, H. Schulzrinne, http://www.ietf.org/internet-drafts/draft-ietf-iptel-cpl-04.txt

[7] LDAP (Lightweight Directory Access Protocol), M. Wahl, T. Howes, S. Kille, December 1997, http://www.ietf.org/rfc/rfc2251.txt

[8] Lightweight Directory Access Protocol (v3): Attribute Syntax Definitions. M. Wahl, A. Coulbeck, T. Howes, S. Kille. December 1997. http://www.ietf.org/rfc/rfc2252.txt

[9] LDAP: UTF-8 String Represantation of Ditinguished Names, M. Wahl, S. Kille, T. Howes, December 1997. http://www.ietf.org/rfc/rfc2253.txt

[10] The String Represantation of LDAP Search Filters, T. Howes, December 1997, http://www.ietf.org/rfc/rfc2254.txt

[11] The LDAP URL Format, T. Howes, M. Smith, December 1997, http://www.ietf.org/rfc/rfc2255.txt

[12] Intelligent Network (IN), http://www.iec.org/tutorials/in/index.html

[13] Internation Intelligent Network (IIN), http://www.iec.org/tutorials/intern_in/index.html

[14] Intelligent network (IN) Service Creation, http://www.iec.org/tutorials/in_serv/index.html

[15] Teach Yourself CORBA in 14 Days, Jerry J. Zhang, University of New Brunswick, http://www.omg.unb.ca/~jzhang/corba/

[16] Document Object Model (DOM) Level 2 Core Specification, Arnaud Le Hors et al., http://www.w3.org/TR/DOM-Level-2-Core/

[17] The Simple API for XML, http://www.megginson.com/SAX/index.html

[18] Session Initiation Protocol (SIP), http://www.cs.columbia.edu/~hgs/sip

[19] Hypertext Markup Language - 2.0, http://www.ietf.org/rfc/rfc1866.txt

[20] Future Interaction in Internet Telephony, Jonathan Lennox, Henning Schulzrinne, Columbia University

[21] International Telecommunications Union, “Packet based multimedia communications systems” Recommendation H.323, Telecommunication Standardization Sector of ITU, Geneva, Switzerland, Fe. 1998, http://www.itu.int/itudoc/itu-t/rec/h/h323.html

[22] Standard Generalized Markup Language (SGML), http://www.iso.ch/cate/d16387.html

[23] R. Cover, “The SGML / XML Web Page”, http://www.oasis-open.org/cover/sgml-xml.html

[24] HTML, W3C’s HTML Home Page, http://www.w3.org/MarkUp/

[25] “Advanced Services Architectures for Internet Telephony: A Critical Overview”, Roch H. Glitho, Ericsson Research Canada, IEEE Network, July/August 2000, p. 38-44

[26] “The Internet: A Global Telecommunications Solution?”, Laurent Mathy, Christopher Edwards, and David Huchison, Lancaster University, IEEE Network, July/August 2000, p. 46-57

[27] “Programming Internet Telephony Services”, Jonathan Rosenberg, Bell Laboratories, Jonathan Lennox and Henning Schulzrinne, Columbia University, IEEE Network, May/June 1999, p. 42-49

[28] The web site of CPL Editor developers (Humbolt University, Berlin), http://www.informatik.hu-berlin.de/~xing/CPLEditor/

[29] Guidelines for Authors of SIP Extensions, J.Rosenberg, Dynamicsoft, H.Schulzrinne, Columbia University, March 2001, http://www.ietf.org/internet-drafts/draft-ietf-sip-guidelines-02.txt

[30] Hear Me SW Phone, downloadable from http://www.hearme.com/applications/sipphones/regform.html

[31] Object Management Group (OMG)’s Web site, http://www.omg.org/

[32] Object Management Group (OMG)’s CORBA Web Site, http://www.corba.org/

[33] The Adaptive Communication Environment (ACE) Research Group’s Web Site, http://www.cs.wustl.edu/~schmidt/ACE.html

[34] World Wide Web Consortium (W3C)’s Web site, http://www.w3.org/